

THE EXPERT'S VOICE® IN OPEN SOURCE

The Definitive Guide to
GCC

Kurt Wall
and William von Hagen

APress Media, LLC

The Definitive Guide to GCC

KURT WALL AND WILLIAM VON HAGEN

APress Media, LLC

The Definitive Guide to GCC
Copyright ©2004 by Kurt Wall and William von Hagen
Originally published by Apress in 2004

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN 978-1-59059-109-3
DOI 10.1007/978-1-4302-0704-7

ISBN 978-1-4302-0704-7 (eBook)

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Gene Sally

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Production Editor: Janet Vail

Proofreader: Elizabeth Berry

Compositor and Artist: Kinetic Publishing Services, LLC

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

*To my teachers, Miss Rhodes, Mrs. Walker, Mrs. Burnette, Miss Nearhoof,
Mrs. Conley, Mrs. Lee, Mr. Bundy, and Mrs. Dunn; and to the schools in which
they worked their magic: Randolph, Madison Pike Elementary School,
Westlawn Junior High School, and S.R. Butler High School.*

—Kurt Wall

*For Dorothy Fisher—I couldn't do it without you!
And for Becky Gable—thanks for all your help with the schematics
over the years!*

—Bill von Hagen

Contents at a Glance

About the Authors	<i>xi</i>
About the Technical Reviewer	<i>xii</i>
Acknowledgments	<i>xiii</i>
Introducing GCC and The Definitive Guide to GCC	<i>xv</i>
Chapter 1 Building GCC	1
Chapter 2 Installing GCC on DOS and Windows Platforms	39
Chapter 3 Basic GCC Usage	59
Chapter 4 Advanced GCC Usage	101
Chapter 5 Optimizing Code with GCC	135
Chapter 6 Performing Code Analysis with GCC	147
Chapter 7 Using Autoconf and Automake	187
Chapter 8 Using Libtool	221
Chapter 9 Troubleshooting GCC	245
Chapter 10 Using GCC's Online Help	265
Chapter 11 GCC Command-Line Options	283
Chapter 12 Additional GCC Resources	347
Appendix A Building and Installing Glibc	363
Appendix B Machine and Processor-Specific Options for GCC	403
Index	497

Contents

About the Authors	<i>xi</i>
About the Technical Reviewer.....	<i>xii</i>
Acknowledgments	<i>xiii</i>
Introducing GCC and The Definitive Guide to GCC	<i>xv</i>
Chapter 1 Building GCC.....	<i>1</i>
Why Build GCC from Source?	<i>1</i>
What Is New in GCC 3?	<i>24</i>
Chapter 2 Installing GCC on DOS and Windows Platforms.....	<i>39</i>
Installing Cygwin	<i>39</i>
Installing DJGPP	<i>47</i>
Chapter 3 Basic GCC Usage	<i>59</i>
General Options	<i>59</i>
Controlling GCC's Output	<i>62</i>
Compiling C Dialects	<i>69</i>
Using GCC with C++	<i>74</i>
Controlling the Preprocessor.....	<i>77</i>
Modifying the Directory Search Path	<i>78</i>
Controlling the Linker	<i>82</i>
Passing Options to the Assembler	<i>86</i>
Enabling and Disabling Warning Messages	<i>86</i>
Adding Debugging Information.....	<i>94</i>

Chapter 4 Advanced GCC Usage	101
Customizing GCC Using Environment Variables	101
Customizing GCC with Spec Files and Spec Strings	103
GCC's C and Extensions	108
Chapter 5 Optimizing Code with GCC	135
A Whirlwind Tour of Compiler Optimization Theory	135
Processor-Independent Optimizations	138
Processor-Specific Optimizations	145
Chapter 6 Performing Code Analysis with GCC	147
Test Coverage Using GCC and gcov	148
Chapter 7 Using Autoconf and Automake	187
Unix Software Configuration, Autoconf, and Automake	187
Installing and Configuring Autoconf and Automake	191
Configuring Software with Autoconf and Automake	200
Running configure Scripts	218
Chapter 8 Using Libtool	221
Introduction to Libraries	221
What Is Libtool?	226
Downloading and Installing Libtool	228
Using Libtool	232
Troubleshooting Libtool Problems	243
Getting More Information About Libtool	244
Chapter 9 Troubleshooting GCC	245
Coping with Known Bugs and Misfeatures	245
Resolving Common Problems	247
Resolving Build and Installation Problems	262

Chapter 10 Using GCC's Online Help.....	265
What Is GNU Info?	265
Getting Started, or Instructions for the Impatient	267
The Beginner's Guide to Using GNU Info	269
Stupid Info Tricks	279
Chapter 11 GCC Command-Line Options	283
Alphabetical List of GCC Options	284
Alphabetical List of GCC Assembler Options	336
Alphabetical List of GCC Linker Options	336
Alphabetical List of GCC Preprocessor Options	338
GCC Option Reference	344
Chapter 12 Additional GCC Resources	347
Usenet Resources for GCC	347
Mailing Lists for GCC	352
World Wide Web Resources for GCC	358
Publications About GCC and Related Topics	359
Appendix A Building and Installing Glibc	363
What Is in Glibc?	363
Alternatives to Glibc	365
Why Build Glibc from Source?	367
Previewing the Build Process	374
Recommended Tools for Building Glibc	376
Downloading and Installing Source Code	379
Configuring the Source Code	385
Compiling Glibc	388
Testing the Build	390
Installing Glibc	390
Getting More Information About Glibc	400
Appendix B Machine and Processor-Specific Options for GCC	403
Alpha Options	405
Alpha/VMS Options	411

AMD x86-64 Options	412
AMD29K Options	413
ARC Options	415
ARM Options	416
AVR Options	422
Clipper Options	424
Convex Options	424
CRIS Options	425
D30V Options	428
H8/300 Options	429
HP/PA (PA/RISC) Options	430
i386 and AMD x86-64 Options	432
IA-64 Options	438
Intel 960 Options	440
M32R Options	442
M680x0 Options	443
M68hc1x Options	447
M88K Options	447
MCore Options	451
MIPS Options	452
MMIX Options	460
MN10200 Options	461
MN10300 Options	462
NS32K Options	462
PDP-11 Options	464
PowerPC (PPC) Options	466
RS/6000 Options	478
RT Options	478
S/390 and zSeries Options	479
SH Options	480
SPARC Options	482
System V Options	487
TMS320C3x/C4x Options	488
V850 Options	491
VAX Options	492
Xstormy16 Options	492
Xtensa Options	492
Index	497

About the Authors



Kurt Wall first touched a computer in 1980 when he learned FORTRAN on an IBM mainframe of forgotten vintage; happily, computer technology has improved considerably since then. A professional technical writer by trade and a historian by training, Kurt has a diverse working history. These days, Kurt works for TimeSys Corporation in Pittsburgh, Pennsylvania. His primary responsibility is managing TimeSys' Content Group, or as it is known in-house, the artists' colony. In addition to directing production of the technical and end-user documentation for TimeSys' embedded Linux operating system and development tools, he also writes much of the documentation for TimeSys' embedded Linux products. Kurt has written all or parts of 15 books on Linux system administration and programming topics. In his spare time . . . he has no spare time. Kurt, who dislikes writing about himself in the third person, receives entirely too much e-mail at kwall@kurtwerks.com.



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, and (now) product manager. Bill has written a number of books including *Hacking the TiVo*, *Linux Filesystems*, *Installing Red Hat Linux 7*, and *SGML for Dummies*, contributed to *Red Hat 7 Unleashed*, and coauthored the *Mac OS X Power User's Guide* with Brian Proffitt. Bill has written articles and software reviews for publications including *Linux Magazine*, *Mac Tech*, *Linux Format* (UK), and *Mac Directory*. He has also written extensive online material for CMP Media, Linux Planet, and Corel. 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 worked on a variety of software projects, from accounting to network management. Gene stumbled into Linux when setting up the network infrastructure for a former employer and hasn't looked back since. When not working, Gene enjoys being tackled by his three children.

Acknowledgments

THANKS TO MY COAUTHOR. I brought Bill in to help me meet the original schedule and because we thought it would be great fun to do a book together. We failed, miserably, to meet the original schedule, but we had fun, anyway. Thanks, Bill! Speaking of schedules, or rather, of our total disregard for them, our editors at Apress were heroically patient as deadlines came and went with great vigor and distressing frequency. Thanks to Martin Streicher and Sofia Marchant. Ami Knox receives more-than-honorable mention for catching and fixing all sorts of amusing diction, grammar, spelling, and syntax mistakes and abuses. Gene Sally, our technical editor, made *many* suggestions that have greatly improved the book. He volunteered for this abuse, not knowing what he was doing, and might never recover. Thanks, Gene.

Kudos to Marta Justak, principal of Justak Literary Services and my agent. Marta has always represented me well, and this book was no exception. Even while she was throwing crystal vases on the floor, Marta managed to keep me on task. Words fail to express my gratitude, Marta.

I work with an incredibly talented group of people at TimeSys. I learn something new from them all every day. In particular, these people deserve due credit for their friendship and support: Bill “Hot Sauce” von Hagen, my former boss; and my Trusty Minions, Erin “Label Mamma” Kelly, Michaela “Eeyore” Jencka, and Kate “We Don’t Need No Steenking GUI” Tabasko. Everyone should have the opportunity to work with people they truly like. Thanks to you all!

Above all, if I have any talent as a writer, credit goes to the God: He gave me the talent, provided me the opportunities to develop and use it, and kept me sober long enough to do so. Thanks, and Amen.

—Kurt Wall

I'D LIKE TO THANK KURT for the opportunity to work with him on this book and at TimeSys, and Marta Justak, my agent as well, for her support and friendship. I'd also like to thank Gene Sally and Ami, Martin, Sofia, and others at Apress for making this book much better than it could have been without their participation. I'd also like to thank Richard Stallman and the FSF for GCC, Emacs (the one true editor), and their other countless contributions to computing as we know it today. Without your foresight and philosophy, this book wouldn't even exist. I'd also like to thank rms for some way cool LMI hacks long ago.

—Bill von Hagen

Introducing GCC and *The Definitive Guide* to *GCC*

THIS BOOK, *The Definitive Guide to GCC*, is about how to build, install, customize, use, and troubleshoot GCC 3.x, the GNU Compiler Collection version 3.x. GCC has long been available for most major hardware and operating system platforms and is often the preferred compiler for those platforms, at least among users. As a general purpose compiler, 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, is one of the showpieces of the free software world and proof that sometimes you *can* get a free lunch.

Why a Book About GCC?

We wrote this book, and you should read it, for a variety of reasons: it covers version 3.x; it is the only book that covers general GCC usage; and it is better than GCC's own documentation. You will not find more complete coverage of its features, quirks, and usage anywhere else collected in a single volume. Moreover, save for GCC's own documentation, when we first started writing, no existing book was dedicated solely to using GCC—at most it got one or two chapters in programming books and only a few paragraphs, perhaps an entire section, in other, more general titles. GCC's existing documentation, although thorough and comprehensive, targets a programming-savvy reader. Although there is nothing wrong with this approach—indeed, it is the proper approach for advanced users—GCC's own documentation leaves the great majority of its users out in the cold. Fully half 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. Showing you how to use GCC is this book's primary goal, but we also hope to explain how GCC works without descending into the minutiae of compiler theory.

The Definitive Guide to GCC is the first and only title that provides a general GCC reference and tutorial. Not surprisingly, our own curiosity provided the original motivation to write about GCC—we suspected that we were barely

scratching the surface of GCC’s capabilities, and it did not take us long to discover that we were right. Most people, including many 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, shove source files in one end, and then receive a compiled, functioning program from the other end. Just as often, many end users, particularly nonprogrammers, use higher level tools, such as the `make` utility or RPM. Such tools make GCC easier to use because they hide GCC behind a simpler interface. Of course, there is nothing wrong with this utilitarian approach. RPM, `make`, and other build automation tools are valuable and essential tools, but they have the unfortunate side effect of hiding GCC’s power and flexibility. So, another goal we had in mind when writing *The Definitive Guide to GCC* was to remove the obscurity, show the power, reveal what the high-level tools like `make` and RPM do behind the scenes, and, in the process, give you the opportunity to understand what the options you use by rote when invoking GCC directly actually mean and do.

Inveterate tweakers, incorrigible tinkers, 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 without harming your existing system.

What You Will Learn

The Definitive Guide to GCC follows the standard tutorial format, proceeding from simple to complex, each chapter building on the material in preceding chapters. You will start with basic GCC invocation for compiling C and C++ source code, move on to intermediate operations such as controlling the compilation process and defining the type of output GCC produces, and finish up with advanced topics such as code optimization, test coverage, and profiling. You will also 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. Programmers and advanced or merely inquisitive readers will appreciate exploring the extensions to the C and C++ programming languages that GCC supports. Developers will also discover how to use libtool and the GNU autoconfiguration tools to simplify configuring software to build with GCC. 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 GCC. 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, we have assumed that most of you are using one version or another of the Linux operating system running on Intel x86 and compatible system. Nevertheless, most of the material will be GCC specific, rather than 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 GCC is a command-line compiler. Accordingly, you should 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, too, 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, you will be familiar with the terminology and processes constantly mentioned in the text. If, on the other hand, this is your first foray into working with source code, Chapters 2 and 3 will get you up and running quickly. You do not need to be a programmer, know how to write C or C++, or how to do your taxes in hexadecimal. If you have experience using a compiled programming language, though, this experience will help you. The only other prerequisite you need in order to benefit from this book is access to a system with GCC, but not even *that* requirement is absolute—jump ahead to Chapter 1 to find out how to download, build, and install GCC from scratch using your system's native compiler.

You should 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 or C++, although the examples used to demonstrate compiler features are written mostly in C (with a few examples using C++). Similarly, the text uses the make utility to simplify compilation, but using make is not covered beyond explaining the examples. Strictly speaking, as you will learn in the next chapter, 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, but the coverage in this book is generally limited to GCC’s usage for C and C++, which reflects its use by the vast majority of its users. Compiler theory also gets short shrift in this book, because 98 percent of you could care less about compiler guts; the other 2 percent of you will have to satisfy yourselves with the limited material describing GCC’s architecture and overall compilation workflow. That said, it is difficult to talk about using a compiler without skimming the surface of compiler theory and operation, so you will learn a few key terms and concepts as necessary.

The Definitive Guide to GCC, Chapter by Chapter

This introduction justifies our existence, or at least why you should buy the book. Seriously, the first half of this introduction details what you will learn, or at least what it proposes to explain, the assumptions made about your experience and knowledge of things digital, and the topics not covered for reasons of space or relevance. It also briefly describes the contents of each chapter. The second half of this introduction looks a little harder at what GCC is and does and includes the obligatory history of GCC. GCC being one of the GNU project’s marquis products, the final pages of this introduction discuss GCC’s development model in order to help you understand why it has the features it has, why it lacks other features, how to submit bug reports, and how you can participate in its development, if you are so inclined.

Conventional wisdom is that building and installing GCC from scratch should not be undertaken except by experts because of the possibility of rendering your system an inert lump of plastic, copper, and silicon. While compiling GCC from scratch clearly falls into the nontrivial task category, you will learn in Chapter 1, “Building GCC,” that conventional wisdom has led you astray: building GCC from source is considerably simpler and more straightforward than commonly believed, due in large part to the excellent work of GCC’s developers in automating the process. Chapter 1 shows you how to download, compile, test, and install GCC 3.x on a Unix or Linux system. Because the 3.x release represents such a milestone in GCC development, Chapter 1 also devotes considerable time and space to exploring GCC 3.x’s new features and to highlighting changes and improvements relative to version 2.95. In a similar vein, Chapter 2, “Installing GCC on DOS and Windows Platforms” shows you how to get GCC 3.x up and running on a DOS or Windows system using DJGPP, Cygwin, and MinGW.

NOTE *Although Chapter 1 spotlights the shiny new GCC wagon, discussion of new and changed features is peppered throughout The Definitive Guide to GCC. Everyone should read and heed the caveats, gotchas, qualifications, and disclaimers pertaining to the 3.x release to avoid weeping, wailing, and gnashing of teeth*

Chapter 3, “Basic GCC Usage,” describes the fundamentals of using GCC. More experienced GCC users can probably safely skim it, but newcomers, novices, and even GCC veterans will find lots of useful information in Chapter 3’s discussion of GCC options and arguments, controlling GCC’s output, and GCC’s usage with respect to the various dialects of C and C++. Chapter 3 also shows you how to control the C preprocessor, modify directory search paths for include files and libraries, and how to pass options to the linker and the assembler.

Chapter 4, “Advanced GCC Usage,” delves into GCC customization, such as controlling its behavior using environment variables and spec strings. You will also learn how to use GCC as a cross-compiler and how to refine the code that GCC generates. GCC has extended the C and C++ standards with some delightfully useful keywords and additional functionality, so you will spend considerable time learning what these extensions are, how to use them, and, in the interest of fairness, why you might not want to use them.

Chapter 5, “Optimizing Code with GCC,” devotes itself to using GCC’s very capable code optimizer. After giving you a jogging tour of compiler optimization theory, Chapter 5 discusses how to use GCC’s various code optimizations, which fall into two broad categories: processor-specific optimizations, and processor-independent optimizations. This material leads into Chapter 6, “Performing Code Analysis with GCC,” which introduces you to GCC’s code analysis capabilities. You will learn how to use gcov, a tool for analyzing test code coverage, and gprof, a code profiling tool. *Test code coverage* refers to determining how much of a program’s code a test suite actually tests, undeniably a valuable piece of information if you wish to make sure a program has been adequately tested before releasing it. *Code profiling* refers to developing a picture of how often a program’s code is executed and how much computing time a given section of code consumes, which allows you to focus your optimization efforts most effectively.

The next two chapters, “Using Autoconf and Automake” (Chapter 7) and “Using Libtool” (Chapter 8) endeavor to help you take better advantage of GCC’s capabilities. Autoconf is a package designed to ease the process of configuring programs to compile on a given platform. Because the compiler in question is often GCC, you can use Autoconf to customize how source code is compiled for a given CPU or hardware platform. The idea is to write portable code and to let Autoconf figure out how to take best advantage of GCC’s capabilities on a given architecture.

Automake similarly generates Makefiles more or less automatically (hence the name), which in turn instruct compilers what to build and how. Again, because the compiler in question is often GCC, it seems sensible to describe how to use Automake to exploit GCC.

Libtool is a wonderful utility that instructs the compiler (GCC, in this case) how to create libraries (either shared or static) for a given platform. Libtool handles the nitty-gritty of compiler and linker invocations necessary to create a proper library, which means that you do not have to remember the magic

incantations (for example, on some platforms, you have to use `-fpic` to create a shared library, while others use `-fPIC`, and still others use another construct altogether).

The last four chapters are devoted to GCC miscellanea and administrivia. Chapter 9, “Troubleshooting GCC,” helps you solve common problems encountered using GCC, and suggests workarounds (spelled k-l-u-d-g-e-s) for the worst of GCC’s known misfeatures and warts. For better or worse, the GNU project insists on using Texinfo for online documentation, which, in GCC’s case at least, mars an otherwise excellent documentation set. To remedy this shortcoming and enable you to take advantage of GCC’s documentation, Chapter 10, “Using GCC’s Online Help,” introduces you to the Info user interface and also suggests some alternative programs you can use if you simply cannot or will not use Info. Chapter 11, “GCC Command-Line Options,” serves as a reference section, listing and briefly describing all of GCC’s documented options. Chapter 12, “Additional GCC Resources,” lists sources of GCC information, such as mailing lists, Web pages, and other additional reading that you might find useful as you work with GCC. Appendices A, “Building and Installing Glibc,” and B, “Machine and Processor-Specific Options for GCC,” provide complete reference information for all of GCC’s command-line options and arguments and the architecture-specific options.

Ways to Read This Book

If we have done our job properly, you should be able to read *The Definitive Guide to GCC* cover to cover and follow its intended progression from introductory material on through the advanced topics. Alternatively, you can read, or ignore, lumps of chapters.

The chapters fall naturally into a few sections that, while providing useful for information for everyone, meet one group of readers’ needs quite precisely. To wit: if all you want to do is get GCC up and running on your system, read the installation instructions in Chapters 1 and 2. Chapters 3 through 5 are best suited for new to intermediate GCC users because these chapters cover the range of normal GCC usage—you will be well ahead of most GCC users in terms of making GCC sing and dance to your tune after reading these three chapters. Advanced users will get the most benefit from Chapters 6 through 8 because these chapters focus on more advanced topics that explore GCC features and capabilities that would leave uninitiated readers dazed and confused. If you are having trouble installing or using GCC, have a look at Chapters 9 and 10. Chapter 11 can be used as a quick reference to all of GCC’s known command-line options and arguments. Programmers, particularly those new to GCC and command-line compilation environments, will want to read Chapters 3 through 10. Whether you start with the appetizer and finish with dessert or simply graze, *The Definitive Guide to GCC*, and GCC, has something that you can use.

More About GCC and *The Definitive Guide to 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 we will also show you GCC's development model, which should help you understand why GCC has some features, 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 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, 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. A partial list of GCC's supported platforms, many of which you might never have heard of, much less used (we certainly have not), includes (at least) the architectures listed in Table 1.

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. Processor Architectures Supported by GCC

Architecture	Description
AMD29K	AMD Am29000 architectures
ARM	Advanced RISC Machines architectures
ARC	Argonaut ARC processors
AVR	ATMEL AVR microcontrollers
DEC Alpha	Compaq (néé Digital Equipment Corporation) Alpha processors
H8/300	Hitachi H8/300 CPUs
HPPA	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
RT	IBM RT PC architectures
SPARC	Sun Microsystems family of SPARC CPUs
Hitachi SH3/4/5	Super Hitachi 3, 4, and 5 family of processors
TMS320C3x/C4x	Texas Instruments TMS320c3x and TMS320C4x DSPs

New GCC users might be confused by mention of EGCS, the Experimental (or Enhanced) GNU Compiler Suite, on the GCC Web site (<http://gcc.gnu.org/>) and elsewhere on the Web and in printed literature. 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.

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 step he needed to take. 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 Labs. 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 Stallman used at the time 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. Development of this ur-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."

As suggested previously, GCC development forked during the 1990s 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. EGCS tracked GCC closely, but was independently and more actively maintained than GCC. Eventually, the two compilers were merged into a single compiler and the EGCS steering committee became GCC's official maintainers. Meanwhile, the Pentium Compiler Group (PCG) project created its own version of GCC, PGCC,

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/>.

Table 2 highlights GCC's release history. As you read it, keep in mind that the EGCS project maintained two version numbers. The first was in the form 2.9m.nn and indicated the relationship between the GCC and EGCS trees. The second number identified EGCS releases and took the form 1.m.n. Version numbering reverted to a single number of the form 2.9m.nn after the April 1999 merger between GCC and EGCS.

Table 2. GCC Release History

Name	Version	Release Date
GCC 3.3.2	3.3.2	October 17, 2003
GCC 3.3.1	3.3.1	August 8, 2003
GCC 3.3	3.3	May 13, 2003
GCC 3.2.3	3.2.3	April 22, 2003
GCC 3.2.2	3.2.2	February 5, 2003
GCC 3.2.1	3.2.1	November 19, 2002
GCC 3.2	3.2	August 14, 2002
GCC 3.1.1	3.1.1	July 25, 2002
GCC 3.1	3.1	May 15, 2002
GCC 3.0.4	3.0.4	February 20, 2002
GCC 3.0.3	3.0.3	December 20, 2001
GCC 3.0.2	3.0.2	October 25, 2001
GCC 3.0.1	3.0.1	August 20, 2001
GCC 3.0	3.0	June 18, 2001
GCC 2.95.3	2.95.3	March 16, 2001
GCC 2.95.2	2.95.2	October 24, 1999

Table 2. GCC Release History (continued)

Name	Version	Release Date
GCC 2.95.1	2.95.1	August 19, 1999
GCC 2.95	2.95	July 31, 1999
EGCS 1.1.2	2.91.66	March 15, 1999
EGCS 1.1.1	2.91.60	December 1, 1998
EGCS 1.1.0	2.91.57	September 3, 1998
EGCS 1.0.3	2.90.29	May 15, 1998
EGCS 1.0.2	2.90.27	March 16, 1998
gcc 2.8.1	2.8.1	March 2, 1998
gcc 2.8.0	2.8.0	January 7, 1998
EGCS 1.0.1	2.90.23	January 6, 1998
EGCS 1.0	2.90.21	December 3, 1997

GCC passed a long-awaited milestone on June 18, 2001 with the release of version 3.0. Version 3.x's key features, which Chapter 1 discusses in detail, include the following:

- *Additional targets*: New CPU targets have been added and significant enhancements have been added for several existing platforms.
- *Better documentation*: Documentation has been updated, corrected, and, in some cases, rewritten.
- *New languages*: Additional languages have been added and support for existing languages has been improved and extended.
- *Optimization enhancements*: Various features of the code optimizer, especially for ISO C99 functions, have been improved.
- *Miscellaneous changes*: Additional enhancements include internal garbage collection and an extensive test suite for verifying GCC source builds.

Who Maintains GCC?

Formally, GCC is a GNU Project, which is directed by the Free Software Foundation (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 free software licenses. 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 that 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, over 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, we 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 to the work of D. J. Delorie, GCC works on most modern DOS versions, including MS-DOS from Microsoft, PC-DOS from IBM, and DR-DOS from Lineo. Indeed, Delorie's work resulted in ports of most of the GNU tools for DOS-like environments. Cygnus Solutions, now owned by Red Hat Software, 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.

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). See <http://www.goof.com/pcg/> for more information.

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 their 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 their 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 generated over 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 the look at alternatives to GCC. This is a book about GCC, after all, so we hope you will forgive us for leaving you on your own when it comes to finding information about other compilers.

CHAPTER 1

Building GCC

BUILDING GCC FROM SCRATCH is generally perceived as a difficult, complex, and risky undertaking. We respectfully disagree. Admittedly, the process of compiling a compiler from source is complex, but the GNU development team takes care of most of the complexity for you. Building the compiler imposes no risk at all, but installing it, or, rather, installing it *incorrectly*, can destabilize your system. You can, however, install the newly built compiler as a supplemental or secondary compiler, rather than as the system compiler, and completely sidestep this risk. On the other hand, if you follow the instructions in this chapter, you can install GCC 3.3 as your primary compiler without impacting your system's overall stability. One of the primary points of this chapter is to demonstrate that compiling GCC from scratch is not difficult. It requires some care and attention to detail—and lots of time!—but, as with many tasks, good instructions make it simple to do. If you need some motivation to undertake the upgrade, the section toward the end of this chapter titled “What Is New in GCC 3?” should suffice. It covers a lot of the improvements that make it worthwhile to upgrade.

Why Build GCC from Source?

There are a number of reasons to build GCC from source. One of the most important is that GCC’s feature set (and, as a result, the pool of possible bugs) continues to grow in each new release. If you want to take advantage of new features, you have two choices: wait for a precompiled binary to become available for your CPU and operating system, or build it yourself. Another important reason to build GCC yourself is that doing so enables you to customize it for your system. For example, the default configuration of prebuilt GCC binaries usually includes all supported languages. If you do not need, for example, the Ada or Java front ends, you can prevent them from being built, tested, and installed by passing the appropriate options to the configure script. Similarly, you can modify installation paths, munge the names of the installed binaries, or create a cross-compiler by invoking GCC’s configure script using the appropriate incantations.

Naturally, if you want to participate in GCC development, you will need to build GCC from source. The flow of patches that fix bugs, add new features, and extend or enhance existing features is steady and high, too. If a particular bug bites you or if a specific enhancement appeals to you, you will only be able to take advantage of the patch if you know how to apply it and rebuild the compiler. While we are on the subject, allow us to point out that the act of building (and,

optionally, testing) GCC releases, snapshots, or the latest CVS source tree constitutes participation in GCC development. How so? Simply put, without binaries to execute, you might as well use your computer for a doorstop or a bookend. In order to create binaries, you need a compiler, preferably one that is stable and efficient. GCC needs to be tested on as many different systems as possible in order to maximize its stability and efficiency and to be as bug free as possible. If GCC builds and tests successfully on your system, that is one more data point in GCC's favor. If your particular combination of hardware and software surfaces a hitherto unknown problem, this is better still, because either you or the developers, or you and the developers together, can identify and fix the problem.

Finally, and although some might find this peculiar, you might find successfully building a very complicated piece of software on your system to be satisfying or even entertaining. We do. Go figure.

Previewing the Build Process

The process of building GCC is best approached as a series of smaller steps that organize the process into bite-sized morsels that are easy to digest. The steps we will follow in this chapter include

1. Verifying software requirements
2. Preparing the installation system
3. Downloading the source code
4. Configuring the source code
5. Building the compiler
6. Testing the compiler
7. Installing the compiler

Verifying Software Requirements

Before you can successfully build GCC, you need to make sure that you have at least the minimum required versions of certain utilities installed on your system. These programs and utilities must be installed before you begin building GCC, or it either will not work at all or will fail at some point in the process. Preparing the installation system amounts to making sure you have sufficient disk space for the build tree and deciding whether you want to install the new compiler as the primary system compiler or as a supplemental or alternate compiler.

Downloading and configuring the source code is straightforward and uncomplicated. Building the compiler is brainless, involving one command (`make bootstrap`) and lots of time. Strictly speaking, testing the compiler is an optional step but in our opinion it should be a required step. Like building GCC, testing the compiler build requires a single command and lots of time. If the build tests out okay, and release versions should, the final step is to install the compiler. You may want to do some comparisons between the old compiler and the new compiler, so we will jog through a couple of examples to show you one way to perform these comparisons.

NOTE *Although this book is about GCC, not the Linux operating system or the Intel x86 architecture, the process of building GCC described in this chapter is demonstrated using an Intel x86-based Linux system, specifically, an Intel Pentium II 350 MHz running Slackware Linux 8.0, which uses Linux kernel version 2.4.18 and GNU libc (Glibc) 2.2.5. It simply is not possible to cover all the possible combinations of hardware and operating systems on which GCC runs. If it makes you feel any better, Chapter 2 shows you how to build and install GCC for various Windows and DOS platforms.*

WARNING *At the time this chapter was written, the GCC development team warned that GNU libc 2.2.3 and earlier should not be built with GCC 3.0. Because GCC 3.0 (and, accordingly, 3.1) introduced new changes in exception handling, building Glibc 2.2.3 (or an earlier version of Glibc) with GCC 3.0 or a later version will result in a binary-incompatible Glibc. This means that programs built against a C library that was linked with an older version of GCC (2.95.3, for example) would be incompatible with a C library built with GCC 3.0 or later. The resulting incompatibility could render your system utterly unusable. Therefore, do not rebuild Glibc 2.2.3 or earlier with GCC 3.0. You can use Glibc 2.2.3 with GCC 3.0 without any problem; you simply should not try to build Glibc 2.2.3 with GCC 3.0.*

Preparing the Installation System

The GCC build process requires considerable disk space, especially in the build tree. The unpacked source tree for version 3.0 alone claims 182MB. Doing a full `make bootstrap` consumes 555MB. The `make bootstrap-lean` command described later in this chapter reduces this figure to 389MB. Performing the `make check` step takes up an additional 18MB. So, the first step to prepare your system for building the compiler is to make sure you have enough free space on the file system you will use. In easily digestible values, we recommend using a file system with at least 1GB free disk space for the build tree. All of the examples in this book use our home directory for the build tree and `/usr/local` for the installed compiler.

NOTE The disk usage figures mentioned here are guidelines because the actual usage will fluctuate from release to release and depend on the size of the binary images, which will vary based on the operating system, architecture, and file system.

Similarly, make sure you have enough space on the disk or partition on which the completed compiler suite will be installed. A complete installation of all supported languages, with national language support (NLS) and with shared libraries, requires approximately 210MB. The section titled “Installing GCC” demonstrates installing the compiler into /usr, but the installation process described in that section also removes the existing compiler, so the net change in disk space usage is relatively small. You can use various methods to reduce the footprint of the build tree and of the installed compiler. These are mentioned in the sections titled “Downloading the Source Code” and “Configuring the Source Code.”

The next step is to decide how to build and install GCC, that is, whether you will build and install it as a mortal user or as root. We recommend building it as a mortal (nonroot) user and using root privileges only to install it. GCC does not require any special access to your system during the build process, so there is no reason to build it as the superuser. If you intend to install GCC in a system directory or in a file system to which you do not have write access, then the `make install` step must be executed by the root user or a user with root equivalence. On the other hand, if you intend to install the new GCC in your home directory, you will not need root access at all, because all of the compiler’s components will be installed relative to your home directory. The examples in this chapter install GCC into a subdirectory of /usr/local, except in the section “Installing GCC,” which uses /usr. So the installation step requires root access.

To give you some concrete numbers with which to plan, the disk footprint after various steps in the configuration process appears in the following list. The first set of figures show the disk usage on our system after doing the complete build and installation process using `make bootstrap`:

- After `configure`: 3MB additional disk space used
- After `make bootstrap`: 555MB additional disk space used
- After `make check`: 571MB additional disk space used
- After `make install`: 210MB additional disk space used

The next set of figures shows the disk usage using the `make bootstrap-lean` process. As you can see, the `make bootstrap-lean` step conserves about 160MB disk space:

- After `configure`: 3MB additional disk space used
- After `make bootstrap-lean`: 389MB additional disk space used
- After `make check`: 407MB additional disk space used
- After `make install`: 208MB additional disk space used

In both sets of figures, the disk space for the first three steps is used on the build file system, whereas the disk space shown for the `make install` step is on the *installation file system*, that is, on the file system where the compiler is installed. As such, the disk space used by the installed compiler is in addition to the build file system.

Downloading the Source Code

We are going to assume that you can download the source code without our help (we are all major league gearheads here, right?). You can download the GCC source code from the GNU FTP site or, better, from a mirror that is close to you in terms of network topology. A list of mirrors can be viewed online at <http://www.gnu.org/server/list-mirrors.html>. The files you want are

- `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-3.3.tar.gz`
- `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-testsuite-3.3.tar.gz`
- `ftp://ftp.gnu.org/gnu/dejagnu/dejagnu-1.4.3.tar.gz`
- `ftp://ftp.gnu.org/gnu/binutils/binutils-2.12.1.tar.gz`

If you are short on disk space or do not want to install the complete compiler, you can download just the core compilation engine plus the files that provide support for specific compilers. In this case, then, instead of downloading `gcc-3.3.tar.gz`, download the compiler core, `gcc-core-3.3.tar.gz`, and the language-specific files for each language you want.

- *C++*: `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-g++-3.3.tar.gz`
- *FORTRAN*: `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-g77-3.3.tar.gz`
- *Java*: `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-java-3.3.tar.gz`
- *Objective C*: `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-objc-3.3.tar.gz`
- *Ada*: `ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-ada-3.3.tar.gz`

You can save a small amount of disk space by leaving out the test suite, <ftp://ftp.gnu.org/gnu/gcc/gcc-3.3/gcc-testsuite-3.3.tar.gz>, which will also save you from having to download and install the test harness, DejaGNU ([dejagnu-1.4.3.tar.gz](ftp://ftp.gnu.org/gnu/dejagnu/dejagnu-1.4.3.tar.gz)).

If you are building a CVS snapshot, you might need one or more of the following tools, also available from the GNU project:

- Autoconf
- Automake
- Bison
- Gperf
- Gettext

The versions of these programs that we used are listed in Table 1-1, which also shows you the latest version available at the time this book went to the printer, and from where you can download them.

Table 1-1. Tools for Building GCC from CVS

Tool	Version Used	Current Version	Download
Autoconf	2.50	2.53	ftp://ftp.gnu.org/gnu/autoconf/autoconf-2.53.tar.gz
Automake	1.4-p4	1.6.1	ftp://ftp.gnu.org/gnu/automake/automake-1.6.1.tar.gz
Bison	1.28	1.35	ftp://ftp.gnu.org/gnu/bison/bison-1.35.tar.gz
Gperf	2.7.2	2.7.2	ftp://ftp.gnu.org/gnu.gperf/gperf-2.7.2.tar.gz
Gettext	0.10.38	0.11.2	ftp://ftp.gnu.org/gnu/gettext/gettext-0.11.2

Installing Autoconf and Automake is described in Chapter 7, so refer to that chapter for detailed installation instructions for these packages. The other packages, Bison, Gperf, and Gettext, should all build and install with minimal fuss. Briefly, to build and install them, use the following steps (which assume you have a working compiler):

1. Uncompress and extract the archive(s).
2. Make the extracted directory the current directory.
3. Run the `configure` script.
4. Execute `make`.
5. Execute `make check` (optional).
6. Execute `make install`.

Let us emphasize that Autoconf, Automake, Bison, Gperf, and Gettext are only necessary if you are building GCC from the CVS tree! Actual releases produced by the GCC steering committee do not require these utilities.

Installing the Source Code

After you download the various source files, create a build directory. On our system, we had downloaded them into `/home/kwall/src`, so we created a build directory named `gccbuild` in this directory:

```
$ cd /home/kwall/src  
$ mkdir gccbuild
```

Naturally, replace `/home/kwall` with your own home directory.

Next, extract the tarballs:

```
$ tar -zxf gcc-3.3.tar.gz  
$ tar -zxf gcc-testsuite-3.3.tar.gz  
$ cd gcc-3.3  
$ tar -zxf ..../binutils-2.12.1.tar.gz  
$ mv binutils-2.12.1 binutils
```

Unless you rename the resulting directories, the source distribution extracts into a directory named `gcc-3.3`. We will refer to this throughout this book as the *source directory* or the *source tree* in order to distinguish the source directory from the *build* or *object directory*. This distinction is important because we adhere to the recommendation of the GCC developers and do not build the compiler in the source directory. Why? Keeping the source tree unpolluted by the

detritus of a build makes it much easier to apply patches. A separate build directory also makes it trivial to delete an old or failed build—you simply delete the entire contents of the build tree. Another reason to maintain separate source and build directories is because a configuration in which the source and build directories are the same is not as well tested. The build process creates directories on the fly, so building in a separate directory makes sure the build process does not clobber a preexisting directory. Finally, we take the developers at their word—if they keep their source and build directories separate, we reckon it is prudent to follow their example.

CAUTION *Do not use a build directory that is a subdirectory of the source tree. Doing so is entirely unsupported and will probably result in mysterious failures and problems.*

Although the build examples in this chapter presume that you opt to install the complete collection of compilers, you are not, of course, required or compelled to do so. You can cherry-pick only the languages you want. If you prefer to use this approach and be more selective, you must at least install and build `gcc-core-3.3.tar.gz`. This file contains the core back-end compilation engine, the C compiler, and other code common to all of the language-specific front ends. The language-specific tarballs in the following list include the front-end compiler for the given language and, for C++, Objective C, Java, and FORTRAN, the runtime library for that language.

If you build only select languages, uncompress and extract the core compiler archive and the specific language files in the same directory. For example, to install only the core C compiler and the C++ compiler, the sequence of commands would be

```
$ tar -zxf gcc-core-3.3.tar.gz  
$ tar -zxf gcc-g++-3.3.tar.gz
```

You can also skip the test suite, in which case you omit the step that extracts `gcc-testsuite-3.3.tar.gz` and do not follow the commands discussed in the section titled “Testing the Build” later in the chapter.

After this point, you should have an arrangement that resembles the following:

```
/home/kwall  
/home/kwall/src  
/home/kwall/src/gcc-3.3  
/home/kwall/src/gccbuild
```

/home/kwall/src/gcc-3.3 is the source directory and /home/kwall/src/gccbuild is the build directory.

Of course, you are not obliged to work in your home directory. We do so because it avoids a lot of potential problems related to file system permissions.

Configuring the Source Code

Configuring the source code is quite simple. Make your build directory your current directory.

```
$ cd /home/kwall/src/gccbuild
```

Invoke the `configure` script in the source directory.

```
$ ../gcc-3.3/configure \
--prefix=/usr/local/gcc33 \
--srcdir=../gcc-3.3
```

Note that we use the \ character to indicate code continuation to the shell. You can, if you wish, type the command on a single line. After a few minutes, the configuration script completes and the previously empty build directory will be configured for a GCC build. The invocation shown in the example uses the `--prefix` option to specify /usr/local/gcc33 (a new directory) into which the completed compiler will be installed. `--srcdir` is a hint to the `configure` script telling it where to find the source code. This option is not strictly required because the `configure` script contains logic to figure out where the source directory is relative to the build directory, but it does not hurt to use `--srcdir`, either.

Depending on the type of compiler you intend to build, you may need to identify more precisely the system on which the compiler will run or the system for which the compiler must generate code. To do so, use the `--build[=system]`, `--host[=system]`, and `--target=[system]` arguments to `configure`, specifying the proper system name in `system`. The GCC build process (more generally, any modern Autoconf-based `configure` script) understands three different system names: the build system, the host system, and the target system. The *build* system is the machine on which you build GCC, the *host* system is the machine on which the completed compiler will run, and the *target* system is the machine for which the compiler will generate code. Perhaps surprisingly, the build, host, and target systems (or system names) do not have to be the same, and often are not.

What Is in a (System) Name?

To understand why the build, host, and target system need not always be the same system, consider how you would build GCC on a system that lacks a functioning compiler. For example, if you want to build GCC for a Microsoft Windows system, you have to have GCC running on it, right? Well, actually, you do not, because you can use a *cross-compiler*, a compiler running on one CPU architecture that generates code for another CPU architecture, neatly sidestepping a potential catch-22 situation. So, to build GCC for a Microsoft Windows-based Intel x86 system (colloquially known as a *Wintel* system), you could use a cross-compiler executing on a SPARC system running Solaris that knows how to create a binary that runs on a Wintel system.

Various permutations of build, host, and target names are possible. If the build, host, and target systems are all the same, you are building a *native compiler*, one that executes on and generates executables for the same type of system as the system on which it is built. Obviously, this is the simplest case. If the build and host systems are the same, but the target system is different, you are building a cross-compiler. If the build and target system are the same, but the host system is different, you are building what is referred to as a *crossback compiler*, that is, you are using a cross-compiler to build a cross-compiler that will run on the system on which the compiler is being built. If the target and host system are the same, but the build system is different, you are building a *crossed native* (or a cross-built native or a host-x-host) compiler, which means you are using a cross-compiler to build a native compiler a third type of system. If the build, host, and target systems are all different, you are building what is called a *canadian*, a compiler that builds on one architecture, runs on another architecture, and creates code for a third architecture.

Confused? Table 1-2 shows some examples that illustrate possible combinations of build, host, and target systems.

Table 1-2. Compiler Types

Build	Host	Target	Compiler Type	Result
x86	x86	x86	Native	Built on an x86 to run on an x86 to generate binaries for an x86
SH	SH	ARM	Cross	Built on a SuperH to run on a SuperH to generate binaries for an ARM
x86	MIPS	x86	Crossback	Built on an x86 to run on a MIPS to generate binaries for an x86
PPC	SPARC	SPARC	Crossed native	Built on a PPC to run on a SPARC to generate code for a SPARC
ARM	SH	MIPS	Canadian	Built on an ARM to run on a SuperH to generate code for a MIPS

Why go to all this trouble? The most common reason is to bootstrap a compiler for a system that lacks one. If you do not have a running compiler for architecture A but you do have one for architecture B, you can use B's compiler to generate binaries for A or to generate a compiler for A. See the section titled “Using GCC As a Cross-Compiler” in Chapter 4 for more gory details about cross-compilation.

Strictly speaking, you do not have to feed `configure` any options at all. Like most GNU software, the default configuration is to install into `/usr/local` (which is equivalent to specifying `--prefix=/usr/local`). In addition, the `configure` script uses sane, well-tested, and generally accepted defaults for other configuration details, so you can usually rely on the compiler being compiled properly and, in general, to Do The Right Thing. Nevertheless, we prefer to specify a directory beneath `/usr/local` so that we can have multiple installations of GCC and access them by changing the path to the `gcc` binary. The next section describes the options that the `configure` script accepts that enable you to customize GCC's compile-time defaults.

Additional Configuration Options

Even though the `configure` script creates a reasonable default configuration, you can exercise greater control over the configuration by calling with various options. Table 1-3 lists the options that `configure` accepts and their default values, and briefly describes each option. Readers accustomed to using `configure` scripts are cautioned that the output from `./configure --help` might list options that are not shown in Table 1-3. Perversely, the `--help` option for GCC's `configure` script might list options that do not work, so we recommend that you disregard the output from `--help`. Indeed, the list the `--help` displays is very short and lacks many of the options shown in Table 1-3.

Table 1-3. Options for GCC's configure Script

Option	Default Value	Description
--prefix= <i>dir</i>	/usr/local	Sets <i>dir</i> as the top-level directory into which GCC will be installed
--exec-prefix= <i>dir</i>	/usr/local	Sets <i>dir</i> as the directory into which architecture-dependent files will be installed
--bindir= <i>dir</i>	/usr/local/bin	Sets <i>dir</i> as the directory into which binaries invoked by users will be installed
--libdir= <i>dir</i>	/usr/local/lib	Sets <i>dir</i> as the directory into which object code and libraries used by GCC itself will be installed
--with-slibdir= <i>dir</i>	/usr/local/lib	Sets <i>dir</i> as the directory into which the shared library, libgcc, will be installed
--infodir= <i>dir</i>	/usr/local/info	Sets <i>dir</i> as the directory into which GCC's Texinfo (info) files will be installed
--mandir= <i>dir</i>	/usr/local/man	Sets <i>dir</i> as the directory into which the generated GCC manual pages will be installed
--with-gxx-include-dir= <i>dir</i>	/usr/local/include/g++-v3	Sets <i>dir</i> as the default into which the g++ header files will be installed
--program-prefix= <i>str</i>	None	Adds <i>str</i> as a prefix to the names of programs installed in <i>prefix/bin</i>
--program-suffix= <i>str</i>	None	Adds <i>str</i> as a suffix to the names of programs installed in <i>prefix/bin</i>
--program-transform-name= <i>pattern</i>	None	Uses the sed script <i>pattern</i> to modify the names of programs installed in <i>prefix/bin</i>
--with-local-prefix= <i>dir</i>	/usr/local	Sets <i>dir</i> as the top-level directory into which local include files are installed
--enable-shared [=pkg[,...]]	Host	Specifies the names of libraries that should be built as shared libraries (recognized names for <i>pkg</i> are libgcc, libstdc++, libffi, zlib, Boehm-gc, libjava)

Table 1-3. Options for GCC's configure Script (continued)

Option	Default Value	Description
--with-gnu-as	N/A	Tells the compiler to assume that the assembler it finds is the GNU assembler
--with-as= <i>path</i>	<i>exec-prefix/lib/gcc-lib/target/version</i>	Specifies the path to the assembler to use
--with-gnu-ld	N/A	Tells the compiler to assume that the linker it finds is the GNU linker
--with-ld= <i>path</i>	<i>exec-prefix/lib/gcc-lib/target/version</i>	Specifies the path to the linker to use
--with-stabs	Host	Instructs the compiler to use BSD STABS debugging information instead of the standard format for the host system (usually DWARF2)
--disable-multilib	Varies	Prevents building multiple targets libraries for supporting different target CPU variants, calling conventions, and other CPU-specific features (normally, GCC builds a default set when appropriate for the CPU)
--enable-threads[= <i>lib</i>]	Target	Specifies that the target supports threading and or the threading model supported by the target (recognized names for <i>lib</i> include aix, dce, mach, no, posix, pthreads, rtems, single, solaris, vxworks, and win32)
--with-cpu= <i>cpu</i>	None	Instructs the compiler to generate code for the CPU variant specified by <i>cpu</i>
--enable-alitvec	None	Tells the compiler that the target supports AltiVec enhancements (only applies to PowerPC systems)
--enable-target-optspace	Disabled	Optimizes target libraries for code space instead of code speed
--disable-cpp	Disabled	Prevents installation of a C preprocessor (cpp) visible to user space
--with-cpp-install-dir= <i>dir</i>	None	Copies the C preprocessor (cpp) to <i>prefix/dir/cpp</i> and to <i>prefix/bin</i>

Table 1-3. Options for GCC's configure Script (continued)

Option	Default Value	Description
--enable-maintainer-mode	Disabled	Regenerates GCC's master message catalog, which requires the complete source tree
--enable-version-specific-runtime-libs	Disabled	Causes the compiler's runtime libraries to be installed in the compiler-specific subdirectory (<i>libsubdir</i>), which is useful if you want to use multiple versions of GCC simultaneously
--enable-languages= <i>lang</i> [,...]	All	Specifies the language front ends to build (recognized names for <i>lang</i> are ada, c, c++, f77, java, and objc)
--disable-libgcj	Disabled	Prevents building GCJ's runtime libraries, even if building the Java front end
--with-dwarf2	Disabled	Instructs the compiler to use DWARF2 debugging information instead of the host default value on systems for which DWARF2 is not the default
--enable-win32-registry	None	Tells GCC hosted on a Microsoft Windows system to look up installation paths in the system registry, using the default key, HKEY_LOCAL_MACHINE\Software\Free Software Foundation\key (key defaults to the GCC version number)
--enable-win32-registry= <i>key</i>	HKEY_LOCAL_MACHINE\Software\Free Software Foundation\vernum	Tells GCC hosted on a Microsoft Windows system to look up the installation path in the system registry, using the key specified in <i>key</i> , where <i>vernum</i> is the GCC version number
--disable-win32-registry	Enabled only for Windows	Disables use of the Win32 registry
--nfp	m68k-sun-sunosN and m68k-isi-bsd only	Identifies processors that lack hardware floating-point capability
--enable-checking	Disabled for releases, enabled in snapshots and CVS versions	Enables extra error checking during compilation but does not change generated code

Table 1-3. Options for GCC's configure Script (continued)

Option	Default Value	Description
--enable-checking= <i>list</i>	misc, tree, gc	Defines the types of extra error checking to perform during compilation (possible values are misc, tree, gc, rtl, and gcac)
--enable-nls	Enabled by default	Enables national language support, which GCC uses to emit error messages and other diagnostics in languages other than American English
--disable-nls	Enabled by default	Disables NLS
--with-included- gettext	Disabled	Configures GCC to use the copy of GNU Gettext if --enable-nls is specified and the build system has its own Gettext library
--with-catgets	Disabled	Configures GCC to use the older catgets interface for message translation if --enable-nls is specified and the system lacks GNU Gettext (ordinarily, in such a case, the build would use the copy of Gettext included in the GCC distribution)
--with-system-zlib	Disabled	Configures the build process to use the zlib compression library installed on the build system rather than the version provided in the GCC sources

Many of the configuration options listed in Table 1-3, or their impact on the resulting compiler, require additional explanation. As a rule, the only change we make in installation directories is to use --prefix to customize the installation directory. This enables us to install the compiler and all of its related files below the single top-level directory specified in *dir*, making it simple to manage multiple versions of GCC installed on a single system. The balance of the directory-manipulation options listed in Table 1-3 (from --exec-prefix=*dir* to --with-gxx-include-dir=*dir*) provide finer control over where the installation process drops the various compiler components during installation. A related configuration option, --enable-version-specific-runtime-libs, disabled by default, installs the compiler's runtime libraries into a compiler-specific subdirectory, which is again useful if you intend to use multiple versions of GCC simultaneously.

--program-prefix and --program-suffix take directory-name mangling down to the filename level. The values of *str* passed to these arguments are attached to the beginning or end, respectively, of the program names when the programs are installed in *prefix/bin*. For example, --program-prefix=v3 turns gcc into v3gcc, g++ into v3g++, cpp into v3cpp, and so on. These two options make it possible to assign unique names to the user-executed binaries created during the build process. Similarly, --program-transform-name enables you utterly to mangle the names of files installed in *prefix/bin* using the sed script specified in *pattern*. Thus, specifying --program-transform-name="s/[[:lower:]]/[[:upper:]]/g" will translate all lowercase letters into their corresponding uppercase forms. As a result, gcc would become GCC, g++ would become G++, and so forth. You can use all three of the options discussed in this paragraph together to create a highly customized (or totally confusing) GCC installation.

The --with-local-prefix option does not do what you think it does. As stated in Table 1-2, it defines the top-level directory into which local include (header) files are installed. That is, it tells GCC where to find local header files, *not* where to install the compiler's files (use --prefix and *kin* for this purpose). So, if your system keeps locally installed headers in /opt/local/include, use --with-local-prefix=/opt/local. The default value, /usr/local, should suffice for all Linux and most Unix and Unix-like systems.

The option --enable-maintainer-mode does not enable any sort of compiler voodoo or black magic known only to GCC's developers. Rather, it causes the build process to regenerate the master message catalog used by the NLS subsystem to display messages and diagnostics in non-English (well, non-American English) languages.

You can use --disable-libgcj if, for some reason, you do not want to build the Java compiler's (gcj) runtime libraries when building the Java compiler. This option can be useful if you change gcj but do not need to change the underlying runtime library because it eliminates the potentially time-consuming process of building or rebuilding libgcj. One common use of --disable-libgcj is if you want to use the gcj compiler with another Java compiler's runtime libraries, if you know how to do this and make it work.

Finally, here is how to make sense of the NLS-related configuration options. If you do not need non-English messages and compiler diagnostics, specify --disable-nls. If you need NLS and have the GNU Gettext library installed, specify --enable-nls. If you need NLS and *do not* have GNU Gettext installed, specify --enable-nls and --with-included-gettext. If you need NLS, do not have Gettext installed, and do not want to use the copy of Gettext shipped with the GCC sources, specify --enable-nls and --with-catgets.

Compiling GCC

Now that you have all of the preliminaries out of the way, you are *finally* ready to build the compiler. In most situations, you only need to use one of two commands,

`make bootstrap` or `make bootstrap-lean`. Briefly, these two `make` commands build the new compiler from scratch according to the configuration you created with the `configure` script. Our suggestion is to put this book aside for a moment, type `make bootstrap` to start the build process, and then finish reading this chapter while the build takes place.

Be clear, of course, that a full bootstrap is time consuming, even on fast machines. On a Pentium II 266 with 256MB RAM, the complete bootstrap took over 3 hours. On Pentium II 350 with 256MB RAM, it took about 90 minutes. On an AMD 1.2 GHz system with 256MB RAM, it took just under 47 minutes, and we had similar results on a Pentium III 800 with 256MB RAM. As you can see, the build process definitely benefits from a faster CPU and more RAM! This section of the chapter explains the build process in detail, or at least in enough detail to understand why it takes so long.

TIP *To maximize the number of CPU cycles and the amount of RAM available to the build process, disable as many nonessential processes running on the build system as possible. Likewise, if you have sufficient administrative privileges on the build system and if the operating system supports it, you can experiment with increasing the priority of the build process. On Linux and Unix systems, you would do this using the `renice` command (as the root user) to modify the priority of the top-level shell running the build (the one executing `make bootstrap` or `make bootstrap-lean`) or by starting the build as the root user (not recommended, by the way) and using the `nice` command when you start the build.*

Compilation Phases

The recommended and preferred method for building the compiler is using the `make bootstrap` command. The discussion that follows applies to a full bootstrap (executed by `make bootstrap`) but can also be applied to the `make bootstrap-lean` target unless noted otherwise. A full bootstrap build executes the following steps:

1. Build all host tools required to build the compiler, such as `texinfo`, `bison`, `gcov`, and `gperf`.
2. If the binary utilities for the compiler target (`bfd`, `binutils`, `gas`, `gprof`, `ld`, and `opcodes`) were moved or linked to the top-level GCC build directory prior to executing `configure`, build them.
3. Perform a three-stage build of the compiler itself. First, build the new compiler with the native compiler, which results in the *stage 1 compiler*. Next, build the new compiler, known as the *stage 2 compiler*, with the stage 1 compiler. Lastly, build the new compiler a third time (the *stage 3 compiler*) with the stage 2 compiler.

4. Compare the stage 2 and stage 3 compilers to each other. They should be identical; otherwise the stage 2 compiler is probably flawed and, as a result, you should not install the stage 3 compiler.
5. Using the stage 3 compiler, compile the needed runtime libraries.

If you use `make bootstrap-lean` instead of `make bootstrap`, the object files from stages 1 and 2 will be deleted after they are no longer needed, minimizing disk space consumption.

During the build process, you will probably see a number of warning messages scroll by. Although disconcerting, they are *warnings*, not *errors*, and so you can safely disregard them. As a rule, these warnings are limited to certain files. You only need to pay attention to actual errors that cause the build process to fail. In other cases, certain commands executed during the build process will return a nonzero error code, which the `make` program recognizes as a failure but nevertheless ignores. Often as not, the “failure” is simply a missing file that is not relevant and, like the warning messages just mentioned, it is safe to ignore such failures unless they cause the build process to terminate abnormally.

If you have an SMP system (lucky you!), you can start a *parallel build*, a build that takes advantage of all the CPUs in your system, by executing the following command if your version of GNU `make` is older than 3.79:

```
$ make bootstrap MAKE="make -j 2" -j 2
```

The reason you have to use the `MAKE` environment variable is because versions of `make` older than 3.79 do not pass the `-j 2` option down to child `make` processes (so-called *submakes*). By setting the `MAKE` environment variable, which is inherited by submakes, you get the parallelism you want. If you have version 3.79 or newer of GNU `make`, a simple

```
$ make -j 2 bootstrap
```

will be sufficient. The numeric argument to `-j` should be no greater than the number of CPUs you have in your system—although making it greater will work, it will not result in any better performance.

Other Makeables

As you might surmise, the GCC build process supports many more targets than `bootstrap` and `bootstrap-lean`. A list of the most useful `makeables` other than `bootstrap` and `bootstrap-lean` includes the targets shown in Table 1-4.

Table 1-4. GCC Makefile Targets

Target	Description
all	Coordinates building all of the other targets, according to the values of <i>host</i> , <i>build</i> , and <i>target</i> you specified (if any) when you executed the <code>configure</code> script.
bubblestrap	Incrementally rebuilds each stage.
check	Executes the test suite.
clean	Performs the <code>mostlyclean</code> step and also deletes all other files created by <code>make all</code> .
cleanstrap	Executes <code>make clean</code> followed by <code>make bootstrap</code> .
compare	Compares the results of stages 2 and 3, which should be the same.
distclean	Deletes the same files as <code>clean</code> , plus files generated by the <code>configure</code> script.
dvi	Generates the DVI-formatted documentation, which is suitable for printing.
extraclean	Executes <code>make clean</code> and also deletes temporary and backup files created during the build.
generated-manpages	Creates the generated manual pages, which are a subset of the complete Texinfo documentation.
install	Installs GCC into the directory or directories specified when you executed the <code>configure</code> script.
maintainer-clean	Deletes the same files as <code>make distclean</code> and also deletes all files generated from other files.
mostlyclean	Deletes the files created by the build process.
quickstrap	Rebuilds the most recently completed stage, meaning you do not have to keep track of the stage you are on.
restage N	Executes <code>make unstageN</code> and the rebuilds stage N with the appropriate compiler flags, where N is 1, 2, 3, or 4.
stage N	Moves the files for each stage N , where N is 1, 2, 3, or 4 to the appropriate subdirectory in the build tree.
uninstall	Deletes the installed files (except for documentation). Note that <code>make uninstall</code> may not work properly.
unstage N	Reverses the actions of stage N , where N is 1, 2, 3, or 4.

Testing the Build

We strongly recommend you test your shiny new compiler before you install it, even though the GCC documentation describes running the test suite as an optional step. The primary reason we recommend testing the compiler is that it will be used to create the software you use every day. The time required to run the test suite seems like a small price to pay for making sure you do not have a flawed or broken compiler. In most cases, you will not encounter any show-stopping errors, but, in our opinion, it does not hurt to make sure. This section explains how to install and run the test suite and briefly describes how to interpret the results.

In order to run the GCC test suite, you need Tcl, Expect, and DejaGNU installed. Tcl is the Tool Command Language, and it is usually part of standard Linux or Unix systems that have a working development environment. On Windows-based systems, you probably *do not* have it installed unless you specifically did so. Expect is a Tcl-based tool that automates interactive programs. It is rather less commonly installed than Tcl. You can download Tcl from the Tcl Developer's Xchange Web site at <http://www.tcl.tk/>. Expect is available from the Expect home page, <http://expect.nist.gov/>.

NOTE *To see if you have Tcl and Expect installed, execute the commands which tcl and which expect. The output should resemble the following:*

```
$ which tcl  
/usr/bin/tcl  
$ which expect  
/usr/bin/expect
```

If the output resembles the following lines, the package is not installed or, at least, is not in your PATH:

```
$ which tcl  
which: no tcl in (/bin:/usr/bin:/usr/local/bin)
```

If Tcl or Expect are not installed, you will need to install them before running the test suite.

Assuming you have Tcl and Expect installed, the next step is to unpack, build, and install the test harness, which automates running the tests. A *test harness* is the term used to describe the framework in which tests are executed. For GCC, the test harness is DejaGNU, which can be downloaded from the GNU FTP

site or one of its mirrors. The latest version available, at the time this was written, was 1.4.3. If you download from the primary GNU FTP site, the URL is <ftp://ftp.gnu.org/gnu/dejagnu/dejagnu-1.4.3.tar.gz>.

After you have downloaded the file, unpack and install DejaGNU. The sequence of commands for *our* crash test dummy system is as follows (we used the same /home/kwall/src directory we used for unpacking the GCC distribution):

```
$ tar -zxf dejagnu-1.4.3.tar.gz
$ cd dejagnu-1.4.3
$ ./configure
$ make
$ su
Password:
# make install
```

We recommend using the default installation directory, /usr/local, to limit the chances of something going wrong with the test process. The `make install` step requires root privileges, so use the `su` command to become the superuser. These commands unpack the source tarball, configure it for our combination of CPU and Linux version, build the test suite, and install it into its default location. “Building” the test harness is somewhat misleading, however. DejaGNU is based on Expect, which in turn is based on Tcl. Tcl is an interpreted programming language that does not require compilation. The “build process” sets a number of variables that DejaGNU uses, which takes easily less than a minute. In fact, we ordinarily build and install DejaGNU when we unpack the GCC sources just to save a little bit of time.

CAUTION *Make sure you installed the test suite itself, as explained in the section titled “Installing the Source Code” earlier in this chapter. The test process will fail if the test suite has not been installed.*

After installing DejaGNU, you might need to set the environment variables `DEJAGNULIBS` and `TCL_LIBRARY` as shown in the following code lines, modifying the pathnames to reflect your particular situation. If Tcl is in your `PATH` and if you use the default installation directory for DejaGNU, however, you will not need to set `TCL_LIBRARY` or `DEJAGNULIBS`.

```
$ export DEJAGNULIBS=/usr/local/share/dejagnu
$ export TCL_LIBRARY=/usr/lib/tcl8.3
```

Finally, make the top-level GCC build directory your current working directory and invoke the test suite using the `make` command shown in the following:

```
$ cd /home/kwall/src/gccbuild
$ make -k check
```

If invoked as shown in the previous example, the testing process will validate as many of the compiler components as possible, thus exercising the C, C++, Objective C, and FORTRAN compilers and validating the C++ and Java runtime libraries. If you wish, you can run subsets of the test suite. For example, to test only the C compiler, execute the following commands:

```
$ cd /home/kwall/src/gccbuild
$ make check-gcc
```

Similarly, to test only the C++ compiler, use the following two commands:

```
$ cd /home/kwall/src/gccbuild/gcc
$ make check-g++
```

In both of these examples, the directory `gcc` is the `gcc` subdirectory of the top-level GCC build directory. Like the build process, running the test suite can be quite time consuming and, like the build process, the faster the host system's CPU, the faster the test suite will complete.

Interpreting the test results from a high level, that is, without getting into the details of the test results, is relatively simple. In each of the test suite subdirectories, the test process creates a number of files ending with `.log`, which contain detailed test results, and `.sum`, which summarize the test results. The summary files list every test that was run and a one-word code indicating the test result. Table 1-5 lists the possible codes and their meaning.

Table 1-5. GCC Test Suite Result Codes

Code	Description
ERROR	The test suite encountered an unexpected error.
FAIL	The test failed but was not expected to do so.
PASS	The test completed successfully and was expected to do so.
UNSUPPORTED	The corresponding test is not supported for the platform in question.
WARNING	The test suite encountered a potential problem.
XFAIL	The test failed and was expected to do so.
XPASS	The test completed successfully but was not expected to do so.

Some tests will result in unexpected failures, such as tests that deliberately torture the compiler's ability to handle extreme situations. Unfortunately, the test suite does not support a greater degree of control over this situation.

NOTE *The test suite is currently in a state of flux, and the GCC steering committee, with input from developers, is considering moving to a more robust and flexible test harness, so the situation may have changed by the time you read this book.*

You can also submit your test results to the GCC project by using the `test_summary` script, which bundles up the summary files and uses the `mail` program to send the test results to a server where additional processing takes place. To use the script, execute the following commands:

```
$ cd /home/kwall/src/gcc-3.3
$ contrib/test_summary [-p extra_info.txt] -m gcc-testresults@gcc.gnu.org | /bin/sh
```

The file `extra_info.txt` can contain any information you think pertinent to the test results. It is not required, however, so you can omit this element of the command if you wish.

Installing GCC

At last, you are ready to install GCC. Your options are limited here, because the installation paths and other installation-related details were largely determined when you ran the `configure` script. The most commonly used command to install your shiny new compiler is `make install` and it *must* be executed by the root user unless you are installing it into nonsystem directories (that is, directories to which you have write access):

```
$ su -
Password:
# cd /home/kwall/src/gccbuild
# make install
# ldconfig -v
```

The `ldconfig` command updates the dynamic linker's cache file, allowing it to pick up the new shared libraries (`libgcc`, and possibly the runtime libraries for C++ and Java). It is not strictly necessary, but it does not hurt, either.

That is it. After considerable time, minimal effort on your part, and a Herculean workout of your CPU, you have a brand new compiler installed. If

you are curious about why you have gone to all the trouble of installing a new compiler, the next section, which *should* be titled “What Is So Great about GCC 3.0?,” might provide some motivation. To start using your newly installed compiler, skip ahead to Chapter 3. For information specific to installing GCC on a DOS or Windows system, read the next chapter.

What Is New in GCC 3?

Now that you have gone to the trouble of downloading all of the source code and, if necessary, upgrading some key utilities in order to compile GCC 3.3, you might want to know why you should even bother. We are firm believers in the old adage (okay, the old, *clichéd, hackneyed* adage) “If it ain’t broke, don’t fix it.” Although the version of GCC most widely used, GCC 2.95.3, is not broken, its newer sibling offers many significant advantages that make the effort to upgrade worthwhile. This section explores the improvements and new features in GCC 3.0 and the incremental improvements in releases 3.1, 3.2, and 3.3.

General Optimizer Improvements

The overall behavior of the optimization passes has been improved considerably. This section describes enhancements applicable regardless of the underlying CPU or operating system.

- *Basic block reordering pass:* Block reordering attempts to improve code performance by reducing bad or missed code branch predictions in the generated assembly code.
- *Tail call elimination:* Tail call elimination is an optimization that reduces the number of redundant function calls in generated code, reducing code size. Tail call elimination is sometimes referred to as *sibling call elimination*.

Tail call elimination affects code that looks like the following, whether it was written this way or wound up looking like this as a result of compiler optimization:

```
int a ()  
{  
    return 1;  
}  
int b ()  
{  
    return a ();  
}
```

As you can see, `b()` can be optimized to avoid a function call by simply returning 1.

- *Static single assignment (SSA) representation support:* SSA representation is an interim representation of function code that assigns each register a value only once. SSA representation makes it easier to determine the execution path from the assignment to the use of a register, which in turn makes possible optimizations that used to be difficult or impossible to perform.
- *Dead code elimination pass:* Dead code, as the name suggests, is code present in a program that never executes. Eliminating dead code reduces code size and enhances the effect of other optimizations. The following code snippet is an example of dead code:

```
if (0) {
/* dead code goes here */
}
```

Code inside the `if` block is never reached because the condition (0) always tests false.

- *Global null pointer test elimination:* Null pointer test elimination attempts to improve code performance by eliminating pointer tests that can be determined to be null either at compile time or before entering a code block.
- *Global code hoisting/unification:* Code hoisting, also known as *code unification*, is a code size optimization. If an expression's value is computed multiple times or in multiple code blocks, code hoisting evaluates whether the expression can be computed once and, if so, moves this computation above the location where it is first used and replaces the other occurrences with the expression's result.

For example, consider the following snippet:

```
int x = 1;
int y = 2;
int z = 0;
for (z = 0; z < 10; z++) {
    printf("%d\n", z*(x+y));
}
```

The expression $(x+y)$ is recomputed in each iteration of the for loop. However, it can be replaced with a constant and the computation moved above the for loop, as shown here:

```
int x = 1;
int y = 2;
int z = 0;
int hoisted= (x + y);
for (z = 0; z < 10; z++) {
    Printf("%d\n", z*hoisted;
}
```

- *More built-in features and optimizations:* These apply to the standard I/O and string libraries (as declared in the headers `<stdio.h>` and `<string.h>`), old BSD Unix functions, and ISO C99 functions. Optimizing these heavily used functions improves performance.
- *If-conversion pass:* GCC's new if-conversion pass scans source code looking for opportunities to convert various forms of if-then statements into simple assignments, which gives the compiler additional opportunities to perform subexpression elimination, code hoisting, and other code optimizations.
- *New built-in:* A new keyword, `_builtin_expect`, enables developers to give hints to the branch predictor units present in many modern CPUs, allowing each CPU to perform its own optimizations, when applicable, to speed up code performance.

New Languages and Language-Specific Improvements

GCC 3 includes support for a new language, Java. This new support is a major change. In addition to the new Java compiler, a number of enhancements have been made to languages GCC has long supported. This section discusses these changes.

- The GNU Compiler for Java (`gcj`) has been integrated into GCC. Support for Java includes the runtime library, a bytecode interpreter, and the Boehm conservative garbage collector. The runtime library contains most of the common nongraphical Java classes. GCJ can compile Java source code and bytecode to native code or to Java class files. GCJ also supports native methods written in either the standard JNI or the more efficient and convenient CNI.

- GCC includes a new C++ application binary interface (ABI). The new ABI reduces the size of binaries by decreasing the size of the symbols used in compiled code and by reducing the size and amount of debugging information embedded in the target binaries. The new ABI also improves GCC's ability to interoperate with other compilers on Intel's IA64 architecture. In addition to the new ABI, GCC's C++ improvements include a new code inliner, a new C++ support library, and dramatically enhanced conformance to the ISO C++ standard.
- `g++`, the C++ compiler, has a new code inliner that is invoked earlier in the compile process, causing it to use less time and memory to compile programs that make heavy use of templates. Plans are in place to use the inliner for C and Java programs in addition to C++.
- Other C++ improvements include the following:
 - `g++` now supports importing member functions from base classes with a `using` declaration.
 - `g++` now enforces access control for nested types.
 - A bug that, in obscure cases, resulted in functions with the same type having the same mangled name has been fixed.
 - Certain invalid conversions that were previously accepted will now be rejected. For example, assigning function pointers of one type to function pointers of another type now requires a cast, whereas previously `g++` would sometimes accept the code even without the cast.
- The following features have been removed from `g++`:
 - Guiding declarations
 - Assignment to `this` (no longer allowed by the C++ standard)
 - Expressions of the form `sizeof (X::Y)` where `Y` is a nonstatic member of `X`, even if the `sizeof` expression occurred outside of a nonstatic member function of `X`
 - Overloading the conditional operator, `?:`

- The C preprocessor, `cpp`, has been rewritten and integrated into the C, C++, and Objective C compilers. The new preprocessor includes many improvements for ISO C99 support and improvements to dependency generation. The dependency generator now does the following:
 - Creates subdirectories for dependencies generated with the `-MD` option.
 - Allows you to specify targets using the `-MT` and `-MQ` options.
 - Writes dependencies to a file using `-MF` if either the `-M` or the `-MM` option is specified.
 - The `-MP` option tells the preprocessor to create a phony target for each dependency other than the main file, causing each to depend on nothing, allowing `make` to sidestep errors that occur when you delete or rename header files without updating the Makefile.
- Support for more ISO C99 features. A partial list of these features includes
 - Restricted character set support via digraphs and `iso646.h`
 - Restricted pointers
 - Flexible array members
 - Type-generic math macros in `tgmath.h`
 - The `long long int` type and library functions
 - Additional floating-point characteristics in `float.h`
 - Implicit `int` removed
 - Reliable integer division
 - Hexadecimal floating-point constants and `%a` and `%A` conversion specifiers for `printf` and `scanf`
 - Designated initializers
 - `//` comments
 - Implicit function declarations removed

- Support mixed declarations and code
- Allow block scopes for selection and iteration statements
- Macros with a variable number of arguments
- The `vscanf` family of functions in `stdio.h` and `wchar.h`
- Trailing commas allowed in `enum` declaration
- `%lf` conversion specifier allowed in `printf`
- The `snprintf` family of functions in `stdio.h`
- Boolean type in `stdbool.h`
- Empty macro arguments
- New struct type tag compatibility rules implemented
- `_Pragma` preprocessing operator
- `__func__` predefined identifier
- `va_copy` macro
- Additional `strftime` conversion specifiers
- Deprecated aliased array parameters removed
- Relaxed constraints on aggregate and union initialization and on portable header names
- `return` without an expression no longer permitted in a function that returns a value
- The heavily used format functions, such as `printf` and `scanf`, have been enhanced with bounds-checking code, include support for ISO C99 format features, have new extensions from the Single Unix Specification, and have additional extensions from GNU's glibc-2.2. The format string checking enhances GCC's ability to generate code less susceptible to bugs caused by format string security exploits. Similar enhancements have been added to `strfmon` formats and features, again to facilitate code audits for format string security bugs.

- GCC's `-Wall` warning option now emits warnings for C code that uses undefined semantics resulting from violations of sequence point rules in the C standard. For example, the following C statements invoke undefined compiler behavior because they depend on expression evaluations that can occur in any order:

```
a = a++;
a[n] = b[n++];
a[i++] = i;
```

- Additional warning option, `-Wfloat-equal`.
- Improvements to `-Wtraditional`.

New Targets and Target-Specific Improvements

- New Intel x86 back end, generating much improved code
- Support for a generic Intel i386-elf target
- New option to emit Intel x86 assembly code using Intel style syntax (`-mintel-syntax`)
- HPUX 11 support contributed
- Improved PowerPC code generation, including scheduled prologue and epilogue
- Ports of GCC to the following platforms:
 - Atmel's AVR microcontrollers
 - Fujitsu's FR30 processor
 - Intel's IA-64 processor
 - Intel's XScale processor
 - Matsushita's AM33 processor (a member of the MN10300 processor family)
 - Mitsubishi's D30V processor

- Motorola's 68HC11 and 68HC12 processors
- Motorola's MCORE 210 and 340
- New unified back end for ARM, Thumb, and StrongARM
- Sun's picoJava processor core

Other Significant Improvements

- The compiler uses garbage collection instead of obstacks for most memory allocation.
- GCC now uses the Lengauer and Tarjan algorithm to compute dominators. The Lengauer and Tarjan algorithm is significantly faster and more space efficient than the previous algorithm used to identify dominators. It also helps the optimizer find the most efficient path through the flow graph that represents the program's control dependencies.
- To facilitate submitting bug reports to the GCC GNATS bug database, you may use the script `gccbug`. `gccbug` ensures greater consistency across bug reports and allows for more streamlined, automated bug reporting than the older HTML front end to GNATS, which continues to be available.
- The internal `libgcc` library can now be built as a shared library on systems that support shared libraries.
- The GCC test suite is now part of the distribution and includes many new tests, broadening GCC's test case coverage. In addition to tests for GCC bugs that have been fixed, many tests have been added for language features, compiler warnings, and built-in functions.
- GCC includes four new language-independent warning options: `-Wpacked`, `-Wpadded`, `-Wunreachable-code`, and `-Wdisabled-optimization`.
- GCC includes three new target-independent code generation options: `-falign-functions`, `-falign-loops`, and `-falign-jumps`.

Documentation Improvements

- Substantially rewritten and improved C preprocessor manual.
- A large number of grammar errors, spelling errors, typos, and thinkos have been corrected throughout the entire documentation set.
- The manual pages for `gcc`, `cpp`, and `gcov` are now generated automatically from the master Texinfo manual, which eliminates the problem of manual pages being out of date with respect to the release versions. The generated manual pages are incomplete extracts of the full manual. The full manual is provided in Texinfo form, so you can generate `info`, HTML, and other soft copies of the full manual. You can also generate a printed manual from the Texinfo source.
- Generated `info` files are included in the release tarballs along with the Texinfo sources, which eliminates problems with building `makeinfo` as part of the GCC distribution.

Additional Changes in GCC 3.0.1

The following list identifies changes that went into GCC 3.0.1, in addition to the many changes in the initial GCC 3.0 release:

- C++ fixes for incorrect code-generation
- Improved cross-compiling support for the C++ standard library
- Fixes for some embedded targets that worked in GCC 2.95.3, but not in GCC 3.0
- Fixes for various exception-handling bugs in the C++ compiler
- A port to the S/390 architecture

Additional Changes in GCC 3.0.2

The following list identifies changes that went into GCC 3.0.2, which incorporated all of the changes through GCC 3.0.1:

- Fixes to correct generation of bad code during loop unrolling and sibling call optimization passes
- Small enhancements to the quality of code generated for x86 CPUs
- Additional features for IA64 code generation
- Elimination of many buglets

Additional Changes in GCC 3.0.3

The following list identifies changes that went into GCC 3.0.3, which incorporated all of the changes through GCC 3.0.2:

- Continued fixes for code generation on multiple architectures
- Fixes to the C++ compiler that corrected code generation problems, crashes in the demangler, debugging information in classes, and buffer overflows in the C++ standard library
- Reversal of a modification that inadvertently changed the PowerPC ABI
- Numerous minor fixes and tweaks for many architectures

Additional Changes in GCC 3.0.4

The following list identifies changes that went into GCC 3.0.4, which incorporated all of the changes through GCC 3.0.3:

- Still more repairs to code generation in the C++ compiler
- Generation of correct debugging information for functions with code in multiple files (that is, functions generated by yacc)
- Various documentation updates
- Repairs to the runtime exception handler
- Support for newer (current) versions of NetBSD on the x86 platform
- New support for Tensilica's Xtensa CPU

Additional Changes in GCC 3.1

The following list identifies changes that went into GCC 3.1, which incorporated all of the changes through GCC 3.0.4:

- A number of bugs on various NetBSD platforms have been fixed.
- The Java compiler is considerably faster than in previous releases and also works in parallel makes (using `make -j`).
- Profile-driven optimization has been added to the compiler (`-fprofile-arcs` and `-fbranch-probabilities`).
- Inline functions are subject to better optimization because the compiler sees them earlier than in previous releases.
- Another new built-in, `_builtin_prefetch`, has been added, enabling programmers explicitly to request instruction prefetching, giving the compiler and the CPU additional opportunities for code optimization.
- Using the `-g3` switch, the compiler can be instructed to emit debugging information for macros on platforms that support the DWARF2 debug format.
- The preprocessor, `cpp`, is faster and uses less memory than the preprocessor in the 3.0.x series of releases.
- A number of changes in the C++ ABI have been implemented, resolving some known bugs.
- The Java runtime library, `libgcj`, now includes RMI and other key Java classes.
- The Java compiler, `gcj`, includes new optimizations, uses built-in functions for certain mathematics methods, and is also compliant with JDK 1.2 and Unicode 3.0.
- Ada support, in the form of the GNAT Ada 95 front end (from Ada Core Technologies), has been released. The Ada front end is not completely integrated, but its addition gives GCC an ISO Ada standards-compliant Ada compiler.
- MMIX, the hypothetical CPU architecture used in Donald Knuth's *The Art of Computer Programming* series (Addison-Wesley, 1998. ISBN: 0-201-148541-9.), is now supported in GCC.

- New target CPU support for CRIS, the Hitachi SuperH SH5, 64-bit mode on the UltraSPARC, and 64-bit support for the PowerPC.
- x86 support has been improved and extended. These enhancements include support for 64-bit AMD CPUs, support for MMX, 3DNow!, SSE, and SSE2 instruction sets on the appropriate CPUs, specific targets (using `-march=arch` and `-mcpu=cpu`) for Pentium MMX, Pentium III, Pentium 4, K6-2, K6-3, and Athlon 4 CPUs, and streamlined float-to-integer conversion code.
- Support for a long list of ports has been declared obsolete, meaning that source code supporting these ports will be removed from the GCC tree when version 3.2 is released.
- The manual *Using and Porting the GNU Compiler Collection* has been split into two separate manuals, *Using the GNU Compiler Collection* and *GNU Compiler Collection Internals*, and both have undergone significant rewriting. These manuals are part of the full GCC distribution.

Additional Changes in GCC 3.1.1

- The optimization option `-fprefetch-loop-arrays` no longer fetches random blocks of memory on non-x86 architectures.
- The Java compiler's speed has been significantly improved and the compiler also works with parallel makes.
- For NetBSD on MIPS platforms, support for nested functions was fixed and missing routines for floating-point support were implemented.
- The `-traditional` command-line option for the C preprocessor was marked deprecated and targeted for removal in GCC 3.3.

Additional Changes in GCC 3.2

GCC 3.2 proved to be a bug-fix series, and significant efforts went into addressing real and perceived performance problems in the compiler. The increased functionality of the 3.x compiler brought with it corresponding performance drops, so after considerable debate on the mailing list, developers worked simultaneously to extend the feature set in one branch of the GCC tree while closing bugs and improving performance in the 3.2 tree. The initial release of GCC 3.2 included an ABI change, so the minor release number was bumped from 3.1.

3.2.1, like 3.2 before it, was a bug fix release. One bug fix from 3.2 was reverted (backed out) because it proved to be non-reentrant. Over 50 bugs were retired. 3.2.2 and 3.2.3 saw additional bugs squashed, the final total for the 3.2.x series approaching 200.

Additional Changes in GCC 3.3

- A number of obsolete systems were removed from the compiler, simplifying maintenance and reducing the overall size of the source tree. Notable systems removed include the Convex and Clipper chips, the Intel i860 chips, Sun's picoJava processor, and almost all of the Motorola 88000 series processors.
- For functions that accept pointers to other functions as arguments, you can use the function attribute `nonnull` to indicate arguments that must be passed with `nonnull` values. The compiler will emit a warning when a `null` value is passed for such an argument.
- A new file format for profile output using `-fprofile-arcs` means that `gcov` for versions of GCC before 3.3 cannot understand profiles generated by 3.3 and newer GCC versions. Similarly, `gcov` version 3.3 is not backwardly compatible with the file format supported by older versions of `gcov`.
- The method used to construct search lists for include files (using the `-I` option) has been modified. If a directory specified with `-I` is a standard system include directory, the preprocessor ignores it in an attempt to preserve the default search order and to preserve the special handling associated with system header files.
- The support for NetBSD on SH3 and SH5 systems has been extended considerably.
- The Java SQL packages (`java.sql` and `javax.sql`) support JDBC 3.0, the assert feature of JDK 1.4 works, and the bytecode interpreter has been speeded up.
- Some 300 additional bugs have been closed.

Caveats and Gotchas

- Enumerations are now properly promoted to `int` in function parameters and function returns. Normally this change is not visible, but when using `-fshort-enums`, this is an ABI change.

- An undocumented extension that allowed C programs to have a label at the end of a compound statement now generates a warning. Scheduled for removal in a future version of GCC, you can avoid the warning by adding a null statement after the label.
- A poorly documented extension that allowed string constants in C, C++, and Objective C to contain unescaped newlines has been targeted for removal in a future version of GCC. As a result, programs that use this extension should be fixed. To do so, the bare newline may be replaced by `\n` or prefixed with `\n\`, or string concatenation can be used to concatenate `\n"`, the bare newline, and a `"` placed at the beginning of the next line.
- The `Chill` compiler is unmaintained because no one volunteered to convert it to use garbage collection.
- The `iostream` methods `filebuf::attach`, `ostream::form`, and `istream::gets` are not included in `libstdc++` version 3.0.
- The `-traditional` compiler option, removed in GCC 3.3, still works for non-C languages.
- As of GCC 3.1, systems that use the ELF binary format default to using the DWARF2 debugging format, which requires GDB 5.1.1 or newer. Solaris, however, does not default to DWARF2 debugging format. DWARF1 debugging format is no longer supported.
- The release and development versions of GDB do not fully support the new C++ ABI. This might be fixed in GDB 5.3.

CHAPTER 2

Installing GCC on DOS and Windows Platforms

FOR BETTER OR WORSE, Windows is the most commonly used desktop operating system, at least on Intel x86-based PCs. Unlike Linux and the various BSD Unix clone operating systems (FreeBSD, NetBSD, and OpenBSD), few Windows systems come with a full suite of compiler tools already installed or easily available *for free*. Over the years, several different solutions for getting GCC on Windows and DOS systems have developed. This chapter looks at two sets of tools and utilities that install a GCC-based development toolchain on DOS and Windows systems: Cygwin and DGJPP.

Installing Cygwin

The Cygwin tools discussed in this section use native Win32 ports of GCC and other GNU development tools, ordinarily referred to as a *toolchain*, combined with a key *dynamic link library* (DLL) to provide a fairly complete emulation of the Unix API. The toolchain provides the Unix look and feel; the DLL provides the Unix functionality. Generally speaking, Cygwin allows greater interoperability between Windows installations and Unix and Linux systems and is marketed (by Red Hat Software) as a migration path from Linux to Windows.

Downloading and Installing Cygwin

To download the free version of Cygwin, point your Web browser at the Cygwin home page, <http://www.cygwin.com/> (or, if that does not work, <http://sources.redhat.com/cygwin/>). By far the easiest way to install Cygwin, if you have Internet access, and preferably, a fast Internet connection, is to perform a network-based installation that downloads the necessary files from the Internet using a small program, `setup.exe`. The following steps walk you through the download and installation process:

1. From the Cygwin home page, click the Install Cygwin now link, shown in Figure 2-1, and save the `setup.exe` file to a location of your choice.

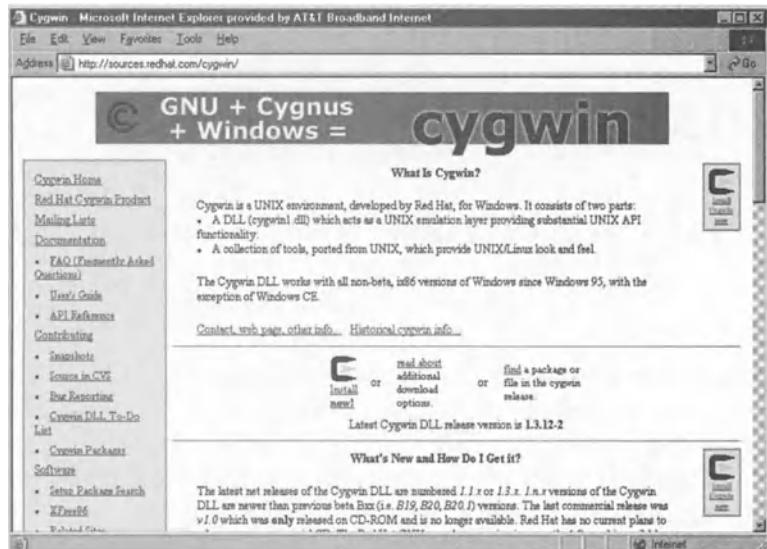


Figure 2-1. Install Cygwin now link

2. Double-click the setup.exe file. You should see the Cygwin Net Release Setup Program Wizard, shown in Figure 2-2. Click the Next button to get started.

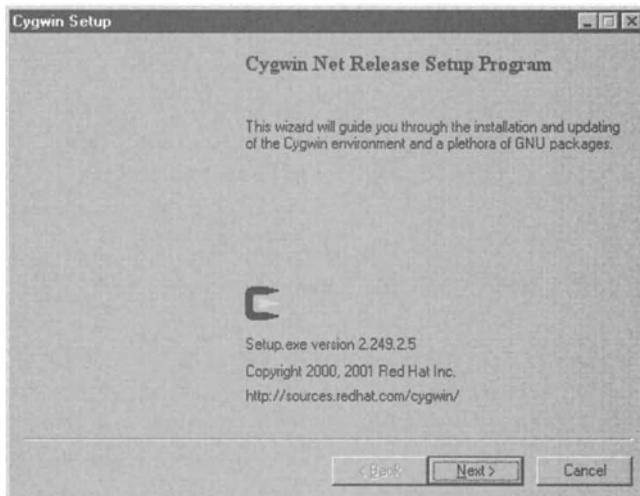


Figure 2-2. The Cygwin Setup dialog box

3. Click the Install from Internet radio button, and then click the Next button (see Figure 2-3). This feature is here so you can download Cygwin, but install it later if you want. If you selected the second option, you would

run the setup again and perform an installation from the local directory. This option is important if you are working with a slow or unreliable Internet connection or you want to install Cygwin on a bunch of machines in the office.

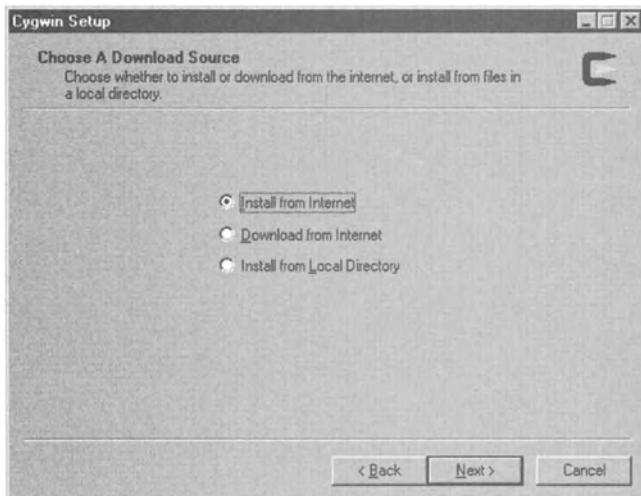


Figure 2-3. The Choose Installation Type dialog box

3. In the Select Root Install Directory dialog box, shown in Figure 2-4, select a *root directory*, which is where Cygwin will be installed. In Figure 2-4, we have used the default root directory, C:\cygwin.

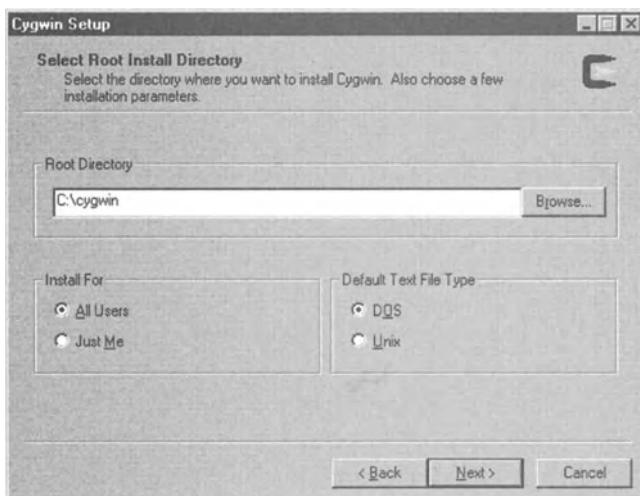


Figure 2-4. The Choose Installation Directory dialog box

4. Decide whether you want all users to be able to use Cygwin, or just yourself, and click the corresponding radio button in the Install For frame (see Figure 2-4).
5. If you will only be using Windows, click the DOS radio button in the Default Text File Type frame. Otherwise, click the Unix radio button. Click Next to continue the installation process (see Figure 2-4). The issue here is that Unix and Windows use different conventions to denote the end of a line that are not compatible. Windows uses a carriage return followed by a line feed (\r\n); Unix uses only a line feed (\n). If you are doing a lot of cross-platform work, pick Unix line endings; some issues with GCC may arise when your mounts use DOS-style line endings. Click the Next button to continue.
6. In the Select Local Package Directory dialog box, choose a directory in which the downloaded files will be stored after downloading. Figure 2-5 shows that we chose C:\My Documents\Downloads\cygwin. Click the Next button to continue.

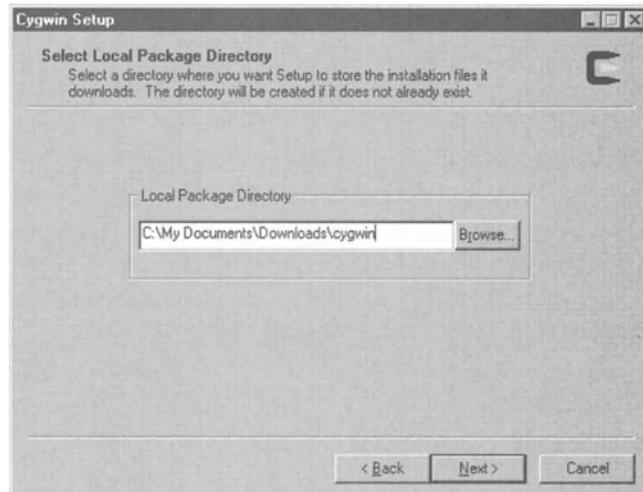


Figure 2-5. The Select Local Package Directory dialog box

7. Specify the type of Internet connection you have in the Select Your Internet Connection dialog box. As shown in Figure 2-6, we selected the Direct Connection radio button because we have a broadband connection to the Internet. Select the connection most appropriate for your situation, and then click the Next button to continue.

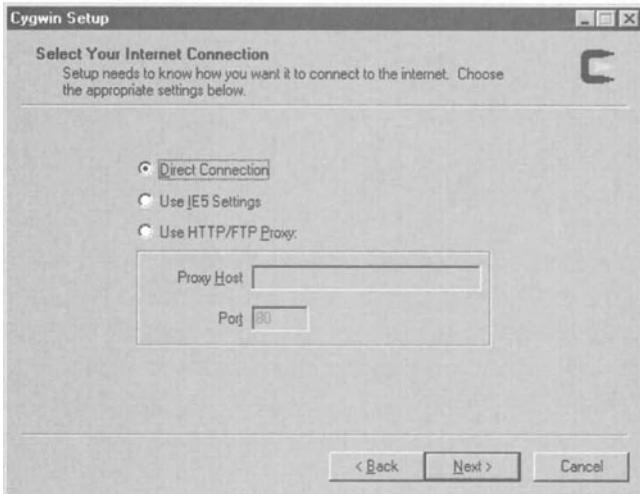


Figure 2-6. The Select Connection Type dialog box

- At this juncture, the installer downloads a list of mirror sites (see Figure 2-7) from which you should select a site closest to you. It might take a few minutes to download the complete list, depending on the speed of your Internet connection and how busy the server is. Scroll through the list, select the site you want, and then click the Next button to continue. If you linger on this dialog box for more than a few minutes, you will tickle a bug that causes the installer to say the connection to the server timed out. Quit and restart the installer to get around this bug.

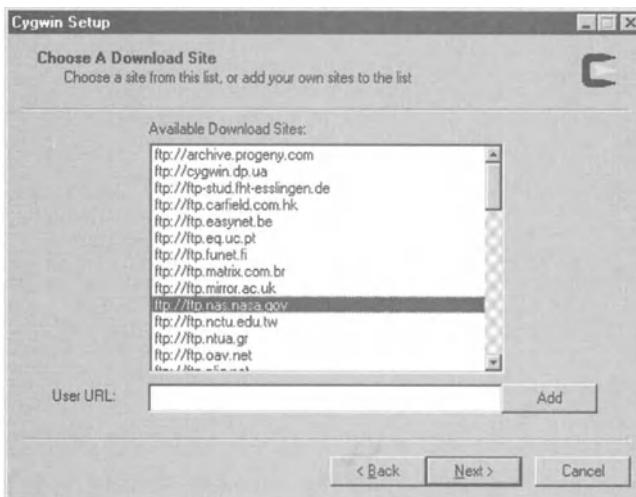


Figure 2-7. The Choose Download Site(s) dialog box

TIP If you want to use a download site not listed, type the site's URL into the User URL text box and then click the Add button, which adds the URL you typed to the list and automatically selects it for you. Click the Next button to continue.

9. In the Select Packages dialog box, select the packages you want to install (see Figure 2-8). The package list might take a little while to display because the installer must download the entire list of packages that are available from the site you selected. To keep things simple, we recommend sticking with the packages in the following list. If you wish, you can add other packages after the installation is complete. After making your package selections, click the Next button to start the download. You may not be able to download the very latest and greatest package for a Cygwin system.
 - All of the packages in the Base category (these should have been preselected for you)
 - The binutils and gcc packages from the Devel category

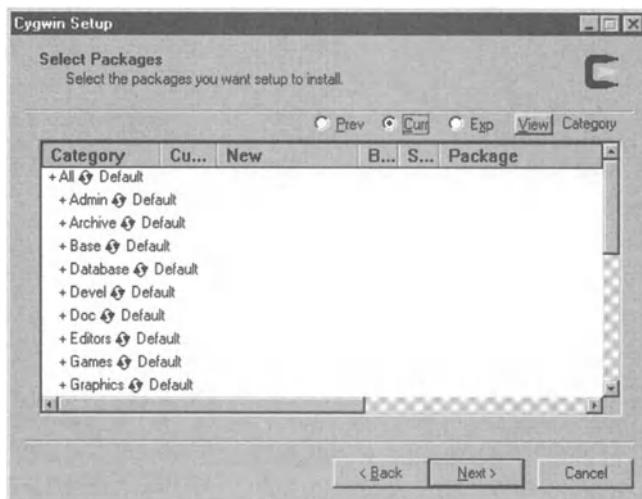


Figure 2-8. The Select Packages dialog box

10. After the download completes, the installer installs the necessary files and offers to create Desktop and Start Menu icons for you. If you do not want the icons, clear the appropriate checkboxes. Click the Finish button to complete the installation. Figure 2-9 shows the Create Icons dialog box.

When you click the Finish button, the setup program opens and closes several DOS windows as it completes the installation process.

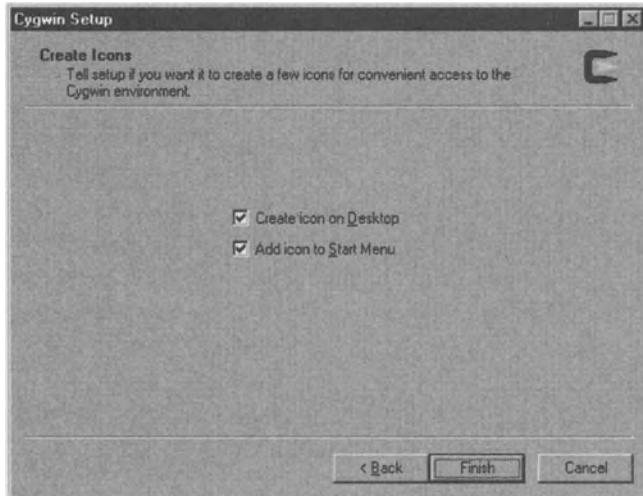


Figure 2-9. The Create Icons dialog box

11. Click OK in the Installation Complete dialog box, shown in Figure 2-10, to complete the installation. Although not required, you may want to reboot your system before continuing the installation and configuration process. Your Cygwin installation is complete.

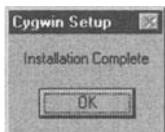


Figure 2-10. The Installation Complete dialog box

The next section discusses using Cygwin in DOS and Windows environments. There is little post-installation configuration you must perform, because the Cygwin installer does most of it for you.

Using Cygwin in DOS and Windows Environments

As we commented earlier in this chapter, the Cygwin GNU toolchain gives you a complete GNU development environment that enables you to use GCC and other popular GNU programming tools in a Win32 environment. The basic Cygwin installation we described in the last section installed a functioning, no-frills

compilation environment, but little else. At a bare minimum, you might want to install an editor, such as Vi Improved (Vim), Emacs, or nano, a pico clone (for those of you familiar with the popular Pine e-mail client). For a fuller environment, especially if you require compatibility with Linux or Unix systems or if you are more comfortable (or want to *become* more comfortable) with Linux or Unix systems, you can even install an XFree86 system, an implementation of the X Window System for Intel x86 architectures. It is important to understand that Cygwin *does not* create a GUI-based Windows IDE for programming with GCC. Rather, it emulates Linux and Unix command-line development environments, even if you are using the Cygwin port of XFree86. Accordingly, you will need to be comfortable working in a command-line environment.

NOTE *If you are interested in a Windows-based graphical development environment, have a look at TimeSys Corporation's TimeStorm tool. Information about TimeStorm is available at http://www.timesys.com/index.cfm?bdy=tools_bdy_timestorm.cfm.*

After the Cygwin installation is complete, and if you created the Desktop icon as described in Step 10 of the Cygwin installation procedure in the previous section, you should have an icon on the desktop labeled Cygwin. Double-clicking that icon starts a Cygwin shell session in a DOS window on your Windows desktop. It will look startlingly like a Linux or Unix shell session because, for all practical purposes, that is precisely what it is: a POSIX development session running on a Win32 platform (see Figure 2-11).

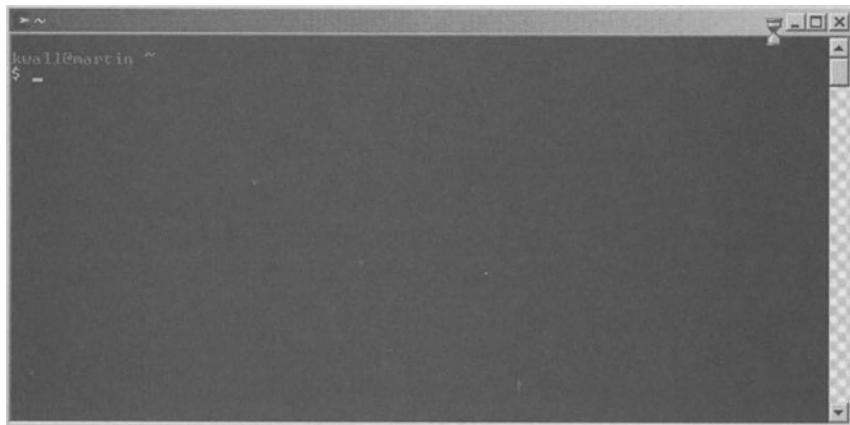


Figure 2-11. A Cygwin session on Windows 2000

It is far beyond this book's scope to teach you how to work in a POSIX (that is, Linux or Unix) environment, however. Nevertheless, the material in the rest of

this book describing how to use GCC and its associated tools is applicable regardless of the underlying platform. When GCC's behavior differs between the various platforms discussed in this book, these differences will be noted and described. The next chapter starts the process of showing how to use GCC.

CAUTION According to the Cygwin developers, users have reported problems running Cygwin on Windows Server 2003. The Cygwin developers are researching the issues. Meanwhile, if you are using Windows Server 2003, be aware that difficulties might arise.

Installing DJGPP

Another popular, though aging, GCC-based development system is DJGPP, D.J. Delorie's free 32-bit development environment for DOS systems. Like Cygwin, DJGPP is a development environment, not just a compiler, because it includes both the core GCC compiler that is this book's focus and a large set of GNU tools that enable you to produce 32-bit protected-mode programs that run on MS-DOS and MS-Windows machines and on any DOS-compatible operating system. DJGPP and programs compiled with it look like standard DOS programs, relying on DOS and the PC BIOS services for input, output, keyboard handling, disk access, and so on. Most of the code in programs built using DJGPP use 32-bit *DOS Protected Mode Interface* (DPMI), giving DJGPP programs access to all of the physical RAM installed in your system using a flat (rather than segmented) memory model, just like in advanced operating systems. Similarly, the DPMI interface also permits DJGPP programs to use the host system's virtual memory. Finally, because DJGPP uses DPMI, protected mode code may issue DOS and BIOS calls from protected mode.

NOTE DJGPP is an acronym for DJ's GNU Programming Platform. DJ Delorie is the primary author and maintainer of DJGPP, which explains where the DJ in DJGPP originated.

The implication of this is that DJGPP programs will run on any system that can execute DOS programs and that provides DPMI services, including but not limited to the following:

- Microsoft's MS-DOS
- Lineo's DR-DOS

- IBM's PC-DOS

- Novell's NW-DOS

- FreeDOS

- Windows 3.x

- Windows 9x

- Windows ME

- Windows NT

- Windows 2000

- Windows XP

- OS/2

- Linux DOSEmu

Although DJGPP programs will run on all of these systems, they will be considered as DOS programs. That is, DJGPP-compiled programs on Windows or OS/2 systems will not be graphical and will not use the Win16 or Win32 API. They will only run in a DOS box or in an actual DOS session. However, DJGPP programs running on Windows 9x systems and on Lineo's DR-DOS *do* support long filenames.

Work to complete DJGPP to Windows 2000 and Windows XP is in progress. Although it works well, according to the authors, efforts to extend this support continue. For the latest information, see the status page for DJGPP on Windows 2000 and Windows XP at <http://clio.rice.edu/djgpp/win2k/main.htm>.

NOTE *Using the RSXNT package together with DJGPP, you can compile Windows applications. For more information, see "Creating Windows Applications" later in this chapter.*

Naturally, the heart of DJGPP is a port of GCC for DOS, but it also includes key utilities such as the assembler and link editor, make, and a Texinfo browser. Another key DJGPP component is the C library. The complete environment is actively developed, maintained, and extended. Once you have installed a functioning DJGPP development environment, you

might find it useful to experiment with some of the many DJGPP-related tools, libraries, and programs available. A list of such projects exists on the DJGPP home page at <http://www.delorie.com/djgpp/dl/elsewhere.html>. You can also scour the DJGPP Webring, starting at the Webring's home page, <http://www.geocities.com/SiliconValley/Vista/6552/djring.html> (watch out for the annoying pop-up ads, though).

Downloading and Installing DJGPP

Because most people lack a functioning C compiler on their system, the typical way to get started with DJGPP is to download a binary version from the DJGPP Web site (<http://www.delorie.com/djgpp/>). Even if you have a working compiler, you might still want to begin with a binary version to avoid problems. By far the simplest way to get started is to use the DJGPP Zip Picker to help you decide what to download and from where to download it. The following steps walk you through the process:

1. Point your Web browser at the DJGPP Zip File Picker page (<http://www.delorie.com/djgpp/zip-picker.html>), shown in Figure 2-12.

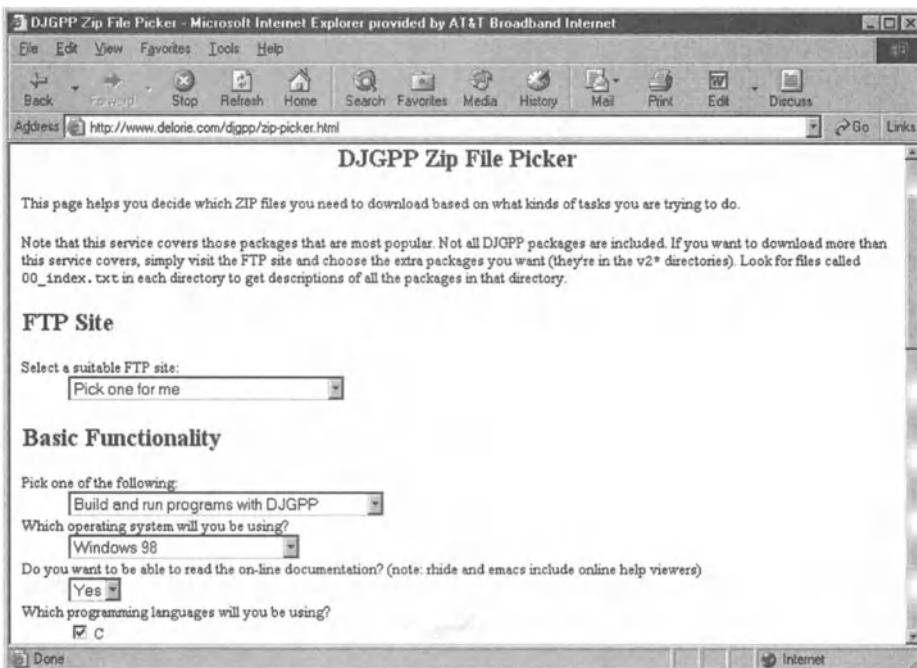


Figure 2-12. The DJGPP Zip File Picker page

2. Select an FTP site close to you or use the Pick one for me option to allow the Zip File Picker page to select a download site for you.
3. Choose the option that best describes what you intend to do with DJGPP. You have three options:
 1. Build and run programs with DJGPP.
 2. Run a program I got that is built with DJGPP.
 3. Use Unix to build DOS programs.

You want the complete environment because, presumably, you want to be able to compile *and* run programs, so select Build and run programs with DJGPP.

4. Select the operating system under which you will be running DJGPP.
5. If you want to be able to read the online (Texinfo) documentation, select Yes from the corresponding drop-down box. If you intend to download one of the IDEs (see Step 7), select No here. Otherwise, select Yes (and read Chapter 10).
6. Select the programming languages you want to use. Your options include the following:
 - C
 - C++
 - Objective C
 - Assembler
 - Bison
 - Flex

Unless you know you need C++ and/or Objective C support, we recommend selecting C, Assembler, Bison, and Flex in order to keep the download time to a minimum. If you have a fat pipe, select them all.

7. Decide whether you want to use an integrated development environment (IDE) and, if so, which one. The options include RHIDE, an environment very similar to the IDE in older Borland compilers, Emacs with everything, and Emacs with just the stuff you cannot live without. To keep the download small, you can skip the IDE and use one of the editors already installed on your system.
8. Choose Yes from the drop-down box for downloading the GNU debugger, GDB. You will need GDB if you want to be able to debug programs.
9. Select the additional utilities you want to download. Table 2-1 lists and briefly describes the available options.

Table 2-1. Optional DJGPP Components

Component	Description
Pakke	A DJGPP installer
Source code	Source code for each item you download
Extra documentation formats	Documentation in texi, DVI, HTML, and PostScript format
Allegro	A graphics, sound, and keyboard library for gaming
GRX	A library of basic graphics primitives for DJGPP (including a port of the Borland graphics library to DJGPP)
RSX	A library for Windows GUI programming
Unix Curses Emulator	A curses emulation library for text mode GUIs

Figure 2-13 shows the completed form. We have opted to let the picker select a download site for us, chosen a complete DJGPP environment for a Windows 2000/XP system, selected C and C++ for our languages, skipped the IDE, included the debugger, and selected no additional packages from the Extra Stuff section.

The screenshot shows a Microsoft Internet Explorer window with the title "DJGPP Zip File Picker - Microsoft Internet Explorer provided by AT&T Broadband Internet". The address bar shows the URL "http://www.delorie.com/djgpp/zip-picker.html". The page content is a form for selecting DJGPP files. It includes fields for selecting an FTP site ("Select a suitable FTP site: US, Pennsylvania (tphost.simtel.net)"), basic functionality preferences ("Build and run programs with DJGPP", "Which operating system will you be using? Windows 98", "Do you want to be able to read the on-line documentation? (Yes)", and programming language preferences (C, C++, Objective C, Assembler, Bison, Flex)). It also includes sections for "Integrated Development Environments and Tools" with options for RHIDE, Emacs, and Emacs (smaller font).

Figure 2-13. The completed Zip Picker form

10. After reviewing your selections, click the Tell me which files we need button to see the list of files you need to download and to get highly condensed installation instructions (see Figure 2-14). We recommend printing or saving the instructions for future reference.

The `unzip32.exe` file is a utility you can use to extract all of the other files. Be sure to download this file if you do not have another file extraction program such as PKZip or WinZip installed on your system.

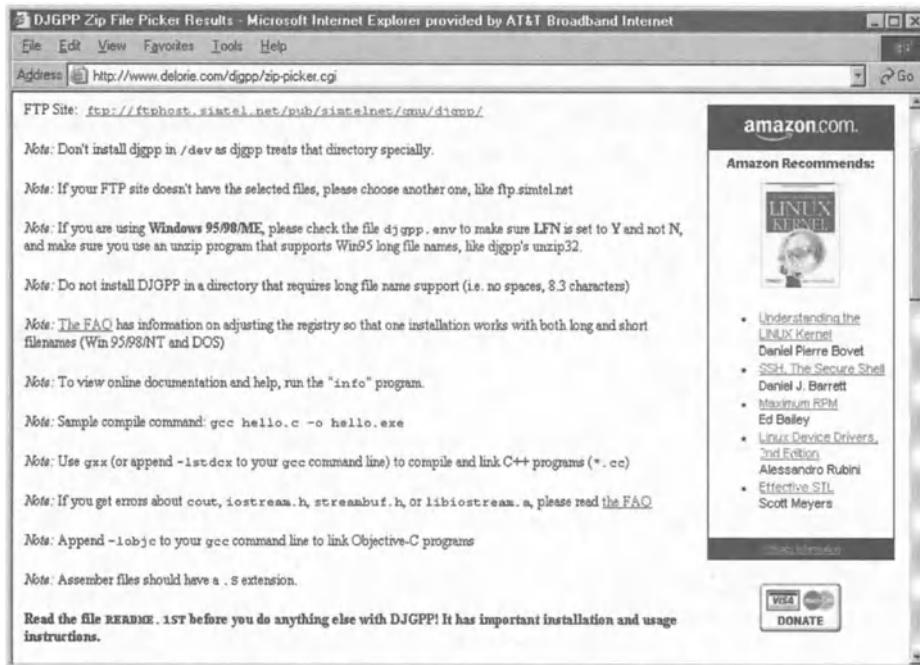


Figure 2-14. The download list and instructions generated by the Zip Picker

11. Create a directory in which to store the downloaded files. We used C:\DOWNLOADS.
12. Using a method of your choosing, download the listed files into the directory created in Step 11.
13. After you finish downloading the listed Zip files, open a DOS window, create a directory, say, C:\DJGPP, in which to install DJGPP, make the newly created directory your current directory, and then unzip the Zip files using the following commands:

```
C:\DJGPP> copy c:\downloads\unzip32.exe .
C:\DJGPP> unzip32 -o c:\downloads\*.zip
[...]
C:\DJGPP>
```

Replace the drive and directory specifications in the example with values that reflect the choices you made. First, we copied the unzip32.exe command into the installation directory we wanted to use. Then we unzipped all of the files into the installation directory. The option **-o** silently overwrites files that have the same name—this is safe to do as

long as you create the installation directory from scratch. The bracketed ellipsis ([...]) indicates output from the `unzip32.exe` command that is not shown in order to save space. The `unzip32.exe` command unzips the files into the current directory, which is why you need to make it your current directory before unpacking the archives.

Configuring DJGPP

Configuring DJGPP is relatively simple. You need to set a couple of environment variables. You can use the following procedure (for Windows NT-based systems):

1. Right-click My Computer, and then select Properties.
2. Click the Advanced tab.
3. Click the Environment Variables button to open the Environment Variables dialog box, shown in Figure 2-15.

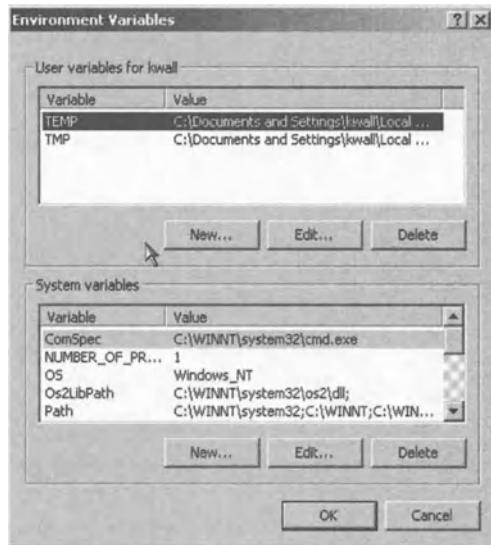


Figure 2-15. The Environment Variables Dialog Box

4. In the System variables pane, select the Path variable and click the Edit button.
5. In the Edit System Variable dialog box, add the following text to the end of the Variable's Value text box:

`;C:\DJGPP\BIN`

The result is shown in Figure 2-16. This step adds C:\DJGPP\BIN to the Windows' PATH environment variable, enabling you to use the programs in that directory without having to type a long command.

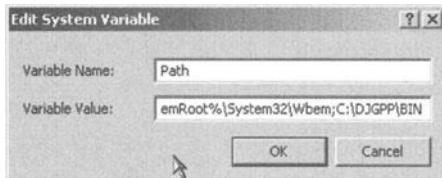


Figure 2-16. Modifying the Path variable

6. Click OK to save your changes.
7. In the System variables pane, click the New button to open the New System Variable dialog box.
8. In the New System Variable dialog box, type **DJGPP** in the Variable Name text box and type **C:\DJGPP\DJGPP.ENV** in the Variable Value text box (see Figure 2-17).

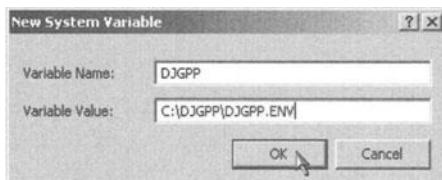


Figure 2-17. The New System Variable dialog box

This step sets the environment variable DJGPP to the value shown. DJGPP's various programs and utilities need the contents of DJGPP.ENV in order to operate properly.

9. Click OK to save the change, OK again to close the Environment Variables dialog box, and OK one last time to close the System Properties dialog box.

The next time you open a command window, you can see that these variables have been set, as shown here:

```
C:\>echo %DJGPP%
C:\DJGPP\DJGPP.ENV
C:\>echo %PATH%
C:\WINNT\System32;C:\WINNT;C:\WINNT\System32\Wbem;C:\DJGPP\BIN
```

If you lack administrative access on your NT-based system, you must add the new PATH entry to the User variables section, or modify the existing one (if it exists). Similarly, nonadministrator users must add a new DJGPP variable to the User variables section.

If you prefer not to modify the environment as described, you can create a DJGPP shortcut to a batch file. Create a batch file named C:\DJGPP\DJGPP.BAT with the following contents:

```
@echo off
set PATH=c:\djgpp\bin;%PATH%
set DJGPP=c:\djgpp\djgpp.env
chdir c:\djgpp\mystuff
command
```

Then create a shortcut to this batch file.

Using DJGPP in DOS and Windows Environments

So, whaddya get with DJGPP? Generally speaking, you get a full GNU development environment that effectively duplicates for DOS and Windows what you get (with a lot less work, frankly) on a Unix or Linux system. In addition to the core compiler and associated utilities (the assembler, preprocessor, link editor, librarian, and loader), you get a number of programmer's toys that have proven to be indispensable, such as gprof, make, strip, strings, and gdb.

Under the hood, DJGPP compensates for many of the deficiencies of a 16-bit operating system that lacks the familiar semantics of more powerful, full-featured operating systems. For example, the length of a DOS command line is limited to 128 characters. Even compiling a simple program is effectively impossible with so short a command line, so DJGPP uses an *in-core* (in memory) transfer buffer to store and pass long command lines around as necessary.

In terms of *using* DJGPP, you should be able to use the DJGPP compiler and tools for DOS and Windows using the instructions provided in this book. Features of GCC discussed in the following chapters that do not work, that work differently, or that require special handling on DOS and Windows will be noted in the text. When possible, workarounds or alternatives will be provided.

Creating Windows Applications

If you want to write true Windows apps, you will have to use auxiliary tools or another compiler. The most mature options are RSXNTDJ, Cygnus GNU-Win32, MinGW, and U/WIN. RSXNTDJ is an add-on package for DJGPP that enables you to develop Win32 programs using the DJGPP development environment. It includes a good IDE. Unfortunately, it relies on the Microsoft SDK for the Win32

header files, which means that the RSXNTDJ patches to the header files break each time Microsoft updates the SDK. For more information about RSXNTDJ, see the GNU C/C++ projects for Win32 and DOS Web pages at <http://www.mathematik.uni-bielefeld.de/~rainer/>.

MinGW, Minimalist GNU for Windows, is another native Win32 port of the GNU development tool chain. Unlike the Cygwin tools, which use an external, third-party DLL, MinGW uses the Microsoft-supplied C runtime library, CRTDLL.DLL or MSVCRT.DLL, for all system services. The implication of the reliance on Microsoft's runtime is that you get very little Unix API compatibility and only a thin veneer of POSIX compliance. Another shortcoming of MinGW is that it does not use the newer versions of GCC—the latest released version is based on GCC 2.95.2.

Another entry in the Win32-capable GCC tool suite is AT&T Research's U/WIN project. Like Cygwin, it uses a DLL (POSIX.DLL) to provide a POSIX-compliant API. Additional libraries provide Unix compatibility. Programs linked with POSIX.DLL run in the Win32 subsystem, which enables them to use both POSIX and Win32 APIs, affording such programs greater portability with Unix and Linux programs while retaining full Win32 compatibility. For more information, see the U/WIN Web pages at <http://www.research.att.com/sw/tools/uwin/>.

CHAPTER 3

Basic GCC Usage

THIS CHAPTER'S GOAL IS to get you comfortable with typical GCC usage. It goes behind the scenes and dissects the half-dozen or so GCC command-line options you use by habit. When you invoke GCC to compile a source code file, the compilation process passes through as many as four stages: preprocessing, compilation, assembly, and linking. You will learn in this chapter how to stop the compilation process at any one of these stages. GCC also accepts many additional options that control the behavior of the preprocessor, the linker, and the assembler, so this chapter shows you the ins and outs of these magic GCC incantations, too. Other options discussed in this chapter allow you to modify the directory search path GCC uses when it runs and also enable you to exercise greater control over the content format of GCC's diagnostic messages. You will also learn how to tell GCC which dialect of C, such as strict ANSI/ISO C or good ole Kernighan and Ritchie (K&R) C, it should expect. Finally, you will learn a variety of command-line options you can use when compiling C++ code.

General Options

Various GCC command-line options can be used to display basic or extended usage tips, display compiler configuration information, or control overall behavior. These options defy easy categorization, so we have lumped them all under the “General Options” heading in order to avoid using the more common “Miscellaneous Options” heading. Adroit handwaving notwithstanding, though, the options we discuss in this section just do not fit neatly into any other category, so we dumped them here.

GCC accepts both single-letter options, such as `-o`, and multiletter options, such as `-ansi`. The consequence of GCC accepting both types of options is that, unlike many GNU programs, you cannot group multiple single-letter options. For example, the multiletter option `-pg` is not the same as the two single-letter options `-p -g`. `-pg` creates extra code in the final binary that outputs profile information for the GNU code profiler, `gprof`. `-p -g`, on the other hand, generates extra code in the resulting binary that outputs 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 GCC's sensitivity to the grouping of multiple single-letter options, you are generally free to mix options and arguments. 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.
```

We wrote “generally free to mix options and 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.

Many options have long names starting with `-f` or with `-W`. Examples include `-fforce-mem`, `-fstrength-reduce`, `-Wformat`, and so on. Similarly, most of these long name options have both positive and negative forms. Thus, the negative form of `-fstrength-reduce` would be `-fno-strength-reduce`.

NOTE *The GCC manual documents only the nondefault version of long name options that have both positive and negative forms. That is, if the GCC manual documents `-mfoo`, the default is `-mno-foo`.*

Table 3-1 lists and briefly describes the options that fall into the miscellaneous category.

Table 3-1. General GCC Options

Option	Description
<code>-dumpmachine</code>	Displays the compiler’s target CPU
<code>-dumpspecs</code>	Displays GCC’s default spec strings
<code>-dumpversion</code>	Displays the compiler version number
<code>--help</code>	Displays basic usage information
<code>-pass-exit-codes</code>	Causes GCC to return the highest error code generated by any failed compilation phase
<code>-pipe</code>	Uses pipes to send information between compiler processes rather than intermediate files

Table 3-1. General GCC Options (continued)

Option	Description
<code>-print-file-name=lib</code>	Displays the path to the library <i>lib</i>
<code>-print-libgcc-file-name</code>	Displays the name of the compiler's companion library
<code>-print-multi-directory</code>	Displays the root directory for all versions of libgcc
<code>-print-multi-lib</code>	Displays the maps between command-line options and multiple library search directories
<code>-print-prog-name=prog</code>	Displays the path to program <i>prog</i>
<code>-print-search-dirs</code>	Displays the directory search path
<code>-save-temp</code>	Saves intermediate files created during compilation
<code>--target-help</code>	Displays help for command-line options specific to the compiler's target
<code>-time</code>	Displays the execution time of each compilation subprocess
<code>-v</code>	Displays the programs and arguments invoked as the compiler executes
<code>-V ver</code>	Invokes version number <i>ver</i> of the compiler
<code>--version</code>	Displays the compiler version information and short license

Spec strings are macro-like constructs that GCC uses to define the paths and default options and arguments for the various components it calls during compilation. We will discuss spec strings in greater detail in the section titled “Customizing GCC Using Spec Strings” in Chapter 4. The `-dumpversion` and `-dumpmachine` options (which cannot be specified together, by the way) show the compiler’s version number and the processor for which it outputs code, respectively.

When `-dumpversion` and `-dumpmachine` are on the same line, only the first argument gets processed.

```
$ gcc -dumpmachine
i686-pc-linux-gnu
$ gcc-dumpversion
3.2
$ gcc --version
```

```
gcc (GCC) 3.2
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The `--version` argument displays more verbose version information. We find `-dumpversion` and `-dumpmachine` are best used in scripts because their output is terse and easily parsed using standard shell utilities. The output from the `-dumpmachine` option may vary between two versions of GCC running on the same system. For example, the default compiler on Slackware Linux version 8.1 is GCC 2.95.3. When passed `-dumpmachine`, it emits `i386-slackware-linux`. GCC version 3.2, compiled and installed as described in Chapter 1 of this book, emits `i686-pc-linux-gnu` when invoked with the `-dumpmachine` option. The various `-print-` options listed in Table 3-1 enable you to determine the paths to the libraries and programs that GCC uses at runtime. This information can be useful when you are trying to track down a problem or want to be sure that a particular library is being used or a specific directory is being searched during the compilation process.

If GCC encounters an error during any compilation phase, it exits and returns an error code of 1 to the calling process. Using the `-pass-exit-codes` option instructs GCC to return the highest error code generated by any compilation phase, rather than 1. If you want to know how long the compiler runs in each phase of the compilation process, specify the `-time` option. The `-time` option can be especially instructive when compiling GCC itself. In fact, this option is used as a rough gauge of GCC's overall performance by the GCC developers: the faster that GCC can compile itself, the better the developers like it. You might also find it interesting to use `-time` when compiling the Linux kernel or any other large program that really stresses a system.

The `-pipe` option uses pipes rather than temporary files to exchange data between compilation phases. However, if you are short on disk space, do not use `-pipe` to try to conserve disk space because that is not the option's purpose. Rather, the issue is speed. As *interprocess communication* (IPC) mechanisms, pipes are faster than files because pipes avoid the overhead of file I/O.

TIP *If you try to use `-pipe` and `-time` together, GCC emits a warning. So do not use `-pipe` and `-time` in the same GCC invocation.*

Controlling GCC's Output

As you probably know, compiling source code potentially passes through as many as four stages: preprocessing, compilation itself, assembly, and linking.

The first three stages, preprocessing, compilation, and assembly, occur at the level of individual source files. The end result of the first three stages is an object file. Linking combines object files into the final executable. GCC evaluates filename suffixes to select the type of compilation it will perform. Table 3-2 maps filename suffixes to the type of compilation GCC performs.

Table 3-2. GCC Operations by Filename Suffix

Suffix	Operation
.c	C source code to preprocess
.C	C++ source code to preprocess
.cc	C++ source code to preprocess
.cpp	C++ source code to preprocess
.cxx	C++ source code to preprocess
.i	C source code not to preprocess
.ii	C++ source code not to preprocess
.m	Objective C source code
.h	C header file (not to compile or link)
.s	Assembly code
.S	Assembly code to preprocess

A filename with no recognized suffix is considered an object file to be linked. GCC's failure to recognize a particular filename suffix does not mean you are limited to using the suffixes listed previously to identify source or object files. You can use the `-x lang` option to identify the language used in one or more input files. *lang* tells GCC the input language to use, and can be `c`, `objective-c`, `c++`, `c-header`, `cpp-output`, `c++-cpp-output`, `assembler`, or `assembler-with-cpp`. Table 3-3 lists the command-line options you can use to exercise more control over the compilation process.

Table 3-3. GCC Output Options

Option	Description
-c	Stops the compilation process before the link stage
-E	Terminates compilation after preprocessing
-o <i>file</i>	Writes output to the file specified by <i>file</i>
-S	Stops the compilation process after generating assembler code
-x <i>lang</i>	Sets the input language of subsequent files to <i>lang</i>
-x none	Turns off the definition of a previous -x <i>lang</i> option

When you use -c, the output is a link-ready object file, which has an .o filename extension. For each input file, GCC generates a corresponding output file. Likewise, if you specify -E, the resulting output will be the preprocessed source code, *which is sent to standard output*. If you want to save the preprocessed output, you should redirect it to a file (either with redirection or using the -o option). If you use the -S option, each input file results in an output file of assembly code with an .s extension. The -o *file* option enables you to specify the output filename, overriding the default output filename conventions.

Given a single source file, myprog.c, compiling it using GCC is simplicity itself. Just invoke gcc, passing the name of the source file as the argument.

```
$ gcc myprog.c
$ ls -l
-rwxr-xr-x    1 kwall    users        13644 Oct  5 16:17 a.out
-rw-r--r--    1 kwall    users         220 Oct  5 16:17 myprog.c
```

NOTE These examples use bold entries to indicate the output from each compilation phase.

The result on Linux and Unix systems is an executable file in the current directory named a.out, which you execute by typing ./a.out in the directory containing the file. The name a.out is a historical artifact dating from C's earliest days. It stands for *assembler output* because, as you might expect, the first C-based executables were the output of assemblers. On Cygwin systems, you will wind up with a file named .a.exe that you can execute by typing either ./a or ./a.exe in the directory containing the file. If you are using DJGPP, you will have the files a.out and a.exe. To execute it, simply type a.

To define the name of the output file GCC uses, use the -o option, as illustrated in the following example:

```
$ gcc myprog.c -o runme
$ ls -l
-rw-r--r--  1 kwall    users        220 Oct  5 16:17 myprog.c
-rwxr-xr-x  1 kwall    users      13644 Oct  5 16:28 runme
```

As you can see, GCC creates an executable file named `runme` in the current directory. The usual convention when compiling a single source file to executable format is to name the executable by dropping the `.c` extension, so that `myprog.c` becomes `myprog`. Naturally, only the simplest programs consist of only a single source code file. More typically, programming projects consist of multiple source code files. Obviously, in such a situation, you need to use the `-o` option in order to name the resulting binary, unless you intend to stick with the default `a.out` name.

Of course, you will also want to know how to compile multiple source files using GCC. Again, the magic incantation is simple. To illustrate, suppose you have a source file `showdate.c` that uses a function declared in `helper.h` and defined in `helper.c`. The standard way to compile these files, ignoring optimization, debugging, and other special cases, is

```
$ gcc showdate.c helper.c
```

Again, invoked as shown, GCC deposits the final executable in a file named `a.out` on Linux and Unix systems (`a.exe` on Cygwin systems, and both `a.out` and `a.exe` on DJGPP systems). In the absence of command-line options instructing otherwise, GCC goes through the entire compilation process: preprocessing, compilation, assembly, and linking. To specify the name of the output file, use the `-o` option.

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

This invocation, using `-o showdate`, leaves the compiled and linked executable in the file named, you guessed it, `showdate`.

Using other options listed in Table 3-3 can be instructive. If you want to stop compilation after preprocessing using the `-E` option, be sure to use `-o` to specify an output filename or use output redirection. For example:

```
$ gcc -E helper.c -o helper.i
$ ls -l helper.*
-rw-r--r--  1 kwall    users        210 Oct  5 12:42 helper.c
-rw-r--r--  1 kwall    users         45 Oct  5 12:29 helper.h
-rw-r--r--  1 kwall    users      40440 Oct  5 13:08 helper.i
```

The `-o helper.i` option saves the preprocessed output (or, if you wish, the post-preprocessor output) in the file `helper.i`. Notice that the preprocessed file is some 200 times larger than the source file. Also bear in mind that, except for the link stage, GCC works on a file-by-file basis—each input file results in a corresponding output file with a filename extension appropriate to the stage at which compilation is stopped. This latter point is easier to see if you use the `-S` or `-c` option, as the following example illustrates:

```
$ gcc -S showdate.c helper.c
$ ls -l
total 20
-rw-r--r--  1 kwall    users        210 Oct  5 12:42 helper.c
-rw-r--r--  1 kwall    users        45 Oct  5 12:29 helper.h
-rw-r--r--  1 kwall    users      741 Oct  5 13:18 helper.s
-rw-r--r--  1 kwall    users      208 Oct  5 12:44 showdate.c
-rw-r--r--  1 kwall    users      700 Oct  5 13:18 showdate.s
```

In this case, we use the `-S` option, which stops compilation after the assembly stage and leaves the resulting assembly code files, `helper.s` and `showdate.s`, which are the assembled versions of the corresponding C source code files. The next example uses `-c` to stop GCC after the compilation process itself:

```
$ gcc -c showdate.c helper.c
$ ls -l
total 20
-rw-r--r--  1 kwall    users        210 Oct  5 12:42 helper.c
-rw-r--r--  1 kwall    users        45 Oct  5 12:29 helper.h
-rw-r--r--  1 kwall    users     1104 Oct  5 13:22 helper.o
-rw-r--r--  1 kwall    users      208 Oct  5 12:44 showdate.c
-rw-r--r--  1 kwall    users     1008 Oct  5 13:22 showdate.o
```

Finally, the following example shows how to use the `-x` option to force GCC to treat input files as source code files of a specific language. First, rename `showdate.c` to `showdate.txt` and then attempt to compile and link the program as shown here:

```
$ gcc showdate.txt helper.c -o showdate
showdate.txt: file not recognized: File format not recognized
collect2: ld returned 1 exit status
```

As you might expect, GCC does not know how to “compile” a `.txt` file and compilation fails. To remedy this situation, use the `-x c` option to tell GCC that

the input files following the `-x` option (`showdate.c` and `helper.c`) are C source files and to disregard the filename extension it does not recognize.

```
$ gcc -xc showdate.txt helper.c -o showdate
$ ls -l
total 28
-rw-r--r-- 1 kwall users 210 Oct 5 12:42 helper.c
-rw-r--r-- 1 kwall users 45 Oct 5 12:29 helper.h
-rwxr-xr-x 1 kwall users 13893 Oct 5 13:38 showdate*
-rw-r--r-- 1 kwall users 208 Oct 5 12:44 showdate.txt
```

Lo and behold, it worked! In fact, judicious use of the `-x` option with the `-c`, `-E`, and `-S` options enables you to exercise precise control over the compilation process. Although you do not ordinarily need to exercise such fine control over the process, you can walk through a complete compilation process one step at a time. First, preprocessing:

```
$ gcc -E helper.c -o helper.pre
$ gcc -E showdate.c -o showdate.pre
$ ls -l
total 92
-rw-r--r-- 1 kwall users 210 Oct 5 12:42 helper.c
-rw-r--r-- 1 kwall users 45 Oct 5 12:29 helper.h
-rw-r--r-- 1 kwall users 40440 Oct 5 13:44 helper.pre
-rw-r--r-- 1 kwall users 208 Oct 5 12:44 showdate.c
-rw-r--r-- 1 kwall users 37152 Oct 5 13:46 showdate.pre
```

We use the `-o` option to save the output in files with `.pre` filename extensions. Next, run the preprocessed files through the assembler:

```
$ gcc -S -x cpp-output helper.pre -o helper.as
$ gcc -S -x cpp-output showdate.pre -o showdate.as
$ ls -l
total 100
-rw-r--r-- 1 kwall users 741 Oct 5 13:47 helper.as
-rw-r--r-- 1 kwall users 210 Oct 5 12:42 helper.c
-rw-r--r-- 1 kwall users 45 Oct 5 12:29 helper.h
-rw-r--r-- 1 kwall users 40440 Oct 5 13:44 helper.pre
-rw-r--r-- 1 kwall users 700 Oct 5 13:47 showdate.as
-rw-r--r-- 1 kwall users 208 Oct 5 12:44 showdate.c
-rw-r--r-- 1 kwall users 37152 Oct 5 13:46 showdate.pre
```

This time, we use the `-o` option to save the assembler output using the file-name extension `.as`. We use the `-x cpp-output` option because the assembler expects preprocessor output files to have the extension `.i` (for C source code). Now, run the assembly code though actual compilation to produce object files.

```
$ gcc -c -x assembler helper.as showdate.as
$ ls -l
total 108
-rw-r--r-- 1 kwall users 741 Oct 5 13:47 helper.as
-rw-r--r-- 1 kwall users 210 Oct 5 12:42 helper.c
-rw-r--r-- 1 kwall users 45 Oct 5 12:29 helper.h
-rw-r--r-- 1 kwall users 1104 Oct 5 13:50 helper.o
-rw-r--r-- 1 kwall users 40440 Oct 5 13:44 helper.pre
-rw-r--r-- 1 kwall users 700 Oct 5 13:47 showdate.as
-rw-r--r-- 1 kwall users 208 Oct 5 12:44 showdate.c
-rw-r--r-- 1 kwall users 1008 Oct 5 13:50 showdate.o
-rw-r--r-- 1 kwall users 37152 Oct 5 13:46 showdate.pre
```

We use the `-x assembler` option to tell GCC that the files `helper.as` and `showdate.as` are assembly language files. Finally, link the object files to create the executable, `showdate`.

```
$ gcc helper.o showdate.o -o showdate
$ ls -l
total 124
-rw-r--r-- 1 kwall users 741 Oct 5 13:47 helper.ass
-rw-r--r-- 1 kwall users 210 Oct 5 12:42 helper.c
-rw-r--r-- 1 kwall users 45 Oct 5 12:29 helper.h
-rw-r--r-- 1 kwall users 1104 Oct 5 13:50 helper.o
-rw-r--r-- 1 kwall users 40440 Oct 5 13:44 helper.pre
-rwxr-xr-x 1 kwall users 13891 Oct 5 13:51 showdate
-rw-r--r-- 1 kwall users 700 Oct 5 13:47 showdate.ass
-rw-r--r-- 1 kwall users 208 Oct 5 12:44 showdate.c
-rw-r--r-- 1 kwall users 1008 Oct 5 13:50 showdate.o
-rw-r--r-- 1 kwall users 37152 Oct 5 13:46 showdate.pre
```

It should not take too much imagination 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 7).

So what was the point of this exercise? Well, it illustrates that the compiler performs as advertised. More importantly, the `-E` option can be remarkably useful when you are trying to track down a problem with a macro that does not behave as you expected. A popular C programming subgenre consists of pre-processor magic, sometimes referred to as *preprocessor abuse* or, as we like to refer to it, *Stupid Preprocessor Tricks* (with apologies to David Letterman). The typical scenario is that the preprocessor does not interpret your macro as you anticipated, causing compilation failure, error or warning messages, or bizarre runtime errors. By halting compilation after preprocessing, you can examine the output, determine what you did wrong, and then correct the macro definition.

The value of `-S` becomes apparent if you want to hand-tune the assembly code the compiler generates. You can also use `-S` to see what kind of assembly output the compiler creates for a given block of code, or even a single statement. Being able to examine the compiler's assembly level output is educational in its own right, and it also allows you to debug a program with a subtle bug. Naturally, to get the maximum benefit from GCC's assembly output feature, you have to know the target system's assembly language, or, as is more often the case, have a ready supply of the target CPU's reference manuals close at hand.

Compiling C Dialects

GCC 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?” we hear you asking. 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 you get options for supporting multiple dialects of C. We will explain *how* to use them a little later in this section.

In addition to the subtle variations that exist in standard C, some of GCC's 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 3-4 lists and briefly describes the options that control the C dialect to which GCC adheres during compilation.

Table 3-4. 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 <i>file</i>	Saves prototypes and other identifying information about functions declared in a translation unit to the file identified by <i>file</i>
-fallow-single-precision	Prevents promotion of single-precision operations to double-precision
-fbuiltin	Recognizes built-in functions that lack the <code>_builtin_</code> 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
-fhosted	Claims that compilation takes place in a hosted environment
-fno-asym	Disables use of <code>asm</code> , <code>inline</code> , and <code>typeof</code> as keywords, allowing their use as identifiers
-fno-builtin	Ignores built-in functions that lack the <code>_builtin_</code> prefix
-fno-signed-bitfields	Indicates that bit fields of undeclared type are to be considered unsigned
-fno-signed-char	Keeps the <code>char</code> type from being signed, as in the type <code>signed char</code>
-fno-unsigned-bitfields	Indicates that bit fields of undeclared type are to be considered signed
-fno-unsigned-char	Keeps the <code>char</code> type from being unsigned, as in the type <code>unsigned char</code>
-fshort-wchar	Forces the type <code>wchar_t</code> to be short unsigned int
-fsigned-bitfields	Indicates that bit fields of undeclared type are to be considered signed

Table 3-4. C Dialect Command-Line Options (continued)

Option	Description
-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= <i>value</i>	Sets the language standard to <i>value</i> (c89, iso9899:1990, iso9899:199409, c99, c9x, iso9899:1999, iso9989:199x, gnu89, gnu99)
-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? We certainly were when we first started sorting this all out, but it breaks down more simply than it seems. To begin with, throw out -aux-info and -trigraphs, because you are unlikely ever to need them. Similarly, we advise you 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 Chapter 4 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 might not be on another one.*

You will rarely ever need to worry about the `-fhosted` and `-ffreestanding` options, but for completeness' sake, we will 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 free-standing implementation (when invoked with `-ffreestanding`), GCC 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 we have already discussed, we are left with `-ansi`, `-std`, `-traditional`, `-traditional-cpp`, `-fno-asm`, `-fbuiltin`, and `-fno-builtin`. Here again, we can simplify matters somewhat. `-traditional`, which enables you to use features of pre-ISO C, 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 3-4 that `-traditional` also implies `-fno-asm`, because `-fno-asm` disables the use of the `inline` and `typeof` keywords, like `-traditional`, and also the `asm` keyword. In traditional 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 the ones defined in the standard C libraries. Invoking the `-fno-builtin` option disables this behavior. The GCC info pages document the complete list of GCC 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 3-4 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 option `-std=c89` or `-std=iso9899:1990` has the same effect as `-ansi`. However, using any of these three option 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 option `-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=iso9899:1999`. Again, to see all of the diagnostic messages required by the C99 standard, use `-pedantic` or `-pedantic-errors` as just 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 specifically request this dialect by specifying `-std-gnu89`. If you want C99 mode with GNU extensions, 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

- Disabling the `asm` and `typeof` keywords
- Enabling trigraphs and digraphs
- Disabling predefined macros, such as `unix` or `linux`, that identify the type of system in use
- Disabling the use of `//` single-line comments in C code in C89 mode (C99 permits `//` single-line comments)
- Defining the macro `_STRICT_ANSI_`, used by header files and functions to enable or disable certain features that are or are not C89-compliant
- Disabling built-in functions that conflict with those defined by the ISO standard
- Disabling all GNU extensions that conflict with the standard (Chapter 4 discusses GNU extensions to C and C++ in detail.)

Using GCC with C++

Although you can use most GCC options regardless of programming language, some are specific to C++. The GCC C++ compiler is ordinarily invoked as `g++`; although many installations also install `g++` as `c++`, just as many installations install `cc` as a synonym for `gcc`. As discussed earlier in the chapter, GCC recognizes `.C`, `.cc`, `.cp`, `.cpp`, `.cxx`, and `c++` as suffixes for C++ source files and `.ii` as the suffix for preprocessed C++ source code. When GCC encounters a file with one of these suffixes, it treats the file as a C++ file, even if GCC was invoked using the `gcc` command. Nonetheless, `gcc` (that is, the C compiler) does not understand the complete chain of dependencies, such as class libraries, that C++ programs often require and it does not know how to compile C++ code, so we advise using `g++` (or `c++` if you are in environments that require this name) to invoke GCC's C++ compiler.

Compiling C++ code using GCC is, perhaps unsurprisingly, quite similar to compiling C code using GCC. For example, given the C++ program in Listing 3-1, the simplest way to compile it is shown here:

```
$ g++ simple.cc
```

Listing 3-1. Simple C++ Program

```
#include <iostream>

int main (void)
{
    cout << "a simple C++ program" << endl
    return 0;
}
```

As with the C compiler, unless you specify the output filename using `-o`, `g++` leaves the compiled program in `a.out` (or `a.exe` on Cygwin systems). In fact, the input and output filename options described for the C compiler also apply to the C++ compiler. We are more interested in this section in the C++-specific command-line options listed in Table 3-5.

Table 3-5. C++-Specific Command-Line Options

Option	Description
<code>-fcheck-new</code>	Ensures that the pointer returned by operator <code>new</code> is not null before accessing the allocated storage
<code>-fconserve-space</code>	Puts global variables initialized at runtime and uninitialized global variables in the common segment, conserving space in the executable
<code>-fdollars-in-identifiers</code>	Permits the \$ symbol in identifiers (the default)
<code>-fms-extensions</code>	Allows G++ to omit warnings about nonstandard idioms in the Microsoft Foundation Classes (MFC)
<code>-fno-access-control</code>	Disables access checking
<code>-fno-const-strings</code>	Forces G++ to assign string constants the <code>char *</code> type, even though the ISO C++ mandates <code>const char *</code>
<code>-fno-elide-constructors</code>	Instructs G++ always to call the copy constructor rather than using a temporary to initialize another object of the same type
<code>-fno-enforce-eh-specs</code>	Disables runtime checks for violations of exception handling
<code>-fno-for-scope</code>	Extends the scope of variables declared in a <code>for</code> initialization statement to the end of the enclosing scope, contrary to the ISO standard

Table 3-5. C++-Specific Command-Line Options (continued)

Option	Description
-fno-gnu-keywords	Disables use of <code>typeof</code> as keyword, so it can be used as an identifier
-fno-implement-inlines	Saves space by not creating out-of-line copies of inline functions qualified by <code>#pragma implementation</code>
-fno-implicit-inline-templates	Saves space by not creating implicit instantiations of inline templates (see <code>-fno-implicit-templates</code>)
-fno-implicit-templates	Saves space by only creating code for explicit instantiations of out-of-line (<code>noninline</code>) templates
-fno-nonansi-builtins	Disables use of built-ins not required by the ANSI/ISO standard
-fno-operator-names	Disables the use of keywords as synonyms for operators (<code>and</code> , <code>bitand</code> , <code>bitor</code> , <code>compl</code> , <code>not</code> , <code>or</code> , <code>xor</code>)
-fno-rtti	Disables generation of runtime type identification information for classes with virtual functions
-fpermissive	Converts errors about nonconformant code to warnings, allowing compilation to continue
-frepo	Allows template instantiation to occur automatically at link time
-ftemplate-depth- <i>N</i>	Prevents template instantiation recursion going deeper than the integral value of <i>N</i>
-fuse-cxa-atexit	Registers destructors for static objects with <code>__cxa_atexit</code> rather than <code>atexit</code>
-nostdinc++	Disables searching for header files in standard directories <i>specific to C++</i>

GCC, rather G++, recognizes other options specific to C++, but these options deal with optimizations, warnings, and code generation, so we will discuss them in the appropriate locations. In particular, the section “Enabling and Disabling Warning Messages” addresses C++-specific warning options and Chapter 5 covers C++-specific optimization options.

Controlling the Preprocessor

The options discussed in this section let you control GCC's preprocessor—the *new* preprocessor, we might add. As mentioned earlier in the chapter, compilation stops after preprocessing if you specify the `-E` option to `gcc`. As you know, the preprocessor executes against each source code file *before* the files are handed off to the compiler proper. Preprocessor options are listed in Table 3-6.

Table 3-6. Preprocessor Options

Option	Description
<code>-Dname</code>	Defines the preprocessor macro <i>name</i> with a value of 1
<code>-Dname=def</code>	Defines the preprocessor macro <i>name</i> with the value specified in <i>def</i>
<code>-Uname</code>	Undefines any preprocessor macro <i>name</i>
<code>-undef</code>	Undefines all system-specific macros, leaving common and standard macros defined

The `-Uname` option cancels the definition of the macro *name* that was defined on the GCC command line using `-D` or defined in one of the source code files. Each instance of `-D` and `-U` is evaluated in the order specified on the command line. If you use the `-imacros file` option to specify which macros defined in *file* should be included, this inclusion takes place *after* all `-D` and `-U` options have been evaluated.

CAUTION *Do not put spaces between -D and -U and their arguments, or the definition will not work.*

Consider Listing 3-2. If the preprocessor macro `DEUTSCH` is defined, the output message will be “Hallo, Welt!” Otherwise, the message will be “Hello, World!”

Listing 3-2. hallo.c

```
#include <stdio.h>

int main (void)
{
```

```
#ifdef DEUTSCH
    puts ("Hallo, Welt!");
#else
    puts ("Hello, World!");
#endif
    return 0;
}
```

In the following example, we leave DEUTSCH undefined, so the program outputs the English language greeting:

```
$ gcc hallo.c -o hallo
$ ./hallo
Hello, World!
```

Specifying `-DDEUTSCH` on the GCC command line, however, defines the `DEUTSCH` macro with a value of 1, causing the compiled binary to issue the German version:

```
$ gcc hallo.c -o hallo -DDEUTSCH
$ ./hallo
Hallo, Welt!
```

With a little command-line or Makefile magic, deftly implemented macros, and the `-D` and `-U` options, you can enable and disable program features without having to edit your code simply by enabling and disabling preprocessor macros when you recompile a program. Of course, overuse of `#ifdef...#endif` blocks in code can make the code unreadable, so use them sparingly.

Modifying the Directory Search Path

GCC makes it possible for you to manipulate its directory search path. Table 3-7 lists the command-line options that make this manipulation possible.

Table 3-7. Options for Modifying Directory Search Paths

Option	Description
<code>-B prefix</code>	Instructs the compiler to add <i>prefix</i> to the names used to invoke its executable subprograms (such as <code>cpp</code> , <code>cc1</code> , <code>as</code> , and <code>ld</code>), libraries, and header files
<code>-I dir</code>	Adds <i>dir</i> to the list of directories searched for header files
<code>-I-</code>	Limits the type of header files searched for when <code>-I dir</code> is specified
<code>-L dir</code>	Adds <i>dir</i> to the list of directories searched for library files
<code>-specs=file</code>	Reads the compiler spec file <i>file</i> after reading the standard spec file, making it possible to override the default values of arguments passed to GCC component programs

If you use `-I dir` to add *dir* to the include directory search list, GCC prepends *dir* to the standard include list. Thus, local and custom header files are searched first, enabling you to override system definitions and declarations, if you so choose. More often, however, you use `-I` to add directories to the header file search path rather than to override already defined functions. For example, if you are compiling a program that uses header files installed in `/usr/local/include/libxml2`, you would specify this extra directory as shown in the following example (the ellipsis indicates other arguments omitted for the sake of brevity and clarity):

```
$ gcc -I/usr/local/include/libxml2 [...] resize.c
```

This invocation causes GCC to look in `/usr/local/include/libxml2` for any header files included in `resize.c` *before* it looks in the standard header file locations.

The default behavior of the `-I` option is to search the specified directories for both system header files, those included using angle brackets (for example, `#include <net/inet.h>`), and user header files, those included using double quotes (for example, `#include "log.h"`). Use the `-I-` syntax to modify this behavior. All included directories specified with `-I` *before* the occurrence of `-I-` will only be searched for user header files (those included using double quotes). Any include directories specified with `-I` after the occurrence of `-I-` will be searched for all header files. You can think of directories specified after `-I-` as reverting to GCC's default search behavior. Consider the following `#include` directives at the top of a source code file, say, one named `resize.c`:

```
#include "libxml2/xmllops.h"
#include <netdev/devname.h>
```

Suppose further that you want to compile this using the following gcc invocation:

```
$ gcc -I /usr/local/include/libxml2 -I- /usr/local/include/netdev resize.c
```

The header file `xmllops.h` is included as a user header file and, because the directory `/usr/local/include/libxml2` appears on the command line before `-I`, the extra directory will be searched and `xmllops.h` will be found. The directory `/usr/local/include/netev`, on the other hand, appears after `-I-`, so the header file `devname.h` will be found, too. Now, consider what would happen if the `#include` directives appeared as shown next:

```
#include <libxml2/xmllops.h>
#include <netdev/devname.h>
```

In *this* case, the `xmllops.h` header file is included as a *system* header file. As a result, because the directory `/usr/local/include/libxml2` appears before `-I-`, it will only be searched for header files included as user header files, so the pre-processor will *not* find `xmllops.h`. As before, though, the preprocessor will find `devname.h` because it is located in a directory specified after the `-I-` option, which causes the search behavior to revert to the default.

TIP *Multiple -I dir options can be specified. If so, they are searched in the order specified, reading left to right.*

`-I-` also has the unfortunate side effect of disabling the use of the current directory for files included using `#include "myfile.h"`. This means that user header files in the current directory must be specified after `-I-` or they must be located in a directory specified before `-I-`. Listing 3-3 shows the code for `swapme.c`, which uses a function named `swap()` declared in `myfile.h` (shown in Listing 3-4) in the current directory and defined in `myfile.c`, also in the current directory. Listing 3-5 shows `myfile.c`.

Listing 3-3. swapme.c

```
#include <stdio.h>
#include "myfile.h"

int main (void)
{
    int i = 1;
    int j = 2;

    printf ("%d %d\n", i, j);
    swap (&i, &j);
    printf ("%d %d\n", i, j);

    return 0;
}
```

Listing 3-4. myfile.h

```
void swap(int *, int *);
```

Listing 3-5. myfile.c

```
#include "myfile.h"
void swap (int *i, int *j)
{
    int t = *i;
    *i = *j;
    *j = t;
}
```

To compile the program, use the following command line:

```
$ gcc -Wall swapme.c myfile.c -o swapme
$ ./swapme
1 2
2 1
```

We use the `-Wall` option to make sure that all errors were reported. Run the resulting program if you like, but the point we want to make here is that the `myfile.h` header file is located and included properly. Now, compile using `-I`:

```
$ gcc -Wall swapme.c myfile.c -o swapme -I-
myfile.c:1:20: myfile.h: No such file or directory
swapme.c:2:20: myfile.h: No such file or directory
swapme.c: In function 'main':
swapme.c:10: warning: implicit declaration of function 'swap'
```

Notice that GCC's preprocessor did not find myfile.h and the program did not compile. Compilation failed because using the `-I-` option prevented the pre-processor from looking in the current directory for the user header file myfile.h. To work around this problem, you can usually specify `-I.` to cause the preprocessor to look in the current directory. This workaround is demonstrated next:

```
$ gcc -Wall swapme.c myfile.c -o swapme -I- -I.
$
```

The result is the same if you place `-I.` before `-I-:`

```
$ gcc -Wall swapme.c myfile.c -o swapme -I. -I-
$
```

As you can see, both options result in successful compilations.

NOTE You cannot use `-I-` to disable searching standard system directories. You must use `-nostdinc` to inhibit the use of the standard system include directories.

The `-L dir` option does for library files what `-I dir` does for header files: it adds `dir` to the library directory search list for libraries specified using the `-l` option (discussed in the next section, “Controlling the Linker”). As with the header files, GCC prepends `dir` to the standard library directory search list.

The `-specs=file` option will be discussed in the next chapter in the section titled “Customizing GCC Using Spec Strings,” so what we will say here is that after learning how to use spec strings, you can store multiple spec strings in a file and apply them as a group by replacing `file` with the filename.

Controlling the Linker

As you know, *linking* is the last step in the compilation process and refers to merging various object files into an executable binary. GCC assumes that any file that does not end in a recognized suffix is an object file or a library. Refer to the

list of recognized filename suffixes listed in the section titled “Controlling GCC’s Output” earlier in this chapter if you need a quick refresher. The linker knows how to tell the difference between object files (.o files) and library files (.so shared libraries or .a archive files) by analyzing the file contents. Bear in mind, too, that options for controlling the linker will be ignored if you use the -E, -c, or -S options, which terminate the compiler before the link stage begins (after preprocessing, object file generation, and assembling, respectively). Table 3-8 lists options you can use to control the GCC linker.

Table 3-8. Link Options

Option	Description
-lname	Searches the library named <i>name</i> when linking.
-nodefaultlibs	Specifies not to use the standard system libraries when linking.
-nostartfiles	Ignores the standard system startup files when linking. System libraries are used unless -nostdlib is also specified.
-nostdlib	Specifies not to use the standard system startup files or libraries when linking (equivalent to specifying -nostartfiles -nodefaultlibs).
-s	Strips all symbol table and relocation information from the completed binary.
-shared	Produces a shared object that can then be linked with other objects to form an executable.
-shared-libgcc	Uses the shared libgcc library, if available, on systems that support shared libraries.
-static	Forces linking against static libraries on systems that default to linking with shared libraries.
-static-libgcc	Uses the statically linked libgcc library, if available, on systems that support shared libraries.
-u <i>sym</i>	Behaves as if the symbol <i>sym</i> is undefined, which forces the linker to link in the library modules that define it.
-Wl, <i>opt</i>	Passes <i>opt</i> as an option to the linker.
-Xlinker <i>opt</i>	Passes <i>opt</i> as an option to the linker.

We have already mentioned the -L *dir* option for adding directories to the library search path. The -lname option functions complementarily, allowing you to specify additional library files to search for given function definitions. Each

library specified with `-lname` defines a file named `libname.a`, which is searched for in the standard library search path plus any additional directories specified by `-L` options. The libraries use static *archive files*, files whose members or contents are other object files. The linker processes the archive file by searching it for members that define symbols (function names) that have been referenced but not yet defined.

Thus, the only difference between specifying an object filename (such as `name.o`) and using an `-lname` option is that `-l` embeds *name* between lib and .a and searches several directories for the resulting file. That is, provided you have created the archive file `libmy.a` (using the `ar` program), the following two `gcc` invocations are equivalent:

```
$ gcc myprog.o libmy.o -o myprog
$ gcc myprog.o -o myprog -L. -lmy
```

You can demonstrate this for yourself using Listings 3-3 and 3-4:

1. Compile `myfile.c` and `swapme.c` to object code.

```
$ gcc -c myfile.c
$ gcc -c swapme.c
```

2. Link the resulting object files into the final binary, `swapme`, and then run the program to persuade yourself that it works.

```
$ gcc myfile.o swapme.o -o swapme
$ ./swapme
1 2
2 1
```

3. Delete the binary.

```
$ rm swapme
```

4. Use the `ar` command to create an archive file named `libmy.a` from the `myfile.o` object file.

```
$ ar rcs libmy.a myfile.o
```

5. Link the object file `swapme.o` and the archive file as shown in the following command:

```
$ gcc swapme.o -o swapme -L. -lmy
```

6. Run the program again to convince yourself that it *still* works.

```
$ ./swapme
1 2
2 1
```

It makes a difference where in the command line you specify `-lname` because the linker searches for and processes library and object files in the order they appear on the command line. Thus, `foo.o -lbaz bar.o` searches library file `libbaz.a` after file processing `foo.o` but before processing `bar.o`. If `bar.o` refers to functions in `libbaz.a`, those functions may not be loaded.

If, for some reason, you do not want the linker to use the standard libraries, specify `-nodefaultlibs` and specify `-L` and `-l` options that point at your replacements for functions defining the standard libraries. The linker will disregard the standard libraries and use only the libraries you specified. The standard startup files will still be used, though, unless you also use `-nostartfiles`.

As noted in Table 3-8, `-nostdlib` is the equivalent of specifying both `-nostartfiles` and `-nodefaultlibs`: the linker will disregard the standard startup files and only the libraries you specified with `-L` and `-l` will be passed to the linker.

NOTE When you specify `-nodefaultlib`, `-nostartfiles`, or `-nostdlib`, GCC still might generate calls to `memcmp()`, `memset()`, and `memcpy()` in System V Unix and ISO C environments or calls to `bcopy()` and `bzero()` for BSD Unix environments. These entries are usually resolved by entries in the standard C library (`libc`). You should supply entry points for `memcmp()`, `memset()`, `memcpy()`, `bcopy()`, and `bzero()` when using one of these three linker options.

One of the standard libraries bypassed by `-nostdlib` and `-nodefaultlibs` is `libgcc.a`. `libgcc.a` contains a library of internal subroutines that GCC uses to compensate for shortcomings of particular systems and to provide for special needs required by some languages. As a result, you need the functions defined in `libgcc.a` even when you want to avoid other standard libraries. Thus, if you specify `-nostdlib` or `-nodefaultlibs`, make sure you *also* specify `-lgcc` as well to avoid unresolved references to internal GCC library subroutines.

The `-static` and `-shared` options are mostly used on systems that support shared libraries. GCC supports static libraries on *all* systems that we have ever encountered. Not all systems, however, support shared libraries, often because the underlying operating system does not support dynamic loading.

On systems that provide `libgcc` as a shared library, you can specify `-static-libgcc` or `-shared-libgcc` to force the use of either the static or shared version of `libgcc`, respectively. There are several situations in which an application should

use the shared libgcc instead of the static version. The most common case occurs with C++ and Java programs, especially those that throw and catch exceptions across different shared libraries—all libraries as well as the application itself should use the shared libgcc. Accordingly, the g++ and gcj compiler drivers automatically add -shared-libgcc whenever you build a shared library or a main executable.

On the other hand, the gcc compiler driver does not always link against the shared libgcc, especially when creating shared libraries. The issue is one of efficiency. If gcc determines that you have a GNU linker that does not support the link option --eh-frame-hdr, gcc links the shared libgcc into shared libraries. If the GNU linker *does* support the --eh-frame-hdr option, gcc links with the static version of libgcc. The static version allows exceptions to propagate properly through such shared libraries, without incurring relocation costs at library load time.

In short, if a library or the primary binary will throw or catch exceptions, you should link it using the g++ compiler driver (if you are using C++) or the gcj compiler driver (if you are using Java). Otherwise, use the option -shared-libgcc so that the library or main program is linked with the shared libgcc.

The final option controlling the linker is the rather odd-looking -Wl,*opt*. This tells GCC to pass the linker option specified by *opt* through directly to the linker. You can specify multiple *opt* options by separating each one with a comma (-Wl,*opt1,opt2,opt3*).

Passing Options to the Assembler

If you have options that you want GCC to ignore and pass through to the assembler, use the -Wa,*opt* option. Like its sibling option for the linker, -Wl,*opt*, you can specify multiple *opt* options by separating each option with a comma.

Enabling and Disabling Warning Messages

By way of definition, a *warning* is a diagnostic message that identifies a code construct that might potentially be an error. GCC also emits diagnostic messages when it encounters code or usage that looks questionable or ambiguous. For the sake of discussion, we divide GCC's handling of warning messages into two groups: general options that control the number and types of warnings that GCC emits, and options that affect language features or that are language specific. We will start with the options that control GCC's overall handling of warnings. Many of these options are listed in Table 3-9 (for the complete, mind-numbing list, see Chapter 11).

Table 3-9. General GCC Warning Options

Option	Description
-fsyntax-only	Performs a syntax check but does not compile the code
-pedantic	Issues all warnings required by ISO standards and rejects GNU extensions, traditional C constructs, and C++ features used in C code
-pedantic-errors	Converts warnings issued by -pedantic into errors that halt compilation
-w	Disables all warnings, including those issued by the GNU preprocessor
-W	Displays extra warning messages for certain situations
-Wall	Combines all of the warnings between -Wchar-subscripts and -Wunknown-pragmas
-Wbad-function-cast	Emits a warning if a function call is cast to an incompatible type (C only)
-Wcast-qual	Displays a warning when a typecast removes a type qualifier
-Wchar-subscripts	Emits a warning when a type char variable is used as a subscript
-Wcomment	Emits a warning when nested comments are detected
-Wconversion	Emits a warning if a negative integer constant is assigned to an unsigned type
-Wdisabled-optimization	Displays a warning when a requested optimization is not performed
-Werror	Converts all warnings into hard errors that halt compilation of the indicated translation unit
-Werror-implicit-function-declaration	Emits an error when a function is not explicitly declared before first use
-Wfloat-equal	Emits a warning when floating-point values are compared for equality
-Wformat	Issues a warning when arguments supplied to printf() and friends do not match the specified format string

Table 3-9. General GCC Warning Options (continued)

Option	Description
-Wformat-security	Issues a warning about arguments to <code>printf()</code> and friends that pose potential security problems
-Wimplicit	Combines -Wimplicit-int and -Wimplicit-function-declaration
-Wimplicit-int	Emits a warning when a declaration does not specify a type
-Wimplicit-function-declaration	Emits a warning when a function is not explicitly declared before first use
-Winline	Issues a warning when functions declared <code>inline</code> are not inlined
-Wlarger-than- <i>n</i>	Emits a warning when an object larger than <i>n</i> bytes is defined
-Wmain	Emits a warning if <code>main()</code> 's return type or declaration is malformed
-Wmissing-braces	Displays a warning when aggregate or union initializers are improperly bracketed
-Wmissing-declarations	Displays a warning if a global function is defined without being declared
-Wnested-externs	Issues a warning if an <code>extern</code> declaration occurs in a function definition
-Wno-deprecated-declarations	Disables warnings about use of features that are marked deprecated
-Wno-div-by-zero	Disables warnings issued if (integer) division by zero is detected
-Wno-format-y2k	Disables warnings about <code>strftime()</code> formats that result in two-digit years
-Wno-format-extra-args	Disables warnings about extra arguments to <code>printf()</code> and friends
-Wno-long-long	Disables warnings about using the <code>long long</code> type
-Wno-multichar	Disables warnings issued if multibyte characters are used
-Wpadded	Issues a warning when a structure is padded for alignment purposes

Table 3-9. General GCC Warning Options (continued)

Option	Description
-Wparentheses	Issues a warning about ambiguous or potentially confusing use (or misuse or disuse) of parentheses
-Wpoint-arith	Issues a warning when a code operation or structure depends on the size of function type or a void pointer
-Wredundant-decls	Displays a warning when an object is multiply declared in the same scope, even in contexts in which multiple declaration is permitted and valid
-Wreturn-type	Issues a warning if a function's return type is not specified or if it returns a value but is declared void
-Wsequence-point	Flags code that violates C sequence point rules
-Wshadow	Displays a warning if a locally declared variable shadows another local or global variable, parameter, or a built-in function
-Wsign-compare	Issues a warning when comparisons between signed and unsigned values might produce incorrect results because of type conversions
-Wstrict-prototypes	Displays a warning if a function is defined with specifying argument types (C only)
-Wswitch	Emits a warning about switch statements with enumerated values for unhandled cases
-Wsystem-headers	Issues warnings for code in system headers as if they occurred in your own code
-Wtraditional	Emits a warning about code constructs that behave differently between ISO and traditional C and about ISO C features with no parallel in traditional C (C only)
-Wtrigraphs	Emits warnings if any trigraphs are encountered outside of comment blocks
-Wundef	Issues a warning if an undefined identifier is used in an #if...#endif construct
-Wunused	Emits a warning when automatic variables are used without being initialized
-Wunknown-pragmas	Displays a warning if a #pragma is used that GCC does not recognize

Table 3-9. General GCC Warning Options (continued)

Option	Description
-Wunreachable-code	Emits a warning about code that never executes
-Wunused	Combines all of the listed -Wunused options
-Wunused-function	Issues a warning about declared functions that are never defined
-Wunused-label	Issues a warning about declared labels that are never used
-Wunused-parameter	Issues a warning about declared function parameters that are never used
-Wunused-value	Issues a warning about computed results that are never used
-Wunused-variable	Issues a warning about declared variables that are never used

When you use `-pedantic`, you can also specify `-std=version` to indicate against which version of the standard the code should be evaluated. It is a mistake, however to use `-pedantic` (and, by extension, `-pedantic-errors`), even in combination with `-ansi`, to see if your programs are strictly conforming ISO C programs. This is the case because these options only emit diagnostic messages in situations for which the standard *requires* a diagnostic—the effort to implement a strict ISO parser would be enormous. In general, the purpose of `-pedantic` is to detect gratuitously noncompliant code, disable GNU extensions, and reject C++ and traditional C features not present in the standard.

Another reason that you cannot use `-pedantic` to check for strict ISO compliance is that `-pedantic` disregards the alternate keywords whose names begin and end with `_` and expressions following `_extension_`. As a rule, these two exceptions do not apply to user programs because the alternate keywords and the `_extension_` usage is limited (or should be) to system header files, which application programs should never directly include.

NOTE *Alternate keywords are discussed in Chapter 4.*

A typical source of trouble in C code emerges from the use of functions in the `printf()`, `scanf()`, `strftime()`, and `strfmon()` families. These calls use format strings to manipulate their arguments. Common problems with these function groups include type mismatches between the format strings and their associated

arguments, too many or too few arguments for the supplied format strings, and potential security issues with format strings (known generically as *format string exploits*). The `-Wformat` warnings help you to identify and solve these problems.

`-Wformat` checks all calls to `printf()`, `scanf()`, and other functions that rely on format strings, making sure that the arguments supplied have types appropriate to the format string and that the requested conversions, as specified in the format string, are sensible. Moreover, if you use `-pedantic` with `-Wformat`, you can identify format features not consistent with the selected standard.

Consider the program shown in Listing 3-6 that uses `printf()` to print some text to the screen.

Listing 3-6. printme.c

```
#include <stdio.h>
#include <limits.h>

int main (void)
{
    unsigned long ul = LONG_MAX;
    short int si = SHRT_MIN;

    printf ("%d\n", ul);
    printf ("%ds\n", si);

    return 0;
}
```

There are a couple of problems here. The first `printf()` statement prints the value of the `unsigned long` variable `ul` using an `int` format string (`%d`). The second `printf()` statement prints the `short int` variable `si` using the `%s` format string (that is, as a string). Despite these problems, `gcc` compiles the program without complaint. It even sort of runs.

```
$ gcc printme.c -o printme
$ ./printme
2147483647
Segmentation fault
```

NOTE Depending on your system, this program may not generate a segmentation fault.

Well, *that* was ugly. Now, add the `-Wformat` option and see what happens:

```
$ gcc printme.c -o printme -Wformat
printme.c: In function 'main':
printme.c:9: warning: int format, long int arg (arg 2)
printme.c:10: warning: format argument is not a pointer (arg 2)
$ ./printme
2147483647
Segmentation fault
```

NOTE Depending on the compiler version, your warning output might be slightly different.

This time, the compiler complains about the mismatched types and helpfully tells us where we can find the problems. The program still compiles, but we can use the `-Werror` option to convert the warning to a hard error that terminates compilation:

```
$ gcc printme.c -o printme -Wformat -Werror
cc1: warnings being treated as errors
printme.c: In function 'main':
printme.c:9: warning: int format, long int arg (arg 2)
printme.c:10: warning: format argument is not a pointer (arg 2)
```

This time, compilation stops after the errors are detected. Once we fix the mismatches, the program compiles and runs properly. Listing 3-7 shows the corrected program.

Listing 3-7. printme.c After Corrections

```
#include <stdio.h>
#include <limits.h>

int main (void)
{
    unsigned long ul = LONG_MAX;
    short int si = SHRT_MIN;

    printf ("%ld\n", ul);
    printf ("%s\n", (char *) &si);
```

```

        return 0;
}
$ gcc printme.c -o printme -Wformat -Werror
$ ./printme
2147483647
$
```

Much better, yes?

For more control over GCC's format checking, you can use the options `-Wno-format-y2k`, `-Wno-format-extra-args`, and `-Wformat-security` (only `-Wformat` is included in the `-Wall` roll-up option). By default, if you supply more arguments than are supported by the available format strings, the C standard says that the extra arguments are ignored. GCC accordingly ignores the extra arguments unless you specify `-Wformat`. If you want to use `-Wformat` and ignore extra arguments, specify `-Wno-format-extra-args`.

The `-Wformat-security` is quite interesting. Format string exploits have become popular in the world of blackhats in the last few years. Specifying `-Wformat` and `-Wformat-security` displays warnings about format functions that represent potential security breaches. The current implementation covers `printf()` and `scanf()` calls in which the format string is not a string literal and in which there are no format arguments, such as `printf (var);`. Such code is problematic if the format string comes from untrusted input and contains `%n`, which causes `printf()` to write to system memory. Recall that the conversion specifier `%n` causes the number of characters written to be stored in the `int` pointer (`int *` or a variant thereof)—it is this memory write that creates the security issue.

The group of options that fall under the `-Wunused` category (see Table 3-8 for the list) are particularly helpful. In optimization passes, GCC does a good job of optimizing away unused objects, but if you disable optimization, the unused cruft bloats the code. More generally, unused objects and unreachable code (detected with `-Wunreachable-code`) are often signs of sloppy coding or faulty design. Our own preference is to use the plain `-Wunused` option, which catches all unused objects. If you prefer otherwise, you can use any combination of the five specific `-Wunused-*` options.

Listing 3-8 is a short example of a program with an unused variable.

Listing 3-8. unused.c

```

int main (void)
{
    int i = 10;
    return 0;
}
```

As you can see, the program defines the `int` variable `i`, but never does anything with it. Here is GCC's output when compiling `unused.c` with no options:

```
$ gcc unused.c
```

Well, perhaps we really should have written, “Here is the *lack* of GCC’s output when compiling `unused.c` with no options,” because adding `-ansi -pedantic` does not change the compiler’s output. Here is GCC’s complaint when we use `-Wunused`:

```
$ gcc -Wunused unused.c -ansi -pedantic
unused.c: In function 'main':
unused.c:3: warning: unused variable 'i'
```

Each of the warning options discussed in this section results in similar output that identifies the translation unit and line number in which a problem was detected, and a brief message describing the problem. We encourage you to experiment with the warning options. Given the rich set of choices, you can debug and improve the overall quality and readability of your code just by compiling with a judiciously chosen set of warning options.

Adding Debugging Information

Referring to a program he used to illustrate a point, Donald Knuth once wrote, “Beware of bugs in the above code; we have only proved it correct, not tried it.” Bugs are an inescapable reality of coding. Accordingly, GCC supports a number of options to help you debug code. If you have used GCC before, you are no doubt familiar with the `-g` and `-ggdb` options, which instruct GCC to embed debugging information in executables in order to facilitate debugging. These two options hardly exhaust GCC’s repertoire of debugging support, though in this section we will show you not only the `-g` and `-ggdb` switches, but also other lesser known GCC options that augment the debugging process. Or, to put it more whimsically, this section helps you debug debugging.

We will start with Table 3-10, which lists and briefly describes all of the debugging options GCC supports.

Table 3-10. Debugging Options

Option	Description
<code>-d[mod]</code>	Generates debugging dumps at compilation points specified by <i>mod</i>
<code>-fdump-class-hierarchy [-option]</code>	Dumps the class hierarchy and vtable information, subject to the control expressed by <i>option</i> , if specified
<code>-fdump-translation-unit [-option]</code>	Dumps the tree structure for an entire unit, subject to the control expressed by <i>option</i> , if specified
<code>-fdump-tree[-switch [-option]]</code>	Dumps the intermediate representation of the tree structure, subject to the controls expressed by <i>switch</i> and <i>option</i> , if specified
<code>-fdump-unnumbered</code>	Inhibits dumping line and instruction numbers in debugging dumps (see <code>-d[mod]</code>)
<code>-fmem-report</code>	Displays memory allocation statistics for each compiler phase
<code>-fpretend-float</code>	Pretends that the target system has the same floating-point format as the host system
<code>-fprofile-arcs</code>	Instruments code paths, creating program dumps showing how often each path was taken during program execution
<code>-ftest-coverage</code>	Generates data for the gcov test coverage utility
<code>-ftime-report</code>	Displays performance statistics for each compiler phase
<code>-g[n]</code>	Generates level <i>n</i> debugging information in the system's native debugging format (<i>n</i> defaults to 2)
<code>-gcoff[n]</code>	Generates level <i>n</i> debugging information in the COFF format (<i>n</i> defaults to 2)
<code>-gdwarf</code>	Generates debugging information in the DWARF format
<code>-gdwarf+</code>	Generates debugging information in the DWARF format, using extensions specific to GDB
<code>-gdwarf-2</code>	Generates debugging information in the DWARF version 2 format
<code>-ggdb[n]</code>	Generates debugging information that only GDB can fully exploit
<code>-gstabs[n]</code>	Generates level <i>n</i> debugging information in the STABS format (<i>n</i> defaults to 2)

Table 3-10. Debugging Options (continued)

Option	Description
-gstabs+	Generates debugging information in the STABS format, using extensions specific to GDB
-gvms[n]	Generates level <i>n</i> debugging information in the VMS format (<i>n</i> defaults to 2)
-gxooff[n]	Generates level <i>n</i> debugging information in the XCOFF format (<i>n</i> defaults to 2)
-gxooff+	Generates level <i>n</i> debugging information in the XCOFF format, using extensions specific to GDB
-p	Generates code that dumps profiling information used by the <code>prof</code> program
-pg	Generates code that dumps profiling information used by the <code>gprof</code> program
-Q	Displays the name of each function compiled and how long each compiler phase takes
-time	Displays the CPU time consumed by each phase of the compilation sequence

CAUTION *The -a option for displaying the profiling information on basic blocks does not work in GCC 3.x. This is a known bug in the documentation, filed as PR 7251. If you try to use -a with GCC 3.1 or 3.2, you get the error message “ccl: unrecognized option ‘-a’” because both the -a and the -ax options, although documented, have been removed from GCC version 3.x.*

Quite a few options, eh? Fortunately, and as is often the case with GCC, you only have to concern yourself with a small subset of the available options and capabilities at any given time or for any given project. In fact, most of the time, you can accomplish a great deal just using -g or -ggdb. Before getting started, though, we will make short work of a few options that we will discuss elsewhere: -fprofile-arcs, -ftest-coverage, -p, and -pg. Although listed as some of GCC’s debugging options, they are best and most often used to profile code or to debug the compiler itself. In connection with their use in code profiling and code analysis, we discuss -fprofile-arcs, -ftest-coverage, -p, and -pg at length in Chapter 6.

The option *-dmod*, referred to as the *dump option* in this section, tells GCC to emit debugging dumps during compilation at the times specified by *mod*, which

can be one of almost every letter in the alphabet (see Table 3-10). These options are almost exclusively used for debugging the compiler itself, but you might find the debugging dumps informative or instructional if you want to learn GCC's deep voodoo. Each dump is left in a file, usually named by appending the compiler's pass number and a phrase indicating the type of dump to the source file's name (for example, myprog.c.14.bb). Table 3-11 lists the possible values for *mod*, briefly describes the corresponding compiler pass, and includes the name of the dump file, when applicable.

Table 3-11. Arguments for the Dump Option

Argument	Description	File
a	Produces all dumps except those produced by m, p, P, v, x, and y	N/A
A	Annotates the assembler output with miscellaneous debugging information	N/A
b	Dumps after computing branch probabilities pass	<i>file.14.bp</i>
B	Dumps after the block reordering pass	<i>file.29.bb</i>
c	Dumps after the instruction combination	<i>file.16.combine</i>
C	Dumps after the first if-conversion	<i>file.17.ce</i>
d	Dumps after delayed branch scheduling	<i>file.31.dbr</i>
D	Dumps all macro definitions after preprocessing	N/A
e	Dumps after performing SSA optimizations	<i>file.04.ssa</i> , <i>file.07.usa</i>
E	Dumps after the second if-conversion pass	<i>file.26.ce2</i>
f	Dumps after life analysis	<i>file.15.life</i>
F	Dumps after purging ADDRESSOF codes	<i>file.09.addressof</i>
g	Dumps after global register allocation	<i>file.21.greg</i>
G	Dumps after global common subexpression elimination (GCSE)	<i>file.10.gcse</i>
h	Dumps after finalizing of EH handling code	<i>file.02.eh</i>
I	Dumps after the sibling call optimization pass	<i>file.01.sibling</i>
j	Dumps after the first jump optimization pass	<i>file.03.jump</i>
k	Dumps after the register to stack conversion pass	<i>file.28.stack</i>
l	Dumps after local register allocation	<i>file.20.lreg</i>

Table 3-11. Arguments for the Dump Option (continued)

Argument	Description	File
L	Dumps after the loop optimization pass	<i>file.11.loop</i>
M	Dumps after the machine-dependent reorganization pass	<i>file.30.mach</i>
m	Prints statistics on memory usage at the end of the run	Standard error
n	Dumps after register renumbering	<i>file.25.rnreg</i>
N	Dumps after the register move pass	<i>file.18.regmove</i>
o	Dumps after the post-reload optimization pass	<i>file.22.postreload</i>
p	Annotates the assembler output with a comment indicating which pattern and alternative was used	N/A
P	Dumps the assembler output with the RTL as a comment before each instruction and enables -dp annotations	N/A
r	Dumps after RTL generation	<i>file.00.rtl</i>
R	Dumps after the second scheduling pass	<i>file.27.sched2</i>
s	Dumps after the first common subexpression elimination (CSE) and jump optimization pass	<i>file.08.cse</i>
S	Dumps after the first scheduling pass	<i>file.19.sched</i>
t	Dumps after the second CSE pass and post-CSE jump optimization	<i>file.12.cse2</i>
v	Dumps a representation of the control flow graph for each requested dump file (except <i>file.00.rtl</i>)	<i>file.pass.vcg</i>
w	Dumps after the second flow analysis pass	<i>file.23.flow2</i>
x	Generates RTL for a function instead of compiling it (used with -dr)	N/A
X	Dumps after the SSA dead-code elimination pass	<i>file.06.ssadce</i>
y	Dumps debugging information during parsing	Standard error
z	Dumps after the peephole pass	<i>file.24.peephole2</i>

CAUTION *The GCC info documentation lists the -dk option twice, once outputting the file file.28.stack and once outputting the file file.32.stack. Our testing indicates that only the file.28.stack is generated.*

Although the options listed in Table 3-11 are most often used for debugging the compiler itself, they can also be used as an analytic aid for any program. The v argument to -d, in particular, proves quite useful because its output can be used with a third-party program to generate a graphical display that corresponds to the dump option with which it was used. For example, if you compile the program myprog.c, using the option -dCv, you will wind up with the files myprog.c.17.ce and myprog.c.17.ce.vcg, in addition to the normal compiler output.

```
$ gcc -O -dCv myprog.c -o myprog
$ ls -l myprog*
-rw-r--r--  1 kwall    users          220 Oct  5 16:17 myprog.c
-rw-r--r--  1 kwall    users        4233 Oct 16 23:04 myprog.c.17.ce
-rw-r--r--  1 kwall    users        4875 Oct 16 23:04 myprog.c.17.ce.vcg
```

The C argument to -d dumps a flow control graph after the first if-conversion pass, which means that you need to enable optimization (see Chapter 5). The v argument creates a second dump file that contains the same information in a format that the Visualization of Compiler Graphs (VCG) tool can read and convert into a pretty graph. VGC is available for Unix and Windows systems. More information and downloadable versions in source and binary format are available from the VCG Web site at <http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html>. VCG is developed and maintained independently of GCC.

CHAPTER 4

Advanced GCC Usage

GCC's **DEFAULT BEHAVIOR** is usually what you want. Rather, GCC usually does The Right Thing and, at least in our experience, rarely violates the Principle of Least Surprise. As you have no doubt begun to appreciate by now, if you want GCC to do something, chances are pretty good that it has a command-line option that will make it do so. Nevertheless, what should you do if you dislike GCC's default behavior, get tired of typing the same command-line option, and do not know enough to modify GCC's code? Naturally, you customize GCC using environment variables or, as you will learn in the next section, specification (spec) strings.

Customizing GCC Using Environment Variables

This section describes environment variables that affect how GCC operates. Some work by defining directories or prefixes to use when searching for various kinds of files, while others control other features of GCC's runtime environment. In all cases, options specified on the command line always override options specified via environment values. Similarly, GCC options defined in the environment always override options compiled into GCC, that is, those constituting GCC's default configuration.

The first two environment variables, `COMPILER_PATH` and `GCC_EXEC_PREFIX`, control modification of GCC's search path. `COMPILER_PATH` contains a colon-delimited list of directories, comparable to the familiar Unix `PATH` environment variable. GCC uses the directories specified in `COMPILER_PATH`, if any, to find subprograms, such as `cc1`, `collect2`, and `cpp0`, in its default search path or `GCC_EXEC_PREFIX`.

`GCC_EXEC_PREFIX`, if set, specifies the prefix to use in the names of the subprograms executed by the compiler. This variable does not necessarily specify a directory name because GCC does not add a slash (/) to the front of the prefix when it combines `GCC_EXEC_PREFIX` with the name of a subprogram. So, if `GCC_EXEC_PREFIX` is `/usr/lib/gcc-lib/i486-slackware-linux/3.2.3/`, GCC attempts to invoke the subprogram `cc1` as `/usr/lib/gcc-lib/i486-slackware-linux/3.2.3/cc1`. On the other hand, if `GCC_EXEC_PREFIX` is `custom-`, GCC attempts to invoke the subprogram `cc1` as `custom-cc1`.

NOTE *The GCC_EXEC_PREFIX environment variable does not seem to work under Cygwin. We are unclear if this is a function of Cygwin's emulation environment or of the version of GCC used.*

If `GCC_EXEC_PREFIX` is unset, GCC uses a heuristic to derive the proper prefix based on the path by which you invoked GCC. The default value is `PREFIX/lib/gcc-lib/`, where `PREFIX` is the value of `--prefix` you specified when you configured the GCC script. If you specify `-B` on the command line, as noted earlier, the command-line specification overrides `GCC_EXEC_PREFIX`.

NOTE *GCC_EXEC_PREFIX also enables GCC to find object files, such as `crt0.o` and `crtbegin.o`, used during the link phase.*

`C_INCLUDE_PATH`, used by the preprocessor, specifies a list of directories that will be searched when preprocessing a C file. This option is equivalent to the `-isystem` option, except that paths specified in `C_INCLUDE_PATH` are searched after directories specified with `-isystem`.

Another environment variable used by the preprocessor, `CPATH`, specifies a list of directories to search for included files during preprocessing. This option is equivalent to the `-I` option, except that directories specified in `CPATH` are searched after directories specified using `-I`. `CPATH` is always used, if defined, and used before `C_INCLUDE_PATH`, `CPLUS_INCLUDE_PATH`, and `OBJC_INCLUDE_PATH`, if they are defined.

`CPLUS_INCLUDE_PATH` functions comparably to `C_INCLUDE_PATH`, but applies to C++ compilations, rather than C compilations. `CPLUS_INCLUDE_PATH` specifies a list of directories that will be searched during preprocessing of a C++ file. This option is equivalent to the `-isystem` option, except that paths specified in `CPLUS_INCLUDE_PATH` are searched after directories specified with `-isystem`.

Finally, `OBJC_INCLUDE_PATH` contains a list of directories to search when preprocessing Objective C source files. This option is equivalent to the `-isystem` option, except that paths specified in `OBJC_INCLUDE_PATH` are searched after directories specified with `-isystem`.

As discussed in Chapter 3, GCC's `-M` option can be used to tell GCC to generate dependency output (for nonsystem header files—system header files are ignored) for the Make utility. The environment variable `DEPENDENCIES_OUTPUT` customizes this behavior, if set. If `DEPENDENCIES_OUTPUT` is a filename, the generated Make rules are written to that file, and GCC guesses the target name based on the name of the source file. If `DEPENDENCIES_OUTPUT` is specified as *file target*, the rules are written to *file* using *target* as the target name. Using `DEPENDENCIES_OUTPUT` is equivalent to combining the options `-MM` and `-MF`, and optionally, `-MT`.

`LIBRARY_PATH` is a colon-delimited list in which GCC searches for special linker files. `LIBRARY_PATH` is searched after `GCC_EXEC_PREFIX`. If you use GCC for linking, that is, if you do not invoke the linker directly, GCC uses the directories specified in `LIBRARY_PATH`, if any, to find libraries specified with `-L` and `-l` (in that order).

GCC tries to play nice with language and locale information, so it is sensitive to the presence of the `LANG`, `LC_ALL`, `LC_CTYPE`, and `LC_MESSAGES` environment variables. GCC uses these variables to select the character set to use when the C and C++ compilers parse character literals, string literals, and comments. If configured for multibyte characters, GCC understands JIS, SJIS, and EUCJP characters if `LANG` has the value `C-JIS`, `C-SJIS`, or `C-EUCJP`, respectively. Otherwise, GCC uses the locale functions `mblen()` and `mbtowc()` to process multibyte characters.

Whereas `LANG` determines how GCC processes character sets, `LC_ALL`, `LC_CTYPE`, `LC_MESSAGES` control how GCC uses locale information to interpret and process other elements of locale customization. GCC uses `LC_CTYPE`, for example, to determine the character boundaries in a string—some multibyte character encodings contain quote and escape characters that are ordinarily interpreted to denote an escape character or the end of a string. `LC_MESSAGES` informs GCC of the language to use when emitting diagnostic messages.

If `LC_CTYPE` or `LC_MESSAGES` are not set, they default to the value of `LANG`. `LC_ALL`, if set, overrides the values of `LC_CTYPE` and `LC_MESSAGES`. If none of these variables are set, GCC defaults to the traditional behavior of a C compiler in a U.S. English setting.

Finally, `TMPDIR` defines the directory to use for temporary files, which is normally `/tmp` on Unix systems. GCC uses temporary files to hold the output of one compilation stage, which will be used as input to the next stage. For example, the output of the preprocessor is the input to the compiler proper. One way to speed up GCC is to define `TMPDIR` to point to a RAM disk, which bypasses the overhead and limitations of physical disk reads and writes during compilation. If `TMPDIR` is not set, GCC uses the default temporary directory, which varies according to system.

Customizing GCC with Spec Files and Spec Strings

One of the points we have tried to emphasize throughout this book is that `gcc` is a driver program that does its job by invoking a sequence of other programs to do the heavy lifting of preprocessing, compiling, assembling, and linking. GCC's command-line parameters and filename arguments help it decide which helper and back-end programs to invoke and how to invoke them. Spec strings control the invocation of the helper programs.

A *spec string* is a list of command-line options passed to a program. Spec strings can also contain variable text substituted into a command line or text that might or might not be inserted into a command line, based on a conditional

statement (in the spec string). Using these constructs, it is possible to generate quite complex command lines. A *spec file* is a plain text file that contains spec strings. Spec files contain multiple directives, separated by blank lines. The type of directive is determined by the first nonwhitespace character on the line, as we will explain very shortly. In most cases there is one spec string for each program that GCC can invoke, but a few programs have multiple spec strings to control their behavior.

TIP As you work through the examples in this section, you might find it easier to see what is happening if you invoke the compiler with `-###`. This option makes it easier to distinguish the output.

To override the spec strings built into GCC, use the `-specs=file` option to specify an alternative or additional spec file named *file*. The gcc driver reads the specified file after reading the defaults from the standard specs file. If you specify multiple spec files, gcc processes them in order from left to right. Table 4-1 lists the possible spec file directives.

Table 4-1. Spec File Directives

Directive	Description
<code>%command</code>	Executes the spec file command specified by <i>command</i> (see Table 4-2)
<code>*spec_name</code>	Creates, deletes, or overrides the spec string specified by <i>spec_name</i>
<code>suffix</code>	Creates a new suffix rule specified by <i>suffix</i>

Before you start to panic at the prospect of facing *yet another* command language to learn, fear not: GCC's spec file command language has only three very simple and easy-to-remember commands: `include`, `include_noerr`, and `rename`. Table 4-2 describes what these spec file commands do.

Table 4-2. Spec File Commands

Command	Description
<code>%include file</code>	Searches for and inserts the contents of <i>file</i> at the current point in the spec file
<code>%include_noerr file</code>	Works like <code>%include</code> , without generating an error message if <i>file</i> is not found
<code>%rename old new</code>	Renames the spec string <i>old</i> to <i>new</i>

`*spec_name` creates, deletes, or redefines spec string *spec_name*. The spec string definition or redefinition continues to the next directive or blank line. To delete a spec string, add a blank line or another directive immediately following `*spec_name`. To append text to an existing spec, use + as the first character of the text defining the spec.

`suffix` creates, deletes, or modifies a spec pair, which is comparable to a suffix rule for the Make utility. As with the `*` directive, the text following `suffix` up to the next directive or a blank line defines the spec string for `suffix`. When the compiler encounters an input file with the named suffix, it uses the spec string to determine how to compile that file. For example, the following suffix directive creates a spec string that says any input file ending with the characters `.d` should be passed to the program `dcc`, with an argument of `-j` and the results of the substitution for `%i` (we discuss substitution shortly).

```
.d:  
dcc -j %i
```

The spec string `#name` following a suffix specification instructs `gcc` to emit an error message that says “*name* compiler not installed on this system.” The actual text of the error message appears to differ depending on the version of `GCC` you are using and the platform (operating system) on which you are working. For example, consider the following suffix spec string:

```
.d:  
#dcc
```

If the `gcc` binary encounters a file whose name ends in `.d`, the `gcc` driver will emit the error message “`dcc` compiler not installed on this system.” Suffix strings exist primarily to simplify extending the `gcc` driver program to handle new backend compilers and new file types.

Now that you know how to create and modify spec strings, take a look at Table 4-3, which lists GCC's built-in spec strings. Use this table to learn how GCC's designers have configured GCC and, more importantly, to identify spec strings you should *not* idly redefine unless you want to break GCC in really interesting ways.

Table 4-3. Built-in Spec Strings

Spec String	Description
asm	Specifies the options passed to the assembler
asm_final	Specifies the options passed to the assembler postprocessor
cc1	Specifies the options passed to the C compiler
cc1plus	Specifies the options passed to the C++ compiler
cpp	Specifies the options passed to the C preprocessor
endfile	Specifies the object files to link at the end of the link phase
lib	Specifies the libraries to pass to the linker
libgcc	Specifies the GCC support library (libgcc) to pass to the linker
link	Specifies the options passed to the linker
linker	Specifies the name of the linker
defines	Specifies the #defines passed to the C preprocessor
signed_char	Specifies the #defines passed to the C preprocessor indicating if a char is signed by default
startfile	Specifies the object files to link at the beginning of the link phase

Here is a small example of a spec string that changes the linker definition from, say, ld, to my_new_ld:

```
%rename linker old_linker
*linker:
my_new_%(old_linker)
```

This example renames the spec string named linker to old_linker. The blank line ends the %rename command. The next line, *linker, redefines the linker spec string, replacing it with my_new_%(old_linker). The syntax %(old_linker), known as a *substitution*, appends the previous definition of linker to the new definition,

much as one would use the Bourne shell command `PATH=/usr/local/gcc/bin:$PATH` to insert `/usr/local/gcc/bin` at the beginning of the existing directory search path.

Table 4-4 shows many of GCC's predefined substitution specs.

Table 4-4. Predefined Substitution Specs

Spec String	Description
<code>%%</code>	Substitutes a % into the program name or argument
<code>%b</code>	Substitutes the currently processing input file's basename (as the basename shell command might generate), without the suffix
<code>%B</code>	Like <code>%b</code> , but includes the file suffix
<code>%d</code>	Denotes the argument following <code>%d</code> as a temporary filename, which results in the file's automatic deletion upon gcc's normal termination
<code>%estr</code>	Designates <i>str</i> as a newline-terminated error message to display
<code>%gsuffix</code>	Substitutes a filename with a suffix matching <i>suffix</i> (chosen once per compilation), marking the argument for automatic deletion (as with <code>%d</code>)
<code>%i</code>	Substitutes the name of the currently processing input file
<code>%jsuffix</code>	Substitutes the name of the host's null device (such as <code>/dev/null</code> on Unix systems), if one exists, if it is writable, and if <code>-save-temp</code> s has not been specified
<code>%o</code>	Substitutes the names of all the output files, delimiting each name with spaces
<code>%O</code>	Substitutes the suffix for object files
<code>%(name)</code>	Inserts the contents of spec string <i>name</i>
<code>%usuffix</code>	Like <code>%g</code> , but generates a new temporary filename even if <code>%usuffix</code> has already been specified
<code>%v1</code>	Substitutes GCC's major version number
<code>%v2</code>	Substitutes GCC's minor version number
<code>%v3</code>	Substitutes GCC's patch-level number
<code>%w</code>	Defines the argument following <code>%w</code> as the current compilation's output file, which is later used by the <code>%o</code> spec (see the entry for <code>%o</code>)

On systems that do not have a null device or some type of bit bucket, `%jsuffix` substitutes the name of a temporary file, which is treated as if specified with the `%u` spec. Such a temporary file should not be used by other spec strings because

it is intended as a way to get rid of junk data automatically, not as a means for two compiler processes to communicate.

Table 4-5 lists the available spec processing instructions.

Table 4-5. Spec Processing Instructions

Spec String	Description
%1	Processes the cc1 spec, which selects the options to pass to the C compiler (cc1).
%2	Processes the cc1plus spec, which builds the option list to pass to the C++ compiler (cc1plus).
%a	Processes the asm spec, which selects the switches passed to the assembler.
%c	Processes the signed_char spec, which enables cpp to decide if a char is signed or unsigned.
%C	Processes the cpp spec, which builds the argument list to pass to cpp, the C preprocessor.
%E	Processes the endfile spec, which determines the final libraries to pass to the linker.
%G	Processes the libgcc spec, which determines the correct GCC support library against which to link.
%l	Processes the link spec, which constructs the command line that invokes the linker.
%L	Processes the lib spec, which selects the library names to pass to the linker.
%S	Processes the startfile spec, which determines the start up files (such as the C runtime library) that must be passed to the linker first. For C, this might be a file named crt0.o.

GCC's C and Extensions

As we remarked at the beginning of this section, GCC, or, rather, GNU C, provides language features not available in ANSI/ISO standard C. Following is a nearly complete summary list of these features:

- *Local labels*: Write expressions containing labels with local scope
- *Labels as values*: Obtain pointers to labels and treat labels as computed gotos

- *Nested functions*: Define functions within functions
- *Constructing calls*: Dispatch a call to another function
- *Typeof*: Determine an expression's type at runtime using `typeof`
- *Zero- and variable-length arrays*: Declare zero-length arrays
- *Variable-length arrays*: Declare arrays whose length is computed at runtime
- *Variadic macros*: Define macros with a variable number of arguments
- *Subscripting*: Subscript any array, even if not an lvalue
- *Pointer arithmetic*: Perform arithmetic on void pointers and on function pointers
- *Initializers*: Assign initializers using variable values
- *Designated initializers*: Label elements of initializers
- *Case ranges*: Represent syntactic sugar in switch statements
- *Mixed declarations*: Mix declarations and code
- *Function attributes*: Declare that a function has no side effects or never returns
- *Variable attributes*: Specify attributes of variables
- *Type attributes*: Specify attributes of types
- *Inline*: Define inline functions (as fast as macros)
- *Function names*: Display the name of the current function
- *Return addresses*: Obtain a function's return address or frame address
- *Pragmas*: Use GCC-specific pragmas
- *Unnamed fields*: Embed unnamed `struct` and `union` fields within named `structs` and `unions`

Locally Declared Labels

Each statement expression is a scope in which *local labels* can be declared. A local label is simply an identifier; you can jump to it with an ordinary `goto` statement, but only from within the statement expression it belongs to. A local label declaration looks like this:

```
__label__ alabel;
```

or

```
__label__ alabel, blabel, ...;
```

The first statement declares a local label named *alabel*, and the second one declares two labels named *alabel* and *blabel*. Local label declarations must appear at the beginning of the statement expression, immediately following the opening brace (`{`) and before any ordinary declarations.

A label declaration defines a label *name*, but does not define the label itself. You do this using *name*: within the statement's expression to define the contents of the label named *name*. For example, consider the following code:

```
int main(void)
{
    label__ something;
    int foo;

    foo=0;
    goto something;
    {
        __label__ something;
        goto something;
        something:
            foo++;
    }
    something:
        return foo;
}
```

This code declares two labels named *something*. The label in the outer block must be distinguished from the label in the local block. *Local label declaration* means that the label must still be unique with respect to the largest enclosing

block. So if you put a label in a macro, you need to give it a mangled name to avoid clashing with a name the user has defined. The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a goto can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function.

Local labels avoid this problem. For example:

```
#define SEARCH(array, target) \
({ \
    __label__ found; \
    typeof (target) _SEARCH_target = (target); \
    typeof (*(array)) *_SEARCH_array = (array); \
    int i, j; \
    int value; \
    for (i = 0; i < max; i++) \
        for (j = 0; j < max; j++) \
            if (_SEARCH_array[i][j] == _SEARCH_target) \
                { value = i; goto found; } \
    value = -1; \
    found: \
    value; \
})
```

Labels As Values

You can get the address of a label defined in a current function using the operator `&&` (a unary operator, for you language lawyers). The value has type `void *`.

The label address is a constant value—it can be used wherever a constant of that type is valid. For example:

```
void *ptr;
ptr = &&foo;
```

To use this `ptr`, you need to be able to jump to it. With a normal label, you might say `goto ptr;`. However, with a label used as a value, you use a *computed goto statement*:

```
goto *ptr;
```

Any expression that evaluates to the type `void *` is allowed. One way to use these computed gotos is to initialize static arrays that will serve as a jump table:

```
static void *jump[] = { &&f, &&g, &&h };
```

In this case, `f`, `g`, and `h` are labels to which code jumps at runtime. To use one, you index into the array and select the label. The code to do this might resemble the following:

```
goto *array[i];
```

You cannot use computed gotos to jump to code in a different function, primarily because attempting to do so subverts (or, perhaps, abuses) the local label feature described in the previous section. Store the label addresses in automatic variables and never pass such an address as an argument to another function.

Nested Functions

A *nested function* is a function defined inside another function. As you might anticipate, a nested function's name is local to the block in which it is defined. To illustrate, consider a nested function named `swap()` that is called twice:

```
f(int i, int j)
{
    void swap(int *a, int *b)
    {
        int tmp = *a;
        *a = *b;
        *b = tmp;
    }
    /* more code here */
    swap(&i, &j);
}
```

The nested `swap()` can access all the variables of the surrounding function that are visible at the point when it is defined (known as *lexical scoping*).

NOTE GNU C++ does not support nested functions.

Nested function definitions are permitted in the same places in which variable definitions are allowed within functions: in any block and before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
inc(int *array, int size)
{
    void swap(int *a, int *b)
    {
        int tmp = *a;
        *a = *b;
        *b = tmp;
    }
    save(swap, size);
}
```

In this snippet, the address of `swap()` is passed as an argument to the function `save()`. This trick only works if the surrounding function, `inc()`, does not exit.

If you try to call a nested function through its address after the surrounding function exits, well, as GCC authors put it, “all hell will break loose.” The most common disaster is that variables that have gone out of scope will no longer exist and you will end up referencing garbage or overwriting something else, either of which can be a difficult bug to track down. You might get lucky, but the path of true wisdom is not to take the risk.

NOTE *GCC uses a technique called trampolines to implement taking addresses of nested functions. You can read more about this technique at <http://people.debian.org/~aaronl/Usenix88-lexic.pdf>.*

Nested functions can jump to a label inherited from a containing function, that is, a local label, if the label was explicitly declared in the containing function, as described in the section “Locally Declared Labels.”

A nested function always has internal linkage, so declaring a nested function with `extern` is an error.

Constructing Function Calls

Using GCC’s built-in functions, you can perform some interesting tricks, such as recording the arguments a function received and calling another function with

the same arguments, but without knowing in advance how many arguments were passed or the types of the arguments.

```
void * __builtin_apply_args();
```

`__builtin_apply_args()` returns a pointer to data describing how to perform a call with the same arguments as were passed to the current function.

```
void * __builtin_apply(void (*function)(), void *args, size_t size);
```

`__builtin_apply()` calls *function* with a copy of the parameters described by *args* and *size*. *args* should be the value returned by `__builtin_apply_args()`. The argument *size* specifies the size of the stack argument data in bytes. As you can probably imagine, computing the proper value for *size* might prove a nontrivial undertaking.

Similarly, and again without advance knowledge of a function's return type, you can record that function's return value and return it yourself. Naturally, though, the calling function must be prepared to receive that data type. The built-in function that accomplishes this feat is `__builtin_return()`.

```
void __builtin_return(void *retval);
```

`__builtin_return()` returns the value described by *retval* from the containing function. *retval* must have been a value returned by `__builtin_apply()`.

Referring to a Type with `typeof`

Another way to refer to or obtain the type of an expression is to use the `typeof` keyword, which shares its syntax with the ISO C keyword `sizeof` but uses the semantics of a type name defined with `typedef`. Syntactically, `typeof`'s usage is what you would expect: you can use `typeof` with an expression or with a type. With an expression, the usage might resemble the following:

```
typeof (array[0](1));
```

As you can see in this example, `array` is an array of pointers to functions. The resulting type will be the values of the functions. `typeof`'s usage with a type name is more straightforward.

```
typeof (char *);
```

Obviously, the resulting type will be pointers to char.

TIP If you are writing a header file that must work when included in ISO C programs, write `_typeof_` instead of `typeof`.

Semantically, `typeof` can be used anywhere a `typedefed` name would be, which includes declarations, casts, `sizeof` statements, or even, if you want to be perverse, within another `typeof` statement. A typical use of `typeof` is to create type-safe expressions, such as a `max` macro that can safely operate on any arithmetic type and evaluates each of its arguments exactly once:

```
#define max(a,b) \
({ typeof (a) _a = (a); \
  typeof (b) _b = (b); \
  _a > _b ? _a : _b; })
```

The names `_a` and `_b` start with underscores to avoid the local variables conflicting with variables having the same name at the scope in which the macro is called. Additional `typeof` uses might include the following, or variations thereof:

- Declares `y` to be of the type to which `x` points:

```
typeof (*x) y;
```

- Declares `y` to be an array of the values to which `x` points:

```
typeof (*x) y[4];
```

- Declares `y` as an array of pointers to characters:

```
typeof (typeof (char *)[4]) y;
```

Clearly, this code is more elaborate than the following standard C declaration, to which it is equivalent:

```
char *y[4];
```

Zero-Length Arrays

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure that is really a header for a variable-length object. Consider:

```

struct data {
    int i;
    int j;
}

struct entry {
    int size;
    struct data[0];
};

int e_sz;
struct entry *e = (struct entry *) malloc(sizeof (struct entry) + e_sz);
e->size = e_sz;

```

In ISO C89, you must define `data[1]`, giving `data` a length of 1. This requirement forces you to waste space (a trivial concern in this example) or, more likely, complicate the `malloc()` call. ISO C99 allows you to use *flexible array members*, which have slightly different characteristics:

- Flexible array members would be defined as `contents[]`, not `contents[0]`.
- C99 considers flexible array members to have incomplete type specifications. You cannot use `sizeof` with incomplete types. With GNU C's zero-length arrays, however, `sizeof` evaluates to 0. It is up to you to decide if that is a feature or not.
- Most significantly, flexible array members may only appear as the last member of nonempty structs.

To support flexible array members, GCC extends them to permit static initialization of flexible array members. This is equivalent to defining two structures such that the second structure contains the first one, followed by an array of sufficient size to contain the data. If that seems confusing, consider the two structures `s1` and `s2` defined as follows:

```

struct f1 {
    int x;
    int y[];
} f1 = { 1, { 2, 3, 4 } };

struct f2 {
    struct f1 f1;
    int data[3];
} f2 = { { 1 }, { 2, 3, 4 } };

```

```

struct s1 {
    int i;
    int j[];
} s1 = {1, {2, 3, 4} };

struct s2 {
    struct s1 s1;
    int array[3];
} s2 = { {1}, {2, 3, 4} };

```

In effect, `s1` is defined as if it were declared like `s2`.

The convenience of this extension is that `f1` has the desired type, eliminating the need to consistently refer to the awkward construction `f2.f1`. This extension is also symmetric with declarations of normal static arrays, in that an array of unknown size is also written with `[]`. For example, consider the following code:

```

struct foo {
    int x;
    int y[];
};

struct bar {
    struct foo z;
};

struct foo a = {1, {2, 3, 4 } };           /* valid */
struct bar b = { { 1, {2, 3, 4 } } };      /* invalid */
struct bar c = { { 1, {} } };                /* valid */
struct foo d[1] = { { 1 {2, 3, 4} } };     /* invalid */

```

TIP *This code will not compile unless you are using GCC 3.2 or newer.*

Arrays of Variable Length

ISO C99 allows variable-length automatic arrays. Not wanting to limit this very handy feature to C99, GCC accepts variable-length arrays in C89 mode *and* in C++ (using `-std=gcc89`). Nevertheless, what ISO C99 giveth, GCC taketh away: GCC's implementation of variable-length arrays does not yet conform in all details to the ISO C99 standard. Variable-length arrays are declared like other automatic arrays, but the length is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace level is exited. For example:

```

FILE *myfopen(char *s1, char *s2, char *mode)
{
    char str[strlen (s1) + strlen (s2) + 1];
    strcpy(str, s1);
    strcat(str, s2);
    return fopen(str, mode);
}

```

Jumping or breaking out of the scope of the array name deallocates the storage. You can use the function `alloca()` to get an effect much like variable-length arrays. The function `alloca()` is available in many, but not all, C implementations. On the other hand, variable-length arrays are more elegant. You might find variable-length arrays more straightforward to use because the syntax is more natural.

Should you use an `alloca()`-declared array or a variable-length array? Before you decide, consider the differences between these two methods. Space allocated with `alloca()` exists until the containing function returns, whereas space for a variable-length array is deallocated as soon as the array name's scope ends. If you use both variable-length arrays and `alloca()` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca()`. You can also use variable-length arrays as arguments to functions.

```

void f(int len, char data[len])
{
    /* function body here */
}

```

Here, you can see the function `foo()` accepts the integer parameter `len`, which is also used to specify the size of the array of data. The array's length is determined when the storage is allocated and is remembered for the scope of the array, in case the length is accessed using `sizeof`.

If you want to pass the array first and the length second, you can use a forward declaration in the parameter list (which is, by the way, another GNU extension). A forward declaration in the parameter list looks like the following:

```

void f(int len; char data[len], int len)
{
    /* function body here */
}

```

The declaration `int len` before the semicolon, known as a *parameter forward declaration*, makes the name `len` known when the declaration of `data` is parsed.

You can make multiple parameter forward declarations in the parameter list, separated by commas or semicolons, but the last one *must* end with a semi-colon. Following the final semicolon, you must declare the actual parameters. Each forward declaration must match a real declaration in parameter name and data type. ISO C99 does not support parameter forward declarations.

Macros with a Variable Number of Arguments

C99 allows declaring macros with a variable number of arguments, just as functions can accept a variable number of arguments. The syntax for defining the macro is similar to that of a function. For example:

```
#define debug(format, ...) fprintf(stderr, format, __VA_ARGS__)
```

The ... specifies the variable argument. When you invoke such a macro, ... represents zero or more tokens until the closing parenthesis that ends the invocation. This set of tokens replaces the identifier __VA_ARGS__ in the macro body wherever it appears. See the C preprocess manual (`info cpp`) for more information about how GCC processes variadic arguments.

Subscripting Non-Lvalue Arrays

In ISO C99, arrays that are not lvalues still decay to pointers, and may be subscripted, although they may not be modified or used after the next sequence point and the unary & operator may not be applied to them. As an extension, GCC allows such arrays to be subscripted in C89 mode; otherwise, they do not decay to pointers outside C99 mode. For example, the following code is valid in GNU C but not in C89:

```
#include <stdio.h>

struct foo {
    int a[4];
};

struct foo f = { { 10, 11, 12, 13 } };

bar(int index)
{
    return f().a[index];
}
```

```

int main(int argc, char *argv[]) {
    printf("%d\n", bar(2));
    return 0;
}

```

Arithmetic on Void and Function Pointers

GNU C supports adding and subtracting pointer values on pointers to void and on pointers to functions. GCC implements this feature by assuming that the size of a void or of a function is 1. As a consequence, `sizeof` also works on void and function types, returning 1. The option `-Wpointer-arith` enables warnings if these extensions are used.

Nonconstant Initializers

GNU C does not require the elements of an aggregate initializer for an automatic variable to be constant expressions. This is the same behavior permitted by both standard C++ and ISO C99. An initializer with elements that vary at runtime might resemble the following code:

```

float f(float f, float g)
{
    float beats [2] = { f-g, f+g };
    /* function body here */
    return beats[1];
}

```

Designated Initializers

Standard C89 requires initializer elements to appear in the same order as the elements in the array or structure being initialized. C99 relaxes this restriction, permitting you to specify the initializer elements in any order by specifying the array indices or structure field names to which the initializers apply. These are known as *designated initializers*. GNU C allows this as an extension in C89 mode, but not, for you C++ programmers, in GNU C++.

To specify an array index, for example, write `[index]=` before the element value as shown here:

```
int a[6] = {[4] = 29, [2] = 15};
```

This is equivalent to

```
int a[6] = {0, 0, 15, 0, 29, 0};
```

The index values must be a constant expression, even if the array being initialized is automatic.

To initialize a range of elements with the same value, use the syntax [*first* ... *last*]= *value*. This is a GNU extension. For example:

```
int widths[] = {[0 ... 9] = 1, [10 ... 99] = 2, [100] = 3};
```

If *value* has side effects, the side effects happen only once, rather than for each element initialized.

In a structure initializer, specify the name of a field to initialize with *.member*=. Given a structure triplet defined as

```
struct triplet {
    int x, y, z;
};
```

the following code snippet illustrates the proper initialization method:

```
struct triplet p = {
    .y = y_val,
    .x = x_val,
    .z = z_val
};
```

This initialization is equivalent to

```
struct triplet p = {
    x_val,
    y_val,
    z_val
};
```

The [*index*] or *.member* is referred to as a *designator*.

You can use a designator when initializing a union to identify the union element to initialize. For example, with a union defined as follows:

```
union foo {
    int i;
    double d;
};
```

the following statement converts 4 to a double to store it in the union using the second element:

```
union foo f = {
    .d = 4
};
```

In contrast, casting 4 to type `union foo` would store it in the union as the integer `i`, because it is an integer.

You can combine designated initializers with ordinary C initialization of successive elements. In this case, initializers lacking designators apply to the next consecutive element of the array or structure. For example, the line

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an enum type. For example:

```
int whitespace[256] = {
    [' '] = 1, ['\t'] = 1, ['\v'] = 1,
    ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

You can also write a series of `.member` and `[index]` designators before = to specify a nested subobject to initialize; the list is taken relative to the subobject corresponding to the closest surrounding brace pair. For example, with the preceding struct triplet declaration, you would use the following:

```
struct triplet triplet_array[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };
```

NOTE If the same field is initialized multiple times, it will have value from the last initialization. If multiple overridden initializations have side effects, the standard does not specify whether the side effect happens or not. GCC currently discards additional side effects and issues a warning.

Case Ranges

You can specify a range of consecutive values in a single case label like this:

```
case m ... n:
```

Spaces around the ... are required. This has the same effect as the proper number of individual case labels, one for each integer value from *m* to *n*, inclusive. This feature is especially useful for ranges of ASCII character codes and numeric values.

```
case 'A' ... 'Z':
```

Mixed Declarations and Code

ISO C99 and ISO C++ allow declarations and code to be freely mixed within compound statements. The GCC extension to this feature is allow mixed declarations and code in C89 mode. For example, you can write

```
int i;
/* other declarations here */
i++;
int j = i + 2;
```

Each identifier is visible from the point at which it is declared until the end of the enclosing block.

Declaring Function Attributes

GNU C enables you to tell the compiler about certain features or behavior of the functions in your program. This is done using keywords that declare *function attributes*. Function attributes permit the compiler to optimize function calls and check your code more carefully. The keyword `_attribute_` allows you to specify function attributes. `_attribute_` must be followed by an attribute specification inside double parentheses. The following attributes are currently defined for functions on all targets:

- alias
- always_inline
- const
- constructor
- deprecated
- destructor
- format
- format_arg
- malloc
- no_instrument_function
- noinline
- noreturn
- pure
- section
- unused
- used
- weak

Several other attributes are defined for functions on particular target systems. Other attributes are supported for variable declarations and for types.

TIP You can also specify attributes by embedding each keyword between `_`. This alternative syntax allows you to use function attributes in header files without being concerned about a possible macro of the same name. For example, you can write `_noreturn_` instead of `_attribute ((noreturn))`.

Generally, functions are not inlined unless optimization is specified. For functions declared `inline`, the `always_inline` attribute inlines the associated function even in the absence of optimization. Conversely, to prevent a function ever to be considered for inlining, use the function attribute `noinline`.

Many functions do not examine or modify any values except their arguments or have any effect other than returning a value. Such functions can be given the `const` attribute. A function that has pointer arguments and examines the data pointed to must not be declared `const`. Likewise, a function that calls a non-`const` function cannot be `const`. Naturally, it does not make sense for a `const` function to return `void`.

The `deprecated` attribute results in a warning if the associated function is used anywhere in the source file. This attribute is useful to identify functions you expect to remove in a future version of a program. The warning also includes the location of the declaration of the deprecated function, enabling users to find further information about why the function is deprecated, or what they should do instead. The warning only occurs when the function's return value is used. For example:

```
int old_f() __attribute__ ((deprecated));
int old_f();
int (*f_ptr)() = old_f;
```

The third line generates a warning, but the second line does not.

TIP You can also use the `deprecated` attribute with variables and `typedefs`.

The `malloc` attribute tells the compiler that a function should be treated as if it were the `malloc()` function. The purpose for this attribute is that the compiler assumes that calls to `malloc()` result in a pointer that cannot alias anything. This will often improve optimization specifically during alias analysis (see Chapter 5 for discussion of GCC alias analysis during optimization).

A few standard library functions, such as `abort()` and `exit()`, cannot return (they never return to the calling function). GCC automatically knows this about standard library functions and its own built-in functions. If your own code defines functions that never return, you can declare them using the `noreturn` attribute to tell the compiler this fact. For example:

```
void bye(int error) __attribute__ ((noreturn));
void bye(int error)
{
    /* error handling here*/
    exit(1);
}
```

The `noreturn` keyword tells the compiler to assume that `bye()` cannot return. The compiler can then optimize without needing to consider what might happen if `bye()` does return, which can result in slightly better code. Declaring functions `noreturn` also helps avoid spurious warnings about uninitialized variables. However, you should not assume that registers saved by the calling function are restored before calling the `noreturn` function. If a function does not return and is given the `noreturn` attribute, it should not have a return type other than `void`.

Many functions have no effects to return a value. Similarly, such functions' return values often depend only on the function parameters and/or global variables. As you will learn in Chapter 5, such functions can easily be optimized during common subexpression elimination and loop optimization, just as arithmetic operators would be. Such functions should be declared with the `pure` attribute.

For example, the following function declaration asserts that the `cube()` function is safe to call optimize using common subexpression elimination:

```
int cube(int i) __attribute__ ((pure));
```

Functions that might benefit from declaration as pure functions include functions that resemble `strlen()` and `memcmp()`. Functions you might *not* want to declare using the `pure` attribute include functions with infinite loops and those that depend on volatile memory or other system resources. The issue with such functions is that they depend on values that might change between two consecutive calls, such as `feof()` in a multithreading environment.

The `function attribute unused` means that it might not be used and that this is acceptable. Accordingly, GCC will omit producing a warning for this function. Likewise, the `used` function attribute declares that code must be emitted for the function even if it appears that the function is never referenced. This is useful, for example, when the function is referenced only in inline assembly.

NOTE *GNU C++ does not currently support the unused attribute because definitions without parameters are valid in C++.*

Before the language lawyers among you start complaining that ISO C's `pragma` feature should be used instead of `__attribute__`, consider the following points the GCC developers make in the GCC Texinfo help file:

At the time __attribute__ was designed, there were two reasons for not [using #pragma]:

1. *It is impossible to generate #pragma commands from a macro.*
2. *There is no telling what the same #pragma might mean in another compiler.*

These two reasons applied to almost any application that might have been proposed for #pragma. It was basically a mistake to use #pragma for anything.

The first point is somewhat less relevant now, because ISO C99 standard includes `_Pragma`, which allows pragmas to be generated from macros. GCC-specific pragmas (`#pragma GCC`), moreover, now have their own namespace. So, why does `__attribute__` persist? Again, GCC developers explain that “it has been found convenient to use `__attribute__` to achieve a natural attachment of attributes to their corresponding declarations, whereas `#pragma GCC` is of use for constructs that do not naturally form part of the grammar.”

Specifying Variable Attributes

You can also apply `__attribute__` to variables. The syntax is the same as for function attributes. GCC supports 10 variable attributes:

- `aligned`
- `deprecated`
- `mode`
- `nocommon`
- `packed`
- `section`
- `transparent_union`
- `unused`
- `vector_size`
- `weak`

To specify multiple attributes, separate them with commas within the double parentheses: for example, `_attribute_ ((aligned (16),packed))`. GCC defines other attributes for variables on particular target systems. Other front ends might define more or alternative attributes. For details, consult the GCC online help (`info gcc`).

The `aligned (n)` attribute specifies a minimum alignment of n bytes for a variable or structure field. For example, the following declaration tells the compiler to allocate a global variable `j` aligned on a 16-byte boundary:

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

You can also specify the alignment of structure fields. For example, you can create a pair of ints aligned on an 8-byte boundary with the following declaration:

```
struct pair {
    int x[2] __attribute__ ((aligned (8)));
};
```

If you choose, you can omit a specific alignment value and simply ask the compiler to align a variable or field appropriately for the target.

```
char s[3] __attribute__ ((aligned));
```

An aligned attribute lacking an alignment boundary causes the compiler to set the alignment automatically to the largest alignment ever used for any data type on the target machine. Alignment is a valuable optimization because it can often make copy operations more efficient. How? The compiler can use native CPU instructions to copy natural memory sizes when performing copies to or from aligned variables or fields.

TIP *The aligned attribute increases alignment; to decrease it, specify packed as well.*

The deprecated attribute has the same effect and behavior for variables as it does for functions.

The `mode (m)` attribute specifies the data type for the declaration with a type corresponding to the mode m . In effect, you declare an integer or floating-point type by width rather than type. Similarly, you can specify a mode of byte or `_byte_` to declare a mode of a 1-byte integer, word or `_word_` for a one-word integer mode, or pointer or `_pointer_` for the mode used to represent pointers.

The packed attribute requests allocating a variable or struct member with the smallest possible alignment, which is 1 byte for variables and 1 bit for a field. You can specify a larger alignment with the aligned attribute. The following code snippet illustrates a struct in which the field s is packed, which means that s immediately follows index—that is, there is no padding to a natural memory boundary.

```
struct node
{
    int index;
    char s[2] __attribute__ ((packed));
    struct node *next;
};
```

Normally, the compiler places the code objects in named sections such as `data` and `bss`. If you need additional sections or want certain particular variables to appear in special sections, you can use the attribute section (*name*) to obtain that result. For example, an operating system kernel might use the section name `.kern_data` to store kernel-specific data. The section attribute declares that a variable or function should be placed in a particular section.

The following small program, adapted from the GCC documentation, declares several section names, `DUART_A`, `DUART_B`, `STACK`, and `INITDATA`:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA"))) = 0;

int main(void)
{
    /* set up stack pointer */
    init_sp(stack + sizeof (stack));

    /* set up initialized data */
    memcpy(&init_data, &data, &edata - &data);

    /* enable the serial ports */
    init_duart(&a);
    init_duart(&b);
}
```

Note that the sample program uses the section attribute with an *initialized* definition of a *global* variable, such as the four global definitions at the beginning of the program. If you fail to use an initialized global variable, GCC will emit

a warning and ignore the section attribute applied to uninitialized variable declarations. This restriction exists because the linker requires each object be defined once. Uninitialized variables are temporarily placed in the .common (or .bss) section and so can be multiply defined. To force a variable to be initialized, specify the -fno-common flag or declare the variable using the nocommon attribute.

TIP *Some executable file formats do not support arbitrary section, so the section attribute is not available on those platforms. On such platforms, use the linker to map the contents of a module to a specific section.*

The transparent_union attribute is used for function parameters that are unions. A transparent union declares that the corresponding argument might have the type of any union member, but that the argument is passed as if its type were that of the first union member. The unused attribute has the same syntax and behavior as the unused attribute for functions.

Inline Functions

When you use `inline` with a function, GCC attempts to integrate that function's code into its callers. As an optimization, inline functions make execution faster by eliminating the overhead of function calls (saving and restoring stack pointers, for example). If the argument values are constant, they can be optimized at compile time, reducing the amount of function code integrated into the callers. Although at first sight inlining might seem to inflate code size, this is not necessarily the case. Other optimizations might allow sufficient code hoisting or subexpression elimination in such a way that the actual integrated code is smaller.

Although C99 includes inline functions, one of the shortcomings of GCC is that its implementation of inline functions differs from requirements of the standard. To declare a function inline, use the `inline` keyword in its declaration:

```
inline long cube(long i)
{
    return i * i * i;
}
```

NOTE *If you are writing a header file for inclusion in ISO C programs, use `_inline_` instead of `inline`.*

In the absence of the `inline` attribute, or in addition to it, you can instruct GCC to inline all “simple enough” functions by specifying the command-line option `-finline-functions`, where GCC decides what constitutes a “simple enough” function.

Code constructs that make it impossible (or, at the least, extremely difficult) to inline functions include using variadic arguments; calling `alloca()` in the function body; using variable-sized data types (such as variable-length arrays); jumping to computed and nonlocal gotos; calling functions before their definition; including recursive function calls with a function definition; and nesting functions. If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that cannot be inlined.

If you need to know when requested inlines could not be implemented, specify the command-line option `-Winline` to emit both a warning that a function could not be inlined and an explanation of why it could not be inlined.

For functions that are both `inline` and `static`, that function’s assembly language code will never be referenced if 1) all calls to that function are integrated into a calling function, and 2) the integrated function’s address is never taken. Accordingly, GCC will not even emit assembler code for the function, behavior you can override by specifying the option `-fkeep-inline-functions`.

Anticipating future compatibility with C99 semantics for inline functions, GCC’s developers recommend using only `static inline` for inline functions. Why? The existing semantics will continue to function as expected when `-std=gnu89` is specified, but the eventual default for `inline` will be the GCC’s behavior when `-std=gnu99` is specified. The issue is that `-std=gnu99` will implement the C99 semantics but that it does not yet do so.

TIP *GCC does not inline any functions when not optimizing unless you specify the `always_inline` attribute for the function, for example:*

```
inline void f (const char) __attribute__ ((always_inline));
```

Function Names As Strings

GCC predefines two magic identifiers that store the name of the current function. `__FUNCTION__` stores the function name as it appears in the source code; `__PRETTY_FUNCTION__` stores the name pretty printed in a language-specific fashion. In C programs, the two function names are the same, but in C++ programs, they will probably be different. To illustrate, consider the following code:

```

extern "C" {
extern int printf(char *, ...);
}

class c {
public:
void method_a(void)
{
    cout << "__FUNCTION__ = " << __FUNCTION__ << endl;
    cout << "__PRETTY_FUNCTION__ = " << __PRETTY_FUNCTION__ << endl;
}
};

int main(void)
{
    c C;
    C.method_a();
    return 0;
}

```

At runtime, the output is

```

$ ./a.out
__FUNCTION__ = method_a
__PRETTY_FUNCTION__ = void  C::method_a (void)

```

The compiler replaces the identifiers with a string literal containing the appropriate name. However, `__FUNCTION__` and `__PRETTY_FUNCTION__` are neither preprocessor macros like `__FILE__` and `__LINE__` nor variables. Because they are not macros, they concatenate easily with other string literals and they can be used to initialize char arrays. So, for example, the following code works as you would expect:

```

#include <stdio.h>
void here(void)
{
    char here[] = "Function " __FUNCTION__ " in " __FILE__;
}

int main(void)
{
    here();
    return 0;
}

```

Running the resulting program (the source file was here.c), you get

```
$ ./a.out
Function main in here.c
```

In C++, `_FUNCTION_` and `_PRETTY_FUNCTION_` are variables.

On the other hand, because they are not macros, `#ifdef _FUNCTION_` is meaningless inside a function because the preprocessor does not do anything special with the identifier `_FUNCTION_` (or `_PRETTY_FUNCTION_`).

CAUTION *These semantics are deprecated. GCC 3.2 handles `_FUNCTION_` and `_PRETTY_FUNCTION_` the same way as `_func_`, which is defined by the C99 standard. `_func_` is a variable, not a string literal, meaning that `_func_` cannot be concatenated with string literals.*

#pragmas Accepted by GCC

GCC supports several types of #pragmas, primarily in order to compile code originally written for other compilers. Note that in general we do not recommend the use of pragmas. In particular, GCC defines pragmas for ARM, Darwin, Solaris, and Tru64 systems.

ARM #pragmas

ARM targets define #pragmas for controlling the default addition of `long_call` and `short_call` attributes to functions.

- `long_calls`: Enables the `long_call` attribute for all subsequent functions
- `no_long_calls`: Enables the `short_call` attribute for all subsequent functions
- `long_calls_off`: Disables the `long_call` and `short_call` attributes for all subsequent functions

Darwin #pragmas

The following #pragmas are available for all architectures running the Darwin operating system. These are useful for compatibility with other Mac OS compilers.

- `mark token`: Accepted for compatibility, but otherwise ignored.
- `options align=target`: Sets the alignment of structure members. The values of `target` may be `mac68k`, to emulate m68k alignment, or `power`, to emulate PowerPC alignment. `reset` restores the previous setting
- `segment token`: Accepted for compatibility, but otherwise ignored.
- `unused (var [, var]...)`: Declares variables as potentially unused, similar to the effect of the attribute `unused`. Unlike the `unused` attribute, `#pragma` can appear anywhere in variable scopes.

Solaris #pragmas

For compatibility with the SunPRO compiler, GCC supports the `redefine_extname oldname newname` pragma. This `#pragma` assigns the C function `oldname` the assembler label `newname`, equivalent to the `asm` label's extension. The `#pragma` must appear before the function declaration. The preprocessor defines `_PRAGMA_REDEFINE_EXTNAME` if this `#pragma` is available.

Tru64 #pragmas

GCC supports the `extern_prefix string` `#pragma` for compatibility with the Compaq C compiler. `#pragma extern_prefix string` prefixes `string` to all subsequent function and variable declarations. To terminate the effect, use another `#pragma extern_prefix` with an empty string. The preprocessor defines `_PRAGMA_EXTERN_PREFIX` if this `#pragma` is available.

CHAPTER 5

Optimizing Code with GCC

THESE DAYS, COMPILERS ARE pretty smart. They can perform all sorts of code transformations, from simple inlining to sophisticated register analysis, that make compiled code run faster. In most situations, faster is better than smaller, because disk space is ridiculously cheap. However, on the embedded systems with which we routinely work, small is often at least as important as fast, because the systems on which we work sometimes have extreme memory constraints and no disk space. This chapter describes GCC's code optimization capabilities. By this point in the book, you have a pretty good grasp of how to compile your code and how to make GCC do what you want it to. The next step, accordingly, is to make your code faster or smaller. If your luck holds out, you might even be able to make your next program faster and smaller. After a quick, high-level overview of compiler optimization theory, you will look at GCC's command-line options for code optimization. First, you will look at GCC's general, architecture-independent optimizations. Then, you will learn about the architecture-specific optimizations.

A Whirlwind Tour of Compiler Optimization Theory

Given the code generated by a straightforward compilation, *optimization* refers to analyzing the compilation's results to determine how to transform the resulting code to run faster, consume less space, or both. Compilers that do this are known as *optimizing compilers* and the resulting code is known as *optimized code*. To produce optimized code, an optimizing compiler performs one or more transformations on the source code provided as input. The purpose is to replace less efficient code with more efficient code while at the same time preserving the meaning and ultimate results of the code. Throughout this section, we will use the term *transformation* to refer to the code modifications performed by an optimizing compiler because we want to distinguish between optimization techniques, such as loop unrolling and common subexpression elimination, and the way in which these techniques are implemented using specific code transformations.

Optimizing compilers uses several methods to determine where generated code can be improved. One method is *control flow analysis*, which examines

loops and other control constructs, such as if-then and case statements, to identify the execution paths a program might take and, based on this analysis, determine how the execution path can be streamlined. Another typical optimization technique examines how data is used in a program, a procedure known as *data flow analysis*. Data flow analysis examines how and when variables are used (or not) in a program and then applies various set equations to these usage patterns to derive optimization opportunities.

Optimization properly includes improving the algorithms that make up a program as well as the mechanical transformations described in this chapter. The classic illustration of this is the performance of bubble sorts compared to, say, quick sorts or Shell sorts. While code transformations might produce marginally better runtimes for a program that sorts 10,000 items using a bubble sort, replacing the naive bubble sort algorithm with a more sophisticated sorting algorithm will produce dramatically better results. The point is simply that optimization requires expending both CPU cycles and human intellectual energy.

For the purposes of this section, we define a *basic block* as a sequence of consecutive statements entered and exited without stopping or branching elsewhere in the program. Transformations that occur in the context of a basic block, that is, within a basic block, are known as *local transformations*. Similarly, transformations that do not occur solely within a basic block are known as *global transformations*. As it happens, many transformations can be performed both locally and globally, but the usual procedure is to perform local transformations first.

NOTE The classic compiler reference, affectionately known as the dragon book due to the dragon that appears on its cover, is Compilers: Principles, Techniques, and Tools, Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison Wesley Longman, 1986. ISBN: 0-201-10088-6.). We are indebted to this seminal text for the material on optimization discussed in this chapter.

Eliminating Redundant Computations

One way to improve performance is to reduce or eliminate redundant computations. Given an expression whose value is computed once, each subsequent occurrence of that expression can be eliminated and its value used instead, if the values of the variables in the expression have not changed since first computed. The subsequent occurrences of the expression are called *common subexpressions*. Consider the code fragment in Listing 5-1.

Listing 5-1. A Typical Common Subexpression

```
#define STEP 3
#define SIZE 100

int i, j, k;
int p[SIZE];

j = 2 * STEP;
for (i = 0; i < SIZE; ++i) {
    j = 2 * STEP;
    k = p[i] * j;
}
```

The expression `j = 2 * STEP` in the for loop is a common subexpression because its value is computed before entering the loop and its constituent variable, `STEP` (actually a macro), has not changed. Common subexpression elimination, or CSE, removes the repeated computation of `j` in the for loop. After CSE, the for loop might look like the following:

```
for (i = 0; i < 100 ++i) {
    k = p[i] * j;
}
```

Admittedly, the example is contrived, but it makes the point: CSE has eliminated 100 redundant computations of `j`. You will see this contrived example again, because we deliberately chose it to illustrate the types of transformations that can be applied even to simple code.

NOTE *Although the examples in this section use C, GCC performs transformations on an intermediate representation of a program called register transfer language (RTL). RTL is better suited than unmodified language-specific source code for compiler manipulations. We use C source code rather than RTL because C makes it easier to illustrate the effects of optimizing transformations.*

Another way to reduce or eliminate redundant computations is to perform copy-propagation transformations. Given an assignment of the form `f = g`, subsequent uses of `f`, called *copies*, use `g` instead. Consider a code fragment such as the following:

```
i = 10;
x[j] = i;
y[MIN] = b;
b = i;
```

Copy-propagation transformation might result in a fragment that looks like the following:

```
i = 10;
x[j] = 10;
y[MIN] = b;
b = 10;
```

Although it is not necessarily clear how copy propagation optimizes anything, it often reveals the possibility to apply other transformations, such as code motion and dead-code elimination.

Processor-Independent Optimizations

GCC's optimization knobs and switches fall into two broad categories: architecture-specific and architecture-independent optimizations. This section covers the *architecture-independent* optimizations, optimizations that do not depend on features specific to a given class of processors, such as Intel IA-32 CPUs, much less on a given iteration of a processor family, such as a Pentium IV (Xeon).

In the absence of optimization, GCC's goal, besides compiling code that works, is to keep compilation time to a minimum and generate code that runs predictably in a debugging environment. For example, in optimized code, a variable whose value is repeatedly computed inside a loop might be moved above the loop if the optimizer determines that its value does not change during the loop sequence. Although this is acceptable (if, of course, doing so does not alter the program's results), debugging the loop becomes impossible because you cannot set a breakpoint on that variable to halt execution of the loop. Without optimization, on the other hand, you can set a breakpoint on the statement computing the value of this variable, assign it a new value, and then continue execution and get the results you would have expected. This is what we meant by "code that runs predictably in a debugging environment."

Starting at the top, GCC's most well-known optimization switches are `-O`, its variant `-On`, where n is an integer between 0 and 3, and `-Os`. `-O0` turns off optimization. `-O` and `-O1` (which we will call *level 1 optimizations*) are equivalent, telling GCC to optimize code. With `-O` or `-O1`, the compiler attempts to minimize both code size and execution time without dramatically increasing compilation

time. Table 5-1 lists the optimizations performed when you specify `-O` or `-O1`. `-O2` and `-O3` increase the optimization level from that requested by `-O1`. To minimize code size, use option `-Os`.

Table 5-1. Optimizations Enabled with `-O` and `-O1`

Optimization	Description
<code>-fcprop-registers</code>	Attempts to reduce the number of register copy operations performed
<code>-fcross-jumping</code>	Collapses equivalent code to reduce code size
<code>-fdefer-pop</code>	Accumulates function arguments on the stack
<code>-fdelayed-branch</code>	Utilizes instruction slots available after delayed branch instructions
<code>-fguess-branch-probability</code>	Uses a randomized predictor to guess branch possibilities
<code>-fif-conversion</code>	Converts conditional jumps into nonbranching code
<code>-fif-conversion2</code>	Performs if-conversion using conditional execution (on CPUs that support it)
<code>-floop-optimize</code>	Applies several loop-specific optimizations
<code>-fmerge-constants</code>	Merges identical constants used in multiple modules
<code>-fomit-frame-pointer</code>	Omits storing function frame pointers in a register
<code>-fthread-jumps</code>	Attempts to reorder jumps so that they are arranged in execution order

The optimizations listed in Table 5-1 are enabled by default when you specify `-O` or `-O1`. To disable one of them while leaving the others enabled, negate the option using `no-` between `-f` and the optimization name. For example, to disable deferred stack pops, the command line might resemble this:

```
$ gcc myprog.c -o myprog -O1 -fno-defer-pop
```

NOTE `-f` denotes a flag whose operation is machine independent, that is, it requests an optimization that can be applied regardless of the architecture (in most cases). Flags, or flag options, modify GCC's default behavior at a given optimization level but do not require specific hardware support to implement the optimization. As usual, you can specify multiple flags as necessity demands.

Level 1 optimizations comprise a reasonable set of optimizations that include both size reduction and speed enhancement. For example, `-fcross-jumping` reduces multiple jumps to code that performs equivalent operations to fewer jumps, thereby reducing the overall code size. Fewer jump instructions mean that a program's overall stack consumption is smaller. `-fcprop-registers`, on the other hand, is a performance optimization that works by minimizing the number of times register values are copied around, saving the overhead associated with register copies.

`-fdelayed-branch` and `-fguess-branch-probability` are instruction scheduler enhancements. If the underlying CPU supports instruction scheduling, these optimization flags attempt to utilize the instruction scheduler to minimize CPU delays while the CPU waits for the next instruction.

The loop optimizations applied when you specify `-floop-optimize` include moving constant expressions above loops and simplifying test conditions for exiting loops. At level 2 and higher optimization levels, this flag also performs strength reduction and unrolls loops.

`-fomit-frame-pointer` is a valuable and popular optimization for two reasons: it avoids the instructions required to set up, save, and restore frame pointers and, in some cases, makes an additional CPU register available for other uses. On the downside, in the absence of frame pointers, debugging (such as generating stack traces, especially from deeply nested functions) can be difficult if not impossible.

`-O2` optimization (we refer to these as *level 2 optimizations*) includes all level 1 optimizations plus the additional optimizations listed in Table 5-2. Applying these optimizations will lengthen the compile time, but as a result you should also see a measurable increase in the resulting code's performance, or, rather, a measurable decrease in execution time.

Table 5-2. Optimizations Enabled with -O2

Optimization	Description
<code>-falign-functions</code>	Aligns functions on powers-of-2 byte boundaries
<code>-falign-jumps</code>	Aligns jumps on powers-of-2 byte boundaries
<code>-falign-labels</code>	Aligns labels on powers-of-2 byte boundaries
<code>-falign-loops</code>	Aligns loops on powers-of-2 byte boundaries
<code>-fcaller-saves</code>	Saves and restores register values overwritten by function calls
<code>-fcse-follow-jumps</code>	Follows jumps whose targets are not otherwise reached
<code>-fcse-skip-blocks</code>	Follows jumps that conditionally skip code blocks
<code>-fdelete-null-pointer-checks</code>	Eliminates unnecessary checks for null pointers

Table 5-2. Optimizations Enabled with -O2 (continued)

Optimization	Description
-fexpensive-optimizations	Performs “relatively expensive” optimizations
-fforce-mem	Stores memory operands in registers
-fgcse	Executes a global CSE pass
-fgcse-lm	Moves loads outside of loops during global CSE
-fgcse-sm	Moves stores outside of loops during global CSE
-foptimize-sibling-calls	Optimizes sibling and tail recursive function calls
-fpeephole2	Performs machine-specific peephole optimizations
-fregmove	Reassigns register numbers for maximum register tying
-frerun-cse-after-loop	Executes the CSE pass after running the loop optimizer
-frerun-loop-opt	Executes the loop optimizer twice
-fsched-interblock	Schedules instructions across basic blocks
-fsched-spec	Schedules speculative execution of nonload instructions
-fschedule-insns	Reorders instructions to minimize execution stalls
-fschedule-insns2	Performs a second schedule-insns pass
-fstrength-reduce	Replaces expensive operations with cheaper instructions
-fstrict-aliasing	Instructs the compiler to assume the strictest possible aliasing rules

The four `-falign-` optimizations force functions, jumps, labels, and loops, respectively, to be aligned on boundaries of powers of 2. The rationale is to align data and structures on the machine’s natural memory size boundaries, which should make accessing them faster. The assumption is that code so aligned will be executed often enough to make up for the delays caused by the no-op instructions necessary to obtain the desired alignment.

`-fcse-follow-jumps` and `-fcse-skip-blocks`, as their names suggest, are optimizations performed during the CSE optimization pass described in the first section of this chapter. With `-fcse-follow-jumps`, the optimizer follows jump instructions whose targets are otherwise unreached. For example, consider the following conditional:

```
if (i < 10) {
    foo();
} else {
    bar();
}
```

Ordinarily, if the condition (*i* < 10) is false, CSE will follow the code path and jump to *foo()* to perform the CSE pass. If you specify *-fcse-follow-jumps*, though, the optimizer will not jump to *foo()*, instead going to the jump in the *else* clause (*bar()*).

-fcse-skip-blocks causes CSE to skip blocks conditionally. Suppose you have an *if* clause with no *else* statement, such as the following:

```
if (i >= 0) {
    j = foo(i);
}
bar(j);
```

If you specify *-fcse-skip-blocks* and if, in fact, *i* is negative, CSE will jump to *bar()*, bypassing the interior of the loop statement. Ordinarily, the optimizer would process the body of the *if* statement, even if *i* tests false.

TIP *If you use computed gotos, one of the GCC extensions discussed in Chapter 4, you might want to disable global CSE optimization using the *-fno-gcse* flag. Disabling global CSE in code using computed gotos often results in better performance in the resulting binary.*

-fpeephole2 performs CPU-specific peephole optimizations. During *peephole optimization*, the compiler attempts to replace longer sets of instructions with shorter, more concise instructions. For example, given the following code:

```
a = 2;
for (i = 1; i < 10; ++i)
    a += 2;
```

GCC might replace the loop with the simple assignment *a* = 57. With *-fpeephole2*, GCC performs peephole optimizations using features specific to the target CPU instead of or in addition to standard peephole optimization tricks such as, in C, replacing arithmetic operations with bit operations.

-fforce-mem copies memory operands and constants into registers before performing pointer arithmetic on them. The idea behind these optimizations is to make memory references common subexpressions, which can be optimized

using CSE. As explained in the first section, CSE can often eliminate multiple redundant register loads, which incur additional CPU delays due to the load-store operation.

`-foptimize-sibling-calls` attempts to optimize away tail recursive or sibling call functions. A *tail recursive call* is a recursive function call made in the tail of a function. Consider the following code snippet:

```
int inc(int i)
{
    printf("%d\n" i);
    inc(i + 1);
}
```

This defines a function named `inc()` that displays the value of its argument, `i`, and then calls itself with 1 + its argument, `i + 1`. The tail call is the recursive call to `inc()` in the tail of the function. Clearly, though, the recursive tail call can be eliminated and converted to a simple iteration, albeit one that lacks a terminal point. `-foptimize-sibling-calls` attempts to perform this optimization. A *sibling call* refers to function calls made in a tail context that can also be optimized away.

The `-Os` option is becoming increasingly popular because it applies all of the level 2 optimizations except those known to increase the code size. `-Os` also applies additional techniques to attempt to reduce the code size. Code size, in this context, refers to a program's memory footprint at runtime rather than its on-disk storage requirement. In particular, `-Os` disables the following optimization flags (meaning they will be ignored if specified in conjunction with `-Os`):

- `-falign-functions`
- `-falign-jumps`
- `-falign-labels`
- `-falign-loops`
- `-fprefetch-loop-arrays`
- `-freorder-blocks`

You might find it instructive to compile a program with `-O2` and `-Os` and compare runtime performance and memory footprint. For example, we have found that recent versions of the Linux kernel have nearly the same runtime performance when compiled with `-O2` and `-Os`, but the runtime memory footprint is 15 percent smaller when the kernel is compiled with `-Os`. Naturally, your mileage may vary and, as always, if it breaks, you get to keep all of the pieces.

`-O3`, finally, enables all level 2 optimizations plus the following:

- `-finline-functions`: Integrates all “simple” functions into their callers
- `-frename-registers`: Reassigns unused registers after register allocation

NOTE If you specify multiple `-O` options, the last one encountered takes precedence. Thus, given the command `gcc -O3 foo.c bar.c -O0 -o baz`, no optimization will be performed because `-O0` overrides the earlier `-O3`.

In addition to the optimizations enabled using one of the `-O` options, GCC has a number of specialized optimizations that can only be enabled by specifically requesting them using `-f`. Table 5-3 lists these options.

Table 5-3. Specific GCC Optimizations

Flag	Description
<code>-fbounds-check</code>	Generates code to validate indices used for array access
<code>-fdefault-inline</code>	Compiles C++ member functions inline by default
<code>-ffast-math</code>	Sets <code>-fno-math-errno</code> , <code>-funsafe-math-optimizations</code> , and <code>-fno-trapping-math</code>
<code>-ffinite-math-only</code>	Disables checks for NaNs and infinite arguments and results
<code>-ffloat-store</code>	Disables storing floating-point values in registers
<code>-fforce-addr</code>	Stores memory constants in registers
<code>-ffunction-cse</code>	Stores function addresses in registers
<code>-finline</code>	Expands functions inline using the <code>inline</code> keyword
<code>-finline-functions</code>	Expands simple functions in the calling function
<code>-finline-limit=n</code>	Limits inlining to functions no greater than <i>n</i> pseudo-instructions
<code>-fkeep-inline-functions</code>	Keeps inline functions available as callable functions
<code>-fkeep-static-consts</code>	Preserves unreferenced variables declared <code>static const</code>

Table 5-3. Specific GCC Optimizations (continued)

Flag	Description
-fmath-errno	Sets errno for math functions executed as a single instruction
-fmerge-all-constants	Merges identical variables used in multiple modules
-ftrapping-math	Emits code that generates user-visible traps for FP operations
-ftrapv	Generates code to trap overflow operations on signed values
-funsafe-math-optimizations	Disables certain error checking and conformance tests on floating-point operations

Many of the options listed in Table 5-3 involve floating-point operations. In order to apply the optimizations in question, the optimizer deviates from strict adherence to ISO and/or IEEE specifications for math functions in general and floating-point math in particular. In floating-point heavy applications, you might see significant performance improvements, but the trade-off is that you give up compatibility with established standards. In some situations, noncompliant math operations might be acceptable, but you are the only one who can make that determination.

NOTE *Not all of GCC's optimizations can be controlled using a flag. GCC performs some optimizations automatically and, short of modifying the source code, you cannot disable these optimizations when you request optimization using -O.*

Processor-Specific Optimizations

Due to the wide variety of processor architectures GCC supports, we cannot begin to cover the processor-specific optimizations in this chapter. Appendix A covers, in detail, all of the processor-specific optimizations that you can apply, so curious readers who know the details of their CPUs are encouraged to review the material there. In reality, though, the target-specific switches discussed in Appendix A are not properly optimizations in the traditional sense. Rather, they are options that give GCC more information about the type of system on which the code being compiled will run. GCC can use this additional information to

generate code that takes advantage of specific features or that works around known misfeatures of the given processor.

Before starting work on this book, we usually used the (architecture-independent) -O2 option exclusively, and left it to the compiler to Do The Right Thing otherwise. After having written this book, we have expanded our repertoire, adding some additional options that we have found to be useful in specific cases. Our goal in this was to provide some guidelines and tips to help you select the right optimization level and, in some situations, the particular optimization you want GCC to apply. These can only be guidelines, though, because you know your code better than we do.

CHAPTER 6

Performing Code Analysis with GCC

THE PREVIOUS CHAPTER DISCUSSED the various types of optimizations that the GCC compilers can perform for you. Some of these optimizations are automatic, whereas others only make sense based on the characteristics of the application that you are trying to optimize. For example, you can only decide where and if to use optimizations such as unrolling or inlining loops after studying your application and identifying loops that might benefit from these optimizations. (Unrolling or inlining loops means to insert the code for each iteration of a loop in sequence so that a loop is no longer present; it is replaced by a series of sequential statements that explicitly perform the contents of the loop for each value of the variable that controls the loop.)

Determining how to analyze an application can be problematic. The most common mechanism for examining variables and analyzing the sequence and frequency of functions called in an application is the same mechanism generally used during debugging—sprinkling `printf()` function calls throughout your code. Aside from being time-consuming, this approach has the unfortunate side effect of changing application performance. Each `printf()` call has a certain amount of overhead, which can mask truly obscure problems such as timing and allocation conflicts in your applications. The `printf()` mechanism also shows only the sequence of function calls within one run of your application. It does not inherently provide information about the execution of those functions unless you do the calculations yourself and then display the statistics. Just like adding calls to `printf()` in the first place, performing execution time calculations in your own debugging functions can change the behavior that you are trying to observe in the first place.

Beyond simply identifying opportunities for optimization, analyzing an application is important for a variety of other reasons such as testing. One measure of a good test suite is that it exercises as much of the code for your application as possible. In order to come up with a good test suite, it is therefore important to be able to identify all of the possible sequences of function calls within your application, and you can do so with an item known as a *call graph*. Attempting to do this manually can be tedious even on a slow day, and requires continuous updating each time you change your application.

Analyzing the behavior, performance, and interaction between function calls in your applications is generally classified into two different, but related, types of analysis:

- *Code coverage analysis*: Shows how well any single run of the application exercises all of the functions in your application.
- *Code profiling*: Provides performance information by measuring the behavior of and interaction between the functions in any single run of your application. There are many different types of profiling—the most common of these is *execution profiling*, which measures and reports on the time spent in each function during a run of an application.

Luckily, (and not too surprisingly) the GNU Compiler Collection addresses both of these needs for developers:

- *Code coverage*: GCC includes an application and two compilation options that make it easy to perform code analysis. The GNU Coverage application, gcov, automatically creates a call graph for your applications, helping you identify how well any given test code exercises the various execution paths and relationships between the functions in an application. This application produces some general profiling information, but its focus is on coverage analysis.
- *Code profiling*: GCC also provides a number of compilation options that make it easy for you to use code profilers such as the GNU profiler, gprof.

This chapter explains how to do code coverage analysis using gcc compilation options and gcov, and then discusses code profiling. The section “Test Coverage Using GCC and gcov” provides an overview of test coverage mechanisms, explains gcc compilation options related to test coverage, and then presents examples of using gcov for coverage analysis and discusses the coverage-related files produced during compilation and by the gcov application. The section “Code Profiling Using GCC and gprof” explains gcc options related to profiling, and then discusses how to use gprof to identify potential bottlenecks and areas for possible optimization in your applications.

Test Coverage Using GCC and gcov

Test coverage is the measurement of how well any given code or run of an application exercises the code that makes up the application. The next few sections provide an overview of test coverage, highlight the options provided by gcc for test coverage, discuss all of the auxiliary data files produced by those options, and supply examples of using gcov to perform test coverage analysis.

Before you can explore the options provided in gcc for performing coverage analysis, you need an understanding of test coverage basics. In the next section, we give you an overview of the most common types of test coverage and explore situations in which tests must be cleverly constructed to augment the built-in test coverage capabilities provided by gcc.

Overview of Test Coverage

A variety of different approaches to test coverage are commonly used to measure how well certain tests exercise a given application. The most common of these are the following:

- *Statement coverage*: Measures whether each statement in an application is exercised. Statements are not necessarily the same things as lines of code, since single lines of code can contain multiple statements, though this is poor practice from a testing point of view. Test output that lists every line of code in an application and whether it is covered can be too voluminous to manage properly. For this reason, most statement coverage testing identifies test coverage in terms of basic blocks of statements that are sequential and nonbranching. This “statement coverage shorthand” is usually referred to as *basic block coverage*. As explained later in this chapter, gcc provides built-in support for basic block coverage reporting. Statement coverage and similar approaches are also often referred to as *line coverage*, *block coverage*, and *segment coverage*.
- *Decision coverage*: Measures whether all of the possible decisions within an application are being tested with all possible values. Decision coverage is essentially a superset of statement coverage, because in order to exercise all values of all conditional expressions within your code, you have to execute all of the statements within the code for each conditional. Decision coverage is also commonly referred to as *branch coverage* or *all edges coverage*.
- *Path coverage*: Measures whether all of the possible execution paths within an application are being tested. This usually involves creating a truth table for all of the functions within an application and ensuring that all of the permutations recorded in the truth table are being tested. Complete path coverage is impossible in a single run of an application that has sequential, opposing conditionals, as only one or the other can be true. Path coverage is essentially a superset of decision coverage; in order to exercise all possible execution paths in an application, you have to execute all of its code, including all of the code in each conditional, at one time or another. Path coverage is also referred to as *predicate coverage*.

- *Modified condition decision coverage:* Measures whether all of the expressions that lead to each decision or execution path are being tested. For example, modified condition decision coverage measures whether tests cover conditionals predicated upon multiple conditions combined by and or or statements. Modified condition decision coverage is also referred to as *condition coverage*, *condition-decision coverage*, and *expression coverage*.

All of these coverage metrics look at your code in different ways, and each have their own advocates. Basic block coverage is the type of test coverage that is built into gcc, and is the most common test coverage mechanism in general use throughout the industry. By creating detailed test code and test suites, you can use basic block coverage to effectively provide other types of test coverage.

Designing effective test suites is almost an art form, and is certainly a science and discipline with its own rules. To design such test suites requires that you understand how the application you are testing works, and also that you take into account any syntactic peculiarities of the language in which the application you are testing was written. For example, it can be difficult to achieve 100 percent coverage of your code using statement or basic block coverage for applications written in C and C++ for a variety of reasons. The most common of these are blocks of code that are designed to catch system error conditions and complex, single-line statements.

Even with well-crafted test suites and applications that are designed to be testable, it is difficult to exercise portions of a robust application that are designed to catch abnormal error conditions that should either never happen or are difficult to induce. An example of a hard-to-induce statement is an error message displayed after using an exec call in a standard Linux/Unix fork()/exec() statement for spawning a child process from within an application. Since an exec() call replaces the code for a running process or thread with an instance of another program, it should never return, and any subsequent statements (usually error messages) should never be seen. An example of this is the fprintf() statement in the following block of sample code:

```
child = fork ();
if (child == 0)
    execvp (program, argument_list);
    fprintf (stderr, "Error in execvp - aborting...\n");
    abort ();
}
```

In this case, the fprintf() statement could only be reached if the execvp() call failed, which would only happen if the program could not be found or an external system error condition occurred. It should be difficult for your test code to reproduce improbable errors in your application. If it is not, perhaps they are not so improbable.

The C and C++ languages provide a commonly used syntax for embedding decisions within single statements such as the following:

```
result = my_function() ? foo : bar ;
```

In this example, the value of result is foo if the call to `my_function()` returns successfully, and bar if it does not. In statement or basic block coverage approaches to testing, this type of expression is identified as being tested if it is executed, which does not necessarily say anything about whether the code in `my_function()` is correct.

In general, it is important to note that statement and basic block coverage only measure whether statements are exercised by a set of tests, not whether those statements are logically correct or do what you intend them to do.

A truly good test or set of tests not only executes all of the functions within an application, but also exercises all of the possible execution paths within that application. At first, this seems to be both simple and obvious on the surface, as most of the code in an application consists of lines, sequences of statements, loops, or simple conditionals that look something like the following:

```
if variable1
foo;
else
bar;
```

Simple conditionals such as if-then-else clauses or their cousins, the switch statements found in programming languages such as C and C++, are easy enough to exercise and examine. Your tests execute your code, providing an instance of each possible value that is being tested for in the conditional clause, and then, if possible, executes the application with a value that is not being tested for to ensure that any open else clause (or the switch statement's default case) is exercised. This type of test coverage provides both statement coverage, in which each statement is tested and executed, and decision coverage, in which every possible execution path within the code is tested by making sure that every possible decision within the code is executed once.

Additional testing complexities arise when single lines contain multiple decision points and where those decision points contain compound logical statements. In these cases, it can be easy to overlook test cases that exercise every possible execution path through the code. As an example, consider the following block of pseudo-code:

```
if (variable1 and (variable2 or function1()))
foo;
else
bar;
```

Most test code would consider this block of code to be fully covered if it caused both the foo and bar statements to be executed. However, both of these branches could be tested by simply manipulating the values of variable1 and variable2—in other words, without ever executing the function `function1()`. This function could not only perform any number of illegal operations, but also manipulate global variables that would substantially change the execution environment of the rest of the program.

Testing applications is a science unto itself. As you will see throughout the rest of this chapter, the GNU Compiler Collection provides excellent built-in support for coverage analysis. For more detailed information about software testing, designing code for testability, and automating software testing, here are a few good references:

- *Automated Software Testing: Introduction, Management, and Performance*, Elfriede Dustin, Jeff Rashka, John Paul (Addison-Wesley, 1999. ISBN: 0-201-43287-0.)
- *Software Test Automation: Effective Use of Test Execution Tools*, Mark Fewster, Dorothy Graham (Addison-Wesley, 1999. ISBN: 0-201-33140-3.)
- *Software Testing and Continuous Quality Improvement*, William Lewis (CRC Press, 2000. ISBN: 0-849-39833-9.)
- *Systematic Software Testing*, Rick D. Craig, Stefan P. Jaskiel (Artech House, 2002. ISBN: 1-580-53508-9.)
- *Testing Computer Software*, Cem Kaner, Hung Q. Nguyen, Jack Falk (John Wiley & Sons, 1999. ISBN: 0471358460.)

Compiling Code for Test Coverage Analysis

In order to use the `gcov` test coverage tool to produce coverage information about a run of your application, you must

- Compile your code using `gcc`. The `gcov` application is not compatible with embedded coverage and analysis information produced by any other compiler.
- Use the `-fprofile-arcs` option when compiling your code with `gcc`.
- Use the `-ftest-coverage` option when compiling your code with `gcc`.
- Avoid using any of the optimization options provided by `gcc`. These options might modify the execution order or grouping of your code, which would make it difficult or impossible to map coverage information into your source files.

Compiling source code with the `-fprofile-arcs` option causes gcc to instrument the compiled code, building a call graph for the application. A *call graph* is a list identifying which functions are called by other functions, and which functions call other functions. The representation of each call from one function to another in a call graph is known as a *call arc*, or simply an *arc*.

Running an application compiled with `-fprofile-arcs` creates a file by the same name as the original source file but with the `.da` extension in the directory where the source file is located.

NOTE *The information in a .da (arc data) file is cumulative. Compiling your code with the `-fprofile-arcs` option and then running it multiple times will add all of the information about all runs to the existing .da file. If you want to guarantee that a .da file only contains profiling and coverage information about a single run on your application, you must delete any existing .da file before running the application.*

Compiling source code with the `-ftest-coverage` option causes gcc to identify and track each basic block within the source file. This information is recorded in two files with the same name as the original source file but with a `.bb` (basic block) and `.bbg` (basic block graph) extension that replaces original file extensions such as `.c` or `.cc`. These files are created at compile time (rather than at runtime, like the `-fprofile-arcs` option's `.da` file), and are located in the directory where the source file is located.

The `.bb` file contains a list of source files and each function within that file, and records the line numbers in the source file that are associated with the beginning and end of each basic block in the file. As mentioned previously, a basic block is a series of sequential statements that must be executed in order and together without any branching. The `.bbg` file contains a list of the arcs from each basic block to another in the source file (which are possible branches taken from one basic block to another). Together, the information recorded in these two files makes it possible for gcov to reconstruct program flow and annotate source files with coverage and profiling information.

For more information about `.da`, `.bb`, and `.bbg` files, see the section “Files Used and Produced During Coverage Analysis.”

Using the gcov Test Coverage Tool

After compiling your source files with the `-fprofile-arcs` and `-ftest-coverage` options, as explained in the previous section, you run your application normally (usually with some test scenario). This produces the summary information in

the .da file. You can then use gcov to generate coverage information for any of the source files in your application, as well as display the profiling information produced by that run of your code.

When run on any of your source modules, the gcov program uses the information in the .bb, .bbg, and .da files to produce an annotated version of your source file, with the extension .gcov. For example, running gcov on the module fibonacci.c produces the annotated source file fibonacci.c.gcov. The annotated source files produced by gcov use rows of hash marks to delineate the beginning and end of any basic block identified in the source file, and also provide numeric summaries of the number of times that each line or basic block in the files was executed.

The gcov program provides a number of command-line options that you can use to get information about gcov or specify the information produced by gcov. These command-line options are shown in Table 6-1.

Table 6-1. Options for the gcov Program

gcov Option	Description
-b, --branch-probabilities	Using either of these command-line options enables you to see how often each branch in your application was executed during the sample run(s). Either of these options causes gcov to include branch frequency information in the annotated source.gcov file, showing how often each branch in your application was executed, expressed as a percentage of the number of times the branch was taken out of the number of times that it was executed. This option also causes gcov to display summary information about program branches to gcov's standard output.
-c, --branch-counts	Using either of these command-line options in conjunction with the -b or --branch-probabilities option modifies the annotated branch count information to show branch frequencies as the number of branches executed rather than the percentage of branches executed. These options do not modify the summary information produced to gcov's standard output by the -b or --branch-probabilities options.
-f, --function-summaries	Use these options to output summary information about function calls in the specified source file in addition to the summary information about all calls in the file.

Table 6-1. Options for the gcov Program (continued)

gcov Option	Description
<code>-h, --help</code>	These options display help about using gcov and exit without analyzing any files.
<code>-l, --long-file-names</code>	When creating annotated gcov output files, these options name the annotated versions of included files by concatenating the name of the source files in which they were found to the name of the included file, separated by two hash marks. This can be very useful when the same include file is referenced in multiple source files. For example, if the files foo.c and bar.c both included baz.h, running gcov with the -l option on the file foo.c would produce the output files foo.c.gcov and foo.c##baz.h.gcov. Similarly, running gcov with the -l option on the file bar.c would produce the output files bar.c.gcov and bar.c##baz.h.gcov.
<code>-n, --no-output</code>	Use these options to specify you do not want the gcov output file created.
<code>-o directory file,</code> <code>--object-directory</code> <code>directory,</code> <code>--object-file</code> <i>file</i>	These options specify either the directory containing the gcov data files or the path to the object file itself. If this option is not supplied, it defaults to the current directory. This option is useful if you have moved a directory containing instrumented source code and want to produce test coverage information without recompiling.
<code>-v, --version</code>	Use these options to display version information about the gcov application that you are running and exit without processing any files.

The next section of this chapter shows an example run of an application and gcov, highlighting the files produced and information displayed at each step.

A Sample gcov Session

This section provides an example of using gcov to provide test coverage information for a small sample application that calculates a specified number of values in the Fibonacci sequence. Listing 6-1 shows the main routine for this application, stored in the file fibonacci.c. Listing 6-2 shows one external routine for this

application, which is stored in the file `calc_fib.c`. The application is stored in multiple files in order to illustrate using gcov with applications constructed from multiple source files, rather than from any programmatic necessity.

Listing 6-1. The Source Code for the Sample fibonacci.c Application

```
/*
 * Simple program to print a certain number of values
 * in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

int calc_fib(int n);

int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
        n = atoi(argv[1]);
    else {
        printf("Usage: fibonacci num-of-sequence-values-to-print\n");
        exit(-1);
    }
    for (i=0; i < n; i++)
        printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}
```

Listing 6-2. The Source Code for the Auxiliary calc_fib.c Function

```
/*
 * Function that actually does the Fibonacci calculation.
 */

int calc_fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else
        return((calc_fib(n-2) + calc_fib(n-1)));
}
```

Before running `gcc` or `gcov`, the contents of the directory containing the source code for the sample application are as follows:

```
$ ls
calc_fib.c fibonacci.c Makefile
```

Chapter 7 explains how to create Makefiles automatically using the GNU Autoconf and Automake programs. For the time being, a simple Makefile that contains the following statement would be fine:

```
fibonacci: fibonacci.o calc_fib.o
```

First, compile the application with the `gcc` options necessary to produce the basic block and call graph information used by `gcov`:

```
$ gcc -fprofile-arcs -ftest-coverage fibonacci.c calc_fib.c -o fibonacci
```

After compilation completes, the contents of the sample application directory are as follows:

```
$ ls
calc_fib.bb    calc_fib.c      fibonacci.bb   fibonacci.c
calc_fib.bbg   fibonacci      fibonacci.bbg  Makefile
```

This illustrates that the `.bb` and `.bbg` files are produced by `gcc` during compilation. As mentioned earlier in this chapter, these files contain information that will be used by `gcov` to annotate any source files that you examine, identifying the number of times each statement or basic block was executed.

Next, we will run the sample application, specifying the command-line argument `11` to generate the first 11 numbers in the Fibonacci sequence:

```
$ ./fibonacci 11
0 1 1 2 3 5 8 13 21 34 55
```

After running the application, the contents of the sample application's source directory are as follows:

```
$ ls
calc_fib.bb    calc_fib.c      fibonacci       fibonacci.bbg  fibonacci.da
calc_fib.bbg   calc_fib.da    fibonacci.bb    fibonacci.c     Makefile
```

Running the application created the .da files for each of the source files that comprise the sample Fibonacci application. These files are automatically produced in the directory where the source code was compiled. If for some reason the directory where the application's source code has been moved or no longer exists, you would see messages like the following when running the application:

```
$ ./fibonacci 11
0 1 1 2 3 5 8 13 21 34 55
arc profiling: Can't open output file /home/wvh/src/fib/fibonacci.da.
arc profiling: Can't open output file /home/wvh/src/fib/calc_fib.da.
```

After running the application, all of the necessary data files used by gcov are now available, so you can run gcov on a source file to see coverage and summary profiling information:

```
$ gcov fibonacci.c
80.00% of 10 source lines executed in file fibonacci.c
Creating fibonacci.c.gcov.
```

Listing the application's src directory now shows that the file fibonacci.c.gcov has been produced by gcov:

```
$ ls
calc_fib.bbg    calc_fib.da      fibonacci.bbg      fibonacci.da
calc_fib.bbg    fibonacci        fibonacci.c       Makefile
calc_fib.c      fibonacci.bb    fibonacci.c.gcov
```

Listing 6-3 shows the file fibonacci.c.gcov. This file is an annotated version of the fibonacci.c source file in which execution counts and basic block information are displayed. Note that only the fibonacci.c.gcov file was created—no gcov output file was produced for the other source file used in the sample application. You must separately execute gcov on every source file for which you want to display coverage and summary profiling information.

Listing 6-3. The Output File Produced by gcov for the Source File fibonacci.c

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

int calc_fib(int n);

1   int main(int argc, char *argv[]) {
1       int i,n;
```

```

1      if (argc == 2)
1          n = atoi(argv[1]);
else {
#####
    printf("Usage: fibonacci num-of-sequence-values-to-print\n");
#####
    exit(-1);
}
12     for (i=0; i < n; i++)
11         printf("%d ", calc_fib(i));
1     printf("\n");
1     return(0);
}

```

The output files produced by gcov contain an indented version of any routines in the specified source file. The beginning and end of each basic block in the source file is marked with a row of hash marks. The numbers in the column at the left of the source code indicate the number of times each line was executed. Listing 6-3 shows that the fibonacci.c source file only contains one basic block, which consists of two statements (and that, in this case, the basic block was never executed because it is a usage message displayed when no command-line arguments are supplied).

NOTE Lines beginning with the string “#####” in profiling output indicate that these lines were never executed during this profiling run.

Some amount of summary profiling information is always collected when executing code that has been instrumented during compilation by using gcc’s -fprofile-arc and -ftest-coverage options. Using gcov’s -b (--branch-probabilities) option when running gcov on an instrumented source module displays this information and also incorporates it into the annotated filename.c.gcov module produced by gcov. Continuing with the fibonacci.c example, executing gcov with the -b option on the fibonacci.c file displays the following output:

```

$ gcov -b fibonacci.c
80.00% of 10 source lines executed in file fibonacci.c
100.00% of 5 branches executed in file fibonacci.c
80.00% of 5 branches taken at least once in file fibonacci.c
60.00% of 5 calls executed in file fibonacci.c
Creating fibonacci.c.gcov.

```

Listing 6-4 shows the output file fibonacci.c.gcov produced by running gcov with the -b option. Note that the output file still identifies basic blocks and provides line execution counts, but now also identifies the line of source code associated with each possible branch and function call. The annotated source code also displays the number of times that branch was executed. The construction of the branch percentages around the for loop in Listing 6-4 and its call to the printf() and calc_fib() functions is interesting. You can see that the for statement was executed 12 times, the last of which it exited, so this line was executed 11 out of 12 times, or 92 percent of the time. Of the 11 times that the loop body was executed, the loop exited the single time that the test was false, or 100 percent of the time the exit condition was satisfied. The increment operation was executed each time the for loop was executed, which was 11 out of 11 times, or 100 percent of the time. The printf() call and its internal call to the calc_fib() routine were each executed 11 out of 11 times, so both of these were executed 100 percent of the time. All of these statistics help you get a feel for the execution path that a given test run takes through your application. This not only shows which code is being used, but also helps you identify dead (unused) code, or code paths that you had expected to be taken more frequently.

Listing 6-4. Branch-Annotated Source Code Showing Branch Percentages

```
/*
 * Simple program to print a certain number of values
 * in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

int calc_fib(int n);

1 int main(int argc, char *argv[]) {
1     int i,n;

1     if (argc == 2)
branch 0 taken = 0%
1         n = atoi(argv[1]);
branch 0 taken = 100%
else {
#####
    printf("Usage: fibonacci num-of-sequence-values-to-print\n");
call 0 never executed
#####
    exit(-1);
```

```

call 0 never executed
    }
12      for (i=0; i < n; i++)
branch 0 taken = 92%
branch 1 taken = 100%
branch 2 taken = 100%
11      printf("%d ", calc_fib(i));
call 0 returns = 100%
call 1 returns = 100%
1      printf("\n");
call 0 returns = 100%
1      return(0);
}

```

The branch percentages displayed in the annotated source file show the number of times the branch was executed divided by the number of times the branch was executed. Using gcov's -c option in conjunction with the -b option causes the branch counts in the annotated source code to be measured absolutely rather than as a percentage. The output of the command is the same, as shown in the following output sample, but Listing 6-5 shows the difference in the contents of the fibonacci.c.gcov file.

Listing 6-5. Annotated Source Code Showing Branch Counts

```

$ gcov -b -c fibonacci.c
80.00% of 10 source lines executed in file fibonacci.c
100.00% of 5 branches executed in file fibonacci.c
80.00% of 5 branches taken at least once in file fibonacci.c
60.00% of 5 calls executed in file fibonacci.c
Creating fibonacci.c.gcov.
/*
 * Simple program to print a certain number of values
 * in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

int calc_fib(int n);

1 int main(int argc, char *argv[]) {
1     int i,n;

```

```

1           if (argc == 2)
branch 0 taken = 0
1           n = atoi(argv[1]);
branch 0 taken = 1
else {
#####
printf("Usage: fibonacci num-of-sequence-values-to-print\n");
call 0 never executed
#####
exit(-1);
call 0 never executed
}
12         for (i=0; i < n; i++)
branch 0 taken = 11
branch 1 taken = 1
branch 2 taken = 11
11         printf("%d ", calc_fib(i));
call 0 returns = 11
call 1 returns = 11
1         printf("\n");
call 0 returns = 1
1         return(0);
}

```

If you are interested in summary information about function calls in the specified source file, gcov's -f option displays summary information about the lines of source code executed in each function as well as summary information about the complete source file. The sample fibonacci.c source file contains a single routine, so the function and source code summaries are identical, as in the following output:

```

$ gcov -f fibonacci.c
80.00% of 10 source lines executed in function main
80.00% of 10 source lines executed in file fibonacci.c
Creating fibonacci.c.gcov.

```

If the source file had contained more than one routine, the function summary information would have provided statistical information about each function.

Using gcov to perform coverage analysis on C and C++ source files that include other source files (typically .h files containing definitions) produces an annotated source.gcov file for the main file and for any included files. If you are performing coverage analysis on multiple source files from a single application, this can be confusing if the same include file is referenced in multiple source files. Since the whole idea of include files is that they facilitate modular development and compilation of C and C++ applications by being included in multiple source files, this is a fairly common occurrence in multifile applications. When analyzing files that include other source files, you can use gcov's -l (or --long-names) options

to cause gcov to produce annotated include files whose names also contain the name of the source file in which they were included. For example, if the files foo.c and bar.c both include baz.h, running gcov with the -l option on the file foo.c would produce the output files foo.c.gcov and foo.c##baz.h.gcov. Similarly, running gcov with the -l option on the file bar.c would produce the output files bar.c.gcov and bar.c##baz.h.gcov.

Files Used and Produced During Coverage Analysis

The gcov application uses three files for each source file that has been compiled for coverage analysis. These files have the same basename as the source files that they are associated with, except that the original file extension is replaced with the .bb, .bbg, and .da extensions. These files are always created in the same location as the source files that they are associated with. None of these files is designed to be read or directly used without access to the data structure and entry information in the source and include files for gcov.

The .bb and .bbg files are generated when the source file is compiled by gcc if the -fcoverage option was specified. Neither of these files is designed to be readable by humans. The .bb file contains a list of all source files associated with the current source file, including a list of any included source files (such as .h files). This file also lists all of the functions within these source files, and identifies the line numbers associated with each basic block or single statement.

The .bbg file contains a list of the program flow arcs (possible branches taken from one basic block to another) for each function in the main source file or any included source file.

The .da file, created when you run your application, contains information about function calls and execution paths in one or more runs of the application. Once this file has been created by a run of your application, subsequent runs of your application append information to this file. To guarantee that a .da file contains information about a single run of your application, you should delete any existing .da file before running your application.

For detailed information about the internal structure of these files, see the gcov information in the GNU GCC documentation.

Code Profiling Using GCC and gprof

As discussed in the introduction to this chapter, profiling an application determines how often each portion of the application executes and how it performs each time it executes. The goal of most application profiling work is to identify the portions of your application that are the most resource intensive, either computationally or in terms of memory consumption. Once you have this information, you can then look for ways to improve the performance of those functions or your entire application in those areas.

The gcov application, discussed in the first half of this chapter, is designed to provide information about how well test runs of your application exercise its code and conditionals. In doing so, gcov provides some general profiling information that tells you how often each basic block (sequence of uninterruptible, sequential statements) executes during one or more test runs of the application. The GNU gprof application is designed for profiling and provides much more detailed analysis of your application and the functions it executes internally than gcov can provide. The GNU gprof application was inspired by the BSD Unix application (also found in many System III and System V Unix variants). Because gcc is widely used on a variety of systems, the gprof application provides options to produce output in the file formats used by prof, in case prof is still supported and used on your computer's operating system.

The GNU gprof application provides several forms of profiling output:

- *A flat profile:* Shows the amount of time your application spent in each function, the number of times each function was called, the amount of time spent in profiling-related overhead in each function, and an estimation of the margin of error in the profiling data.
- *A call graph:* Shows the relationships between all of the functions in your application. The call graph shows the calling sequence for the functions in your application, and also shows how much time was spent in each function and any other functions that it called.
- *An annotated source code listing:* Shows the number of times that each line in the program's source code was executed.

As with gcov, the internal code that is added to your application in order to produce profiling data is done by specifying a variety of options when compiling your application using gcc. These options tell gcc to automatically instrument each function in your application so that it produces profiling data that is automatically written to external files that can subsequently be displayed and analyzed by gprof.

As you will see later in this section, the profiling options that are automatically available to you when using gcc also enable you to write your own profiling routines. These routines will be inserted in your compiled code, which will be executed at the entry and exit points of each function in the application. This enables you to extend and customize the profiling functionality you get for free with the GCC compiler package. Not a bad deal.

Obtaining and Compiling gprof

Unlike gcov, gprof is not part of the standard GCC distribution. The gprof application is a part of the binutils package, which is a collection of GNU utilities for

managing and manipulating binary object files. The binutils package includes the GNU Linker (`ld`), the GNU Assembler (`as`), the GNU Library Archiver (`ar`) and its companion indexer (`ranlib`), the `addr2line` utility for mapping binary addresses to lines of source code, the `objcopy` and `objdump` utilities for converting and dumping object files, and many other GNU/Linux favorites.

At the time this book was written, the latest version of the binutils package was 2.13. You can obtain the latest version of the binutils package from its home page at <http://sources.redhat.com/binutils>. Once you have obtained the binutils package, compiling and installing it is much like you would do with any other GNU package:

1. Download the source archive from <http://sources.redhat.com/binutils>. The home page lists the latest version—this example uses <http://ftp.gnu.org/gnu/binutils/binutils-2.13.tar.gz>, which was the latest version at the time this book was written.
2. Extract the contents of the source archive using a command like the following:

```
tar zxvf binutils-2.13.tar.gz
```

3. Change your working directory to the directory produced in the previous step. In this example, this is the directory `binutils-2.13`:

```
cd binutils-2.13
```

4. Run the `configure` script to configure the archive for your system's configuration and to create the Makefiles for the various libraries and utilities in the package:

```
./configure
```

5. Build all of the utilities in the package by using a single top-level `make` command:

```
make
```

6. If there were no errors in the previous step, install the updated utilities, related libraries, and associated man pages and other online information using the following two-level `make` command:

```
make install
```

By default, this sequence of commands will install the updated version of `gprof` (and the other utilities in the binutils package) in `/usr/local/bin`. Make

sure that this directory is located in your path before /usr/bin, the traditional installed location of the gprof utility, or you may accidentally run an older version of gprof. You can determine the gprof binary that appears first in your path by executing the which gprof command. You can then determine the version of that copy of gprof by executing the command gprof -v.

Compiling Code for Code Profile Analysis

In order to cause your application to produce profiling information that can subsequently be analyzed by gprof, you must

- Compile your code using gcc. The gprof application is not inherently compatible with embedded profiling and analysis information produced by any other compiler.
- Use the -pg option when compiling your code with gcc to activate and link the profiling libraries

NOTE *If you are using gcc on a Solaris system or other system that provides the older Unix prof application, you can produce profiling information suitable for analysis using prof by compiling your code with the -p option rather than the -pg option.*

- (Optional) Use the -g option if you want to do line-by-line profiling. This standard debugging option tells the compiler to preserve the symbol table in the compiled code, but also inserts the debugging symbols that gprof (and gdb) use to map addresses in the executable program back to the lines of source code that they were generated from.
- (Optional) Use the -finstrument-functions option if you have written your own profiling functions to be called when each instrumented function is entered and before it is returned from. Using this option and writing profiling functions is discussed in the section “Adding Your Own Profiling Code Using gcc.”

During initial profiling, you also want to avoid using any of the optimization options provided by gcc. These options might modify the execution order or grouping of your code, which would make it difficult or impossible to identify areas for optimization.

Running an application compiled with the `-pg` option creates the file `gmon.out` when your program exits normally. Unlike the `source.da` coverage information file produced by `gcov`, the `gmon.out` file is created each time you run your application and only contains information about a single run of the application. Unlike the output files produced by `gcov`, this file is produced in the working directory of your program upon exiting. For example, programs that use the `chdir()` or `chroot()` functions will create the `gmon.out` file in the last directory they set using `chdir()`, or relative to the new file system root directory set using `chroot()`. If a file by the name `gmon.out` already exists in the directory where your program exists, its contents will be overwritten with the new profiling information.

NOTE *Because the gmon.out file is produced when your program exits normally (by returning from the main() function or by calling the exit() function), programs that crash or are terminated by a signal will not create the gmon.out file. Similarly, the gmon.out file will not be produced if your program exits by calling the low-level _exit() function.*

Versions of GCC prior to GCC 3.2 used the `-a` option to generate basic block profiling information, writing this information to a file named `bb.out`. This option and the output file it produced are obsolete—this functionality is now provided by using `gcov` to produce an annotated source code listing after compiling your code with the basic block coverage options discussed in the section “Compiling Code for Test Coverage Analysis.”

Using the gprof Code Profiler

After compiling your source files with the options discussed in the previous section, you run your application normally (usually with some test scenario). This produces profiling information in the `gmon.out` file. You can then use `gprof` to display profiling information in a number of different ways. The `gprof` application can even use the information in the `gmon.out` file to suggest automatic optimizations, such as function ordering in your executable or link ordering during the linking phase of compiling your application.

The next section of this chapter shows an example run of an application and `gprof`, highlighting the types of information that you may commonly want to produce and examine using `gprof`.

The `gprof` program provides a huge number of command-line options that you can use to specify the types of information produced by `gprof` or the types of analysis that it performs on the data in the `gmon.out` file.

Symbol Specifications in gprof

Some of the command-line options supported by gprof enable you to restrict the output of gprof to specific functions, any functions used in a specific file, or any functions used in a specific line of a specific file. These are known as *symbol specifications*. The syntax of a symbol specification can be one of the following:

- *filename*: Causes gprof to produce profiling output for any function called in the specified source file. For example, setting fibonacci.c as a symbol specification causes gprof to produce profiling output only for any functions called in the file fibonacci.c.
 - *function-name*: Causes gprof to produce profiling output only for any function in the source code with the specified name. For example, setting the symbol specification to calc_fib causes gprof to produce profiling output only for the function calc_Fib() and any functions that it calls. If you have multiple functions with the same name (e.g., global and local functions), you can specify a function from a particular source file by using the notation *source-file:function-name*.
 - *filename:line-number*: Causes gprof to produce profiling information only for any functions that are called from the specified line in the specified source file.
-

Table 6-2 shows the command-line options provided by gprof.

Table 6-2. Command-Line Options for the gprof Program

Option	Description
-a, --no-static	Suppress the printing of profiling information for functions that are not visible outside the file in which they are defined. For example, profiling an application that is partially composed of a source file that contains definitions of the functions foo() and bar(), where bar() is only called by foo(), will only report calls to the function foo() in profiling output. Any time spent in these “local” functions will be added to the profiling output for the function(s) that call them in gprof’s flat profile and call graph output.

Table 6-2. Command-Line Options for the gprof Program (continued)

Option	Description
<code>-A[<i>symbol-specification</i>], --annotated-source [=<i>symbol-specification</i>]</code>	Cause gprof to print annotated source code. If a symbol specification is provided, annotated source output is only generated for functions that match that specification. In order to print annotated source code, you must have used the <code>-g</code> option when compiling your application, so that mandatory symbol information is present in the profiled binary.
<code>-b, --brief</code>	Cause gprof not to print the explanations of the flat profile and call graph. Though useful, once you are familiar with this information, it is helpful to be able to suppress it in your output.
<code>-c, --static-call-graph</code>	Cause gprof to provide extra analysis of your application's call graph, listing functions that were present in the source code but were never actually called. Symbol table entries for these functions must be present—for example, if you link with an object file that was not compiled with profiling support, any functions that are present in that object file but not called by your application will not be displayed unless that object file was compiled with <code>-g</code> and/or <code>-pg</code> . Using these options provides a quick and easy way to identify <i>dead code</i> , which is code that is never used and which you may be able to remove.
<code>-C[<i>symbol-specification</i>], --exec-counts [=<i>symbol-specification</i>]</code>	Print summary information about the functions in your sample application and the number of times that they were called. Providing a symbol specification restricts the summary information to functions matching the symbol specification.
<code>-d[<i>num</i>], --debug[=]<i>num</i></code>	Specify debugging options. If no debug level is specified, these options display all debugging information. Debugging options are rarely used unless you suspect a problem with gprof, in which case the maintainer may request that you provide output using a specific debugging level.

Table 6-2. Command-Line Options for the gprof Program (continued)

Option	Description
--demangle[=style], --no-demangle	(Used only with C++ applications) Specify whether the symbol names in compiled C++ code will be decoded from simple low-level names into user-level names, including class information, enabling gprof to differentiate between them in its output. The optional style argument enables you to specify a demangling style, based on your compiler. Possible styles are auto, which causes gprof to automatically select a demangling style based on examining your program's source code; gnu, the default value for code compiled using gcc and g++; lucid, which specifies that gprof demangle symbol names based on the algorithm used by the Lucid C++ compiler's name encoding algorithm; and arm, which results in gprof demangling function names using the algorithm specified in <i>The Annotated C++ Reference Manual</i> , Margaret A. Ellis and Bjarne Stroustrup (Addison-Wesley, 1990. ISBN: 0-201-51459-1.).
-D, --ignore-non-functions	Ignore symbols that are not known to be functions. These options are only supported on certain systems, such as Solaris and HP-UX systems.
--file-ordering <i>map_file</i>	Prints a suggested order in which object files in the application should be linked. This ordering is based on profiling data that shows which functions call which other functions, and can improve paging and cache behavior in the compiled application. This option is typically used with the -a option in order to suppress linking information about external functions that are contained in libraries and whose link order you therefore have no control over.
--function-ordering	Prints a suggested order in which functions in the program should be organized to improve paging and cache behavior in the compiled application.
-i, --file-info	Print summary information about the number of histogram records, the number of records in the call graph, and the number of basic block count records in the gmon.out file, and exits.
-I <i>dirs</i> , --directory-path= <i>dirs</i>	Specify a colon-separated list of directories that should be searched for source files related to the application.

Table 6-2. Command-Line Options for the gprof Program (continued)

Option	Description
<code>-J[<i>symbol-specification</i>], --no-annotated-source [=<i>symbol-specification</i>]</code>	Suppress printing of annotated source code. If a symbol specification is provided, an annotated source code listing is printed that does not include symbols that match the specification. In order to print annotated source code, you must have used the <code>-g</code> option when compiling your application, so that mandatory symbol information is present in the profiled binary.
<code>-k <i>from/to</i></code>	Deletes specified arcs from the call graph. Arcs are specified as <i>from/to</i> pairs, where <i>from</i> is the function calling a second function, and <i>to</i> is the function being called.
<code>-l, --line</code>	Enable line-by-line profiling, which causes histogram and call graph profiling information to be associated with specific lines in the application's source code. Line-by-line profiling is much slower than basic block profiling, and can magnify statistical inaccuracies in gprof output.
<code>-L, --print-path</code>	Display the full pathname of source files when doing line-by-line profiling.
<code>-m <i>num</i>, --min-count=<i>num</i></code>	Suppress information about symbols that were executed less than <i>num</i> times. These options are only meaningful in execution count output.
<code>-n[<i>symbol-specification</i>], --time [=<i>symbol-specification</i>]</code>	Restrict the symbols whose values are propagated in gprof's call graph analysis to those that match the symbol specification.
<code>-N[<i>symbol-specification</i>], --no-time [=<i>symbol-specification</i>]</code>	The inverse of the <code>-n</code> option, these cause gprof not to propagate call graph data for symbols matching the symbol specification.
<code>-O<i>name</i>, --file-format=<i>name</i></code>	Specify the format of the gmon.out data file. Valid values are auto (the default), bsd, 4.4bsd, and magic.
<code>-p[<i>symbol-specification</i>], --flat-profile [=<i>symbol-specification</i>]</code>	Display flat profile information. If a symbol specification is specified, gprof only prints flat profile information for symbols matching that symbol specification.
<code>-P[<i>symbol-specification</i>], --no-flat-profile [=<i>symbol-specification</i>]</code>	Suppress printing flat profile information. If a symbol specification is provided, gprof prints a flat profile that excludes matching symbols.

Table 6-2. Command-Line Options for the gprof Program (continued)

Option	Description
<code>-q[<i>symbol-specification</i>], --graph [=<i>symbol-specification</i>]</code>	Display call graph analysis information. If a symbol specification is provided, gprof only prints call graph information for symbols matching that symbol specification and for children of those symbols.
<code>-Q[<i>symbol-specification</i>], --no-graph [=<i>symbol-specification</i>]</code>	Suppress printing call graph analysis. If a symbol specification is provided, gprof prints a call graph that excludes symbols matching that symbol specification.
<code>-s, --sum</code>	Summarizes profile data information and writes it to the file gmon.sum. By using the <code>-s</code> option and specifying gmon.sum as a profiling data file on the gprof command line, you can merge the data from multiple profile data files into a single summary file and then use a gprof command line such as <code>gprof executable gmon.sum</code> to display the summary information.
<code>-T, --traditional</code>	Display output in traditional BSD style.
<code>-v, --version</code>	Display the version number of gprof and then exit without processing any data.
<code>-w <i>width</i>, --width=<i>width</i></code>	Set the width of the lines in the call graph function index to <i>width</i> .
<code>-x, --all-lines</code>	Annotate each line in annotated source output with the annotation for the first line in the basic block to which it belongs. In order to print annotated source code, you must have used the <code>-g</code> option when compiling your application, so that mandatory symbol information is present in the profiled binary.
<code>-y, --separate-files</code>	Cause annotated source output to be written to files with the same name as the source file, appending “-ann” to the filename. In order to print annotated source code, you must have used the <code>-g</code> option when compiling your application, so that mandatory symbol information is present in the profiled binary.
<code>-z, --display-unused-functions</code>	List all functions in the flat profile, even those that were never called. You can use these options with the <code>-c</code> option to identify functions that were never called and can potentially therefore be eliminated from the source code for your application.

Table 6-2. Command-Line Options for the gprof Program (continued)

Option	Description
<code>-Z[symbol-specification]</code> ,	Suppress printing summary count information about
<code>--no-exec-counts</code>	all of the functions in your application and the
<code>[=symbol-specification]</code>	number of times that they were called. If a symbol specification is provided, summary count information is printed, but excludes symbols matching the symbol specification.

NOTE In addition to these command-line options, you can also affect the behavior of gprof by setting the GPROF_PATH environment variable. If used, this environment variable should be set to a list of directories in which gprof should search for application source code files if you are using gprof options that produce annotated source code listings.

A Sample gprof Session

This section walks through an example of using gprof to provide profiling information for the same small application used in the gcov discussion of this chapter. This program calculates a specified number of values in the Fibonacci sequence. The sample source code for this application is shown in Listings 6-1 and 6-2, earlier in this chapter.

Before running gcc or gprof, the contents of the directory containing the source code for the sample application are as follows:

```
$ ls
calc_fib.c fibonacci.c Makefile
```

First, compile the application using gcc's -pg option to integrate profiling information in the binary:

```
$ gcc -pgfibonacci.c calc_fib.c -o fibonacci
```

After compilation completes, the contents of the sample application directory are as follows:

```
$ ls
calc_fib.c fibonacci fibonacci.c Makefile
```

Next, you run the sample application, specifying the command-line argument 11 to generate the first 11 numbers in the Fibonacci sequence:

```
$ ./fibonacci 11
0 1 1 2 3 5 8 13 21 34 55
```

After running the application, you get the following as the contents of the sample application's source directory:

```
$ ls
calc_fib.c fibonacci fibonacci.c gmon.out Makefile
```

Running the application creates the gmon.out file, which contains profiling information based on the run of the application. You can now run gprof to produce information about the behavior and performance of the application.

Running gprof in its simplest form, gprof fibonacci (gprof fibonacci.exe for Cygwin users) produces the output shown in Listing 6-6. The default output of gprof is somewhat verbose, but contains a good deal of useful information about interpreting the output you receive.

Listing 6-6. Default Output from gprof

Flat profile:

Each sample counts as 0.01 seconds.

no time accumulated

%	cumulative	self	calls	self	total	
time	seconds	seconds		Ts/call	Ts/call	name
0.00	0.00	0.00	11	0.00	0.00	calc_fib

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self ms/call the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call the average number of milliseconds spent in this function and its descendants per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) no time propagated

	index	% time	self	children	called	name
[1]					442	calc_fib [1]
			0.00	0.00	11/11	main [8]
			0.0	0.00	11+442	calc_fib [1]
					442	calc_fib [1]

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so it is easier to look up where the function in the table.

% time This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children	This is the total amount of time propagated into this function by its children.
called	This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.
name	The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self	This is the amount of time that was propagated directly from the function into this parent.
children	This is the amount of time that was propagated from the function's children into this parent.
called	This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.
name	This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self	This is the amount of time that was propagated directly from the child into the function.
children	This is the amount of time that was propagated from the child's children to the function.

called	This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.
name	This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

```
[1] calc_fib
```

The default output of gprof provides several types of information:

- *Flat profile information:* Shows the number of times each function was called and the amount of time spent in each call. In the case of the sample application, this shows that the `calc_fib()` routine was called 11 times from the Fibonacci program's main routine, and that it ran so quickly that it appears to have taken no time. This is because profiling measurements are expressed in milliseconds but summaries are show in 1/10 seconds—the program did not require enough execution time to round up to 1 millisecond.
- *Call graph information:* Shows how often each function was called and the functions from which it was called. This shows that the `calc_fib()` routine was called 11 times from the program's main routine, but that the `calc_fib()` routine was called by itself 442 times (since it is a recursive routine).
- *A function index:* Summarizes all of the functions in the application, indexed to make it easy to correlate this to the flat profile and call graph output.

You can display just the summary information by executing `gprof` with the `-b` (brief) option, as in the following example:

```
$ gprof -b fibonacci
Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated

% cumulative self self total
time seconds seconds calls Ts/call Ts/call name
0.00 0.00 0.00 11 0.00 0.00 calc_fib
```

Call graph

```
granularity: each sample hit covers 4 byte(s) no time propagated

index % time self children called name
[1] 0.0 0.00 0.00 442 calc_fib [1]
          0.00 0.00 11/11 main [8]
          11+442 calc_fib [1]
          442 calc_fib [1]
```

Index by function name

```
[1] calc_fib
```

This example provides the basic profiling information without the explanation of any of the fields that are part of gprof's default output.

Displaying Annotated Source Code for Your Applications

As shown in the previous section, gprof has a tremendous number of options, not all of which are necessary to show here. However, one of gprof's most useful options is the **-A** option, which displays annotated source code for the functions in the application, marked up with profiling information. In order to display annotated source code from gprof, complete symbol information must be present in the binary, so the program must be compiled using **gcc** with both the profiling (**-pg**) and symbol debugging (**-g**) switches, as in the following example:

```
$ gcc -pg -g calc_fib.c fibonacci.c -o fibonacci
```

You can now run gprof with the **-A** option to display annotated source code for the only function in the application, `calc_fib()`, as shown in the following example:

```
$ gprof -A fibonacci

*** File /home/wvh/writing/gcc/src/calc_fib.c:
/*
 * Function that actually does the Fibonacci calculation.
 */

11 -> int calc_fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else
        return((calc_fib(n-2) + calc_fib(n-1)));
}
```

Top 10 Lines:

Line	Count
5	11

Execution Summary:

1	Executable lines in this file
1	Lines executed
100.00	Percent of the file executed
11	Total number of line executions
11.00	Average executions per line

If you want to see the complete, annotated source code with profiling information for a simple application, you can combine it into a single source module, as shown in Listing 6-7.

Listing 6-7. Single Source Module for the Sample Application

```
/*
 * Simple program to print a certain number of values
 * in the Fibonacci sequence.
*/

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

```

static int calc_fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else
        return((calc_fib(n-2) + calc_fib(n-1)));
}

int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
        n = atoi(argv[1]);
    else {
        printf("Usage: fibonacci num-of-sequence-values-to-print\n");
        exit(-1);
    }
    for (i=0; i < n; i++)
        printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}

```

Compiling this module and then using the `gprof -A fibonacci` command to display the annotated source code listing will present the entire listing, as shown in Listing 6-8.

Listing 6-8. Annotated Source Code Listing of the Single Source Module

```

*** File /home/wvh/writing/gcc/src/fibonacci_all.c:
/*
 * Simple program to print a certain number of values
 * in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

```

```

11 -> static int calc_fib(int n) {
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else
            return((calc_fib(n-2) + calc_fib(n-1)));
    }

##### -> int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
        n = atoi(argv[1]);
    else {
        printf("Usage: fibonacci num-of-sequence-values-to-print\n");
        exit(-1);
    }
    for (i=0; i < n; i++)
        printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}

```

Top 10 Lines:

Line	Count
9	11

Execution Summary:

```

2 Executable lines in this file
2 Lines executed
100.00 Percent of the file executed

11 Total number of line executions
5.50 Average executions per line

```

NOTE If you compile your source code without using the `-g` option, trying to display annotated source code output using `gprof` will return a message like the following on a Linux system:

```
$ gprof -A fibonacci
gprof: could not locate '/home/wvh/src/glibc-2.2.5/csu/gmon-start.c'
```

This message is somewhat misleading, as it provides the location of the source code for the entry point for gcc's monitoring routines, based on the location of this source file when gcc was compiled on your system. To eliminate this message and display annotated source code, compile your source code with the `-g` option.

In conjunction with the profiling options provided by gcc, the `gprof` application can give you substantial insights into the execution of your application and the amount of time spent in each of its routines. The `gprof` application can be quite useful as a guide to optimization, but cannot directly help with debugging. However, gcc has one more profiling card up its sleeve—the ability to automatically insert user-defined code at the entry and exit points of every function in your application via the `-finstrument-functions` compilation option. As explained in the next section, this gcc option can provide the link between profiling and debugging that you may find useful when testing and optimizing your applications.

Adding Your Own Profiling Code Using gcc

The gcc compiler's `-finstrument-functions` option automatically inserts a call to two profiling functions that will automatically be called just after entering each function and just before returning from each function. A call to the `_cyg_profile_func_enter()` function is inserted after entering each function, and a call to the `_cyg_profile_func_exit()` function is made just before returning from each function. The prototypes for these functions are shown here:

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

The parameters to these functions are the address of the current function and address from which it was called, as provided in the application's symbol table.

Mapping Addresses to Function Names

You can display the values of these parameters from within these functions using the standard `printf` or `fprintf` `%p` format string. You can then use the GNU `addr2line` function (part of the `binutils` package, discussed in the section “Obtaining and Compiling `gprof`”) to translate the address to the line in a specified source file from which it was called, as in the following example:

```
$ addr2line -e fibonacci3inst 0x80484b4  
/home/wvh/writing/gcc/src/fibonacci.c:22
```

The `addr2line` program’s `-e` option specifies the name of the executable that contains the specified address. The output from the `addr2line` command shows the source file and the number of the line of source code in that file that maps to the specified address.

In order to use the `addr2line` function, you must have preserved symbolic debugging information in the application by compiling it with the `-g` option. If you forget to use the `-g` option when compiling, the `addr2line` function displays a message like the following:

```
$ addr2line -e fibonacci3inst 0x80484b4  
/home/wvh/src/glibc-2.2.5/csu/init.c:0
```

Because this executable does not contain symbolic debugging information, the `addr2line` function cannot find the specified address, and therefore returns a pointer to one of the GNU C library’s initialization routines. This is the same sort of problem discussed in the previous section when trying to use `gprof` to display annotated source code for applications that were not compiled with symbolic debugging information.

Simply defining the `_cyg_profile_func_enter()` and `_cyg_profile_func_exit()` functions in one of the existing source files for your application will enable you to add your own code to these functions in order to extract additional profiling or debugging data beyond that which is automatically provided by `gprof`. Adding calls to these functions obviously slows the performance of your application, but should do so consistently in all instrumented functions. It is also simple enough to set up your `Makefile` so that you can easily compile your application without the `-finstrument-functions` options whenever you want to measure pure performance.

Common Profiling Errors

The most common error made when implementing these profiling functions is to simply add them to one of the source files you are compiling with the

`-finstrument-functions` option. By default, this will cause these functions themselves to be instrumented, which will cause your application to crash, because the `__cyg_profile_func_enter()` function will recursively call itself until a stack overflow occurs. You can eliminate this problem in either of two ways:

- Put the code for these two functions in a separate source module that you do not compile with the `-finstrument-functions` option. This is the simplest approach, but requires special handling of this source file in your Makefile, which is easy to forget or overlook.
- Use the `no_instrument_function` attribute on the entry and exit functions to specify that these functions should not be instrumented.

Listing 6-9 shows the complete source code for the sample `fibonacci.c` application used throughout this section, with sample code for these functions inserted that shows the syntax required to set the `no_instrument_function` attribute.

Listing 6-9. Sample Application Showing Manual Profiling Calls

```
/*
 * Simple program to print a certain number of values
 * in the Fibonacci sequence.
 */

#include <stdio.h>
#include <stdlib.h>

void __attribute__((__no_instrument_function__))
    __cyg_profile_func_enter(void *this_fn, void *call_site)
{
    printf(" Entered function %p, called from %p\n", this_fn, call_site);
}

void __attribute__((__no_instrument_function__))
    __cyg_profile_func_exit(void *this_fn, void *call_site)
{
    printf("Exiting function %p, called from %p\n", this_fn, call_site);
}

static int calc_fib(int n) {
    if (n == 0) {
        return 0;
    }
    else {
        return calc_fib(n - 1) + calc_fib(n - 2);
    }
}
```

```

} else if (n == 1) {
    return 1;
} else
    return((calc_fib(n-2) + calc_fib(n-1)));
}

int main(int argc, char *argv[]) {
    int i,n;

    if (argc == 2)
        n = atoi(argv[1]);
    else {
        printf("Usage: fibonacci num-of-sequence-values-to-print\n");
        exit(-1);
    }
    for (i=0; i < n; i++)
        printf("%d ", calc_fib(i));
    printf("\n");
    return(0);
}

```

Some sample output from a run of this application, compiled with the `-finstrument-functions` option, is shown here:

```

Entered function 0x804852a, called from 0x40052657
Entered function 0x80484b4, called from 0x804859d
Exiting function 0x80484b4, called from 0x804859d
0 Entered function 0x80484b4, called from 0x804859d
Exiting function 0x80484b4, called from 0x804859d
1 Entered function 0x80484b4, called from 0x804859d
Entered function 0x80484b4, called from 0x80484fa
Exiting function 0x80484b4, called from 0x80484fa
Entered function 0x80484b4, called from 0x8048508
...

```

When combined with the compilation options required for test coverage analysis and profiling, the `-finstrument-functions` option is a final, powerful arrow in your debugging, analysis, and profiling quiver. Calling user-specified profiling code can save you time in the debugger, and can also make it easy to integrate applications compiled with gcc with any third-party or in-house profiling capabilities that you may already be using to fine-tune or measure application performance.

CHAPTER 7

Using Autoconf and Automake

MUCH OF THE APPEAL OF UNIX, beyond simply being a robust, high-powered multiuser operating system written in a powerful and popular programming language, lies in the promise of portability. In reality, as soon as the split between the AT&T and BSD variants of Unix occurred, portability became more of a possible goal than an actual reality.

One way of resolving portability issues is to use platform-specific definitions throughout your code, leading to source code that may contain as many conditionals (`#ifdef YOUR-UNIX-VARIANT`) as actual lines of source code. Though often unreadable, this is a workable solution, except for the fact that it requires the maintainer of a portable application to be aware of the nuances of every possible system on which people might want to use that application.

A more workable solution is to test dynamically the characteristics of both your hardware and operating system and generate appropriate Makefiles that are customized for your platform. The applications discussed in this chapter, Autoconf and Automake, do exactly that. Autoconf is an application that generates shell scripts that automatically configure source code packages. Automake is a companion application that can automatically generate Makefiles that are compliant with the GNU Coding Standards.

This chapter explains the division of labor between Autoconf and Automake and surveys the applications that they use in order to simplify the configuration process. Subsequent sections of this chapter then demonstrate how to install and configure Automake and Autoconf themselves, and discuss how to use Autoconf and Automake to make it easy to distribute application source code so that it can be painlessly configured and compiled on a wide variety of Unix, Linux, and other Unix-like operating systems.

Unix Software Configuration, Autoconf, and Automake

As mentioned in the introduction to this chapter, one of the original selling points of Unix was that Unix applications were easily ported from one Unix system to another. This was largely due to the fact that they were written in the C programming language, which was born on Unix systems and is still the default programming language of all Unix and Unix-like systems.

However, simply using the same programming language does not guarantee as much as one might hope. As Unix became more widely used, differences between Unix implementations began to be more pronounced, especially between the AT&T-base SYSIII and SYSV flavors of Unix and academically inspired versions of Unix, most of which were based on one version or another of the Berkeley Standard Distribution, commonly known as BSD. Variations in underlying subsystems, such as networking, were one of the major differences between the two, but other differences became more pronounced and more widespread as time passed. By the mid-1980s, versions of Unix such as AIX, AOS, DG/UX, HP-UX, IRIX, SunOS, SYSV, Ultrix, USG, VENIX, and XENIX were among the many flavors of Unix that existed.

Unfortunately, many of these Unix variants used different application programming interfaces (APIs) and different libraries of related functions ranging from basic data manipulation such as sorting, all the way to system-level APIs for networking, file and directory manipulation, and so on. A common approach to portability in C source code is to use conditional compilation via `#ifdef` symbols, which use system-defined symbols that identify the operating system version and platform on which the application is being compiled. Although conditionals are necessary and still used today, including conditionals for each of the variety of Unix systems that are available could make the source code for even the simplest C application unreadable.

Conditionalizing code for operating system-dependent features is a pain because many Unix variants share the same sets of features and libraries, leading to a significant amount of duplication within `#ifdefs`. Portable Unix applications therefore began to move toward `#ifdefs` that were based on sets of features rather than specific operating system versions whenever possible. Rather than system-specific statements like `#ifdef AIX`, an `#ifdef` such as `#ifdef HAS_BCOPY` could be used to differentiate between classes of Unix and Unix-like systems.

Feature-based portability and the generally increasing complexity of setting `#ifdefs` and maintaining applications that were portable across multiple types of Unix systems cried out for a programmatic solution. One of the earliest attempts along these lines was the `metaconfig` program by Larry Wall, Harlan Stenn, and Raphael Manfredi, which produces interactive shell scripts named `Configure` that are still used to configure applications such as Perl. Other significant attempts at cross-platform, single-source configuration were the `configure` scripts used by Cygnus Solutions (a major open source/GNU pioneer that was later acquired by Red Hat) and `gcc`, and David MacKenzie's `autoconf` program.

NOTE *The early Configure scripts produced by the metaconfig program were not only useful, but also entertaining. While testing the parameters and configuration of your system, they also offered some humorous insights such as my personal favorite:*

Congratulations! You're not running Eunice...

Eunice was a Unix emulation layer that ran on VMS systems—though useful because it opened up VMS systems to the world of freely available Usenet applications and enabled Unix users to get real work done on VMS systems almost immediately, the whole notion was somewhat unholy.

In a somewhat rare gesture of application solidarity in the Unix environment, David MacKenzie's autoconf program was the eventual winner of the Unix/Linux configuration sweepstakes because it provided a superset of the capabilities of its brethren, with the exception of the interactive capabilities of the metaconfig program.

When he began writing autoconf, MacKenzie was responsible for maintaining many of the GNU utilities for the Free Software Foundation (FSF), and needed a standard way to configure them for compilation on the variety of platforms on which they were used. The vast number of GNU utilities and possible platforms was a strong motivation to write a powerful and flexible configuration utility.

The autoconf program provides an easily extended set of macros that are written for use with the Unix m4 macro processor. Macros are single statements that are subsequently expanded into many other statements by a macro processor. The m4 macro processor is the most common macro processor used on Unix systems, but most programming languages also provide one, such as the multistage macro preprocessor named cpp that is used with the C programming language.

The autoconf program invokes the m4 macro processor to generate a shell script named configure that a user then executes to customize and configure application source code for a particular computer system. The autoconf program uses a single input file, configure.ac, (formerly known as configure.in), which contains specific macro statements that provide information about an application and the other applications and utilities required to compile it. Complex applications that use autoconf may have multiple configure.ac files in various subdirectories, each of which defines the configuration requirements for the library or distinct functionality provided by the source code located in that directory.

Unfortunately, identifying the location and names of relevant include files, supporting libraries, and related applications located on a specific system is only half of the battle of successfully compiling an application on a particular platform. Since the dawn of Unix time, most Unix applications have been compiled using a rule-driven application called `make`, which invokes compilers and other utilities actually to produce an executable from all the components of a program's source code. The `make` program does the following:

- Enables developers to identify relationships and dependencies between the source modules, include files, and libraries that are required for successful compilation
- Specifies the sequence in which things must be compiled in order to build an application successfully
- Avoids recompilation by limiting compilation to the portions of an application that are affected by any change to source code or statically linked libraries

The `make` program was originally written by Stu Feldman at AT&T for an early version of Unix. As a part of AT&T Unix, the source code for the `make` program was not freely available, and has therefore largely been replaced by the GNU `make` program. GNU `make` provides all of the features of the original `make` program and a host of others. Other versions of the `make` program are still active, such as the `bmake` program used by most modern BSD-inspired versions of Unix (for example, FreeBSD and NetBSD), and the `imake` program, used by the X Window System project; but GNU `make` is probably the most common version of the program used today. Subsequent references to the `make` program throughout this chapter therefore refer to GNU `make`.

The rules used by any `make` program are stored in text files named `Makefile`, which contain dependency rules and the commands necessary to satisfy them. As you might expect, the `Makefiles` for complex applications are themselves extremely complex as well as platform-specific, since they need to invoke platform-specific libraries and utilities that may be located in different directories or have different names on different systems.

To simplify creating `Makefiles`, David MacKenzie also developed the `automake` program, which uses a simple description of the build process that is employed to generate `Makefiles`. The `automake` program was quickly rewritten in Perl by Tom Tromey, who still maintains and enhances it. The `automake` program generates a `Makefile` for an application from an application-specific input file named `makefile.am`. A `makefile.am` file contains sets of `make` macros and rules that are expanded into a file called `makefile.in`, which is then processed by the `autoconf` program to produce a `Makefile` that is tailored for a specific system.

Like the `autoconf` program's `configure.ac` files, a complex application that uses `automake` may have multiple `makefile.am` files in various subdirectories and

reflect specific requirements for the Makefile used to compile the library or distinct functionality provided by the source code located in that directory. The Makefiles produced by the automake program are fully compliant with the Free Software Foundation's GNU Makefile conventions, though these files are only readable by experts or people who do not get migraine headaches. For complete information about the GNU Makefile conventions, see http://www.gnu.org/prep/standards_50.html, which is a portion of the larger GNU Coding Standards document at <http://www.gnu.org/prep/standards.html>.

The last component of a truly platform-independent compilation, build, and execution environment is a tool that facilitates linking to libraries of functions, such as the generic C language libraries, that are standard on a given platform and are also external to the application you are building. Linking is fairly simple when static libraries are actually linked to the executable you are compiling, but becomes complex when shared libraries are being used. The GNU Libtool application automatically handles identifying and resolving library names and dependencies when compiling an application that uses shared libraries. We will cross that bridge when we come to it—Chapter 8 explains using this application during the compilation and build process.

The remainder of this chapter shows you how to compile and install autoconf and automake, and how to create the configuration files required to use them to create portable, flexible distributions of your applications that can quickly and easily be compiled on a variety of platforms.

Installing and Configuring Autoconf and Automake

As you might hope, utilities such as autoconf and automake take advantage of their own capabilities in order to simplify setup. This section explains where to obtain the latest versions of autoconf and automake, and how to configure, compile, and install them.

Deciding Whether to Upgrade or Replace Autoconf and Automake

The default location for installing autoconf and automake as described in this chapter is /usr/local/bin. This will leave any existing versions of autoconf, automake, and related applications on your system. In order to make sure that you execute the new versions rather than any existing ones, you will have to verify that the /usr/local/bin directory appears in your path before the directory in which the default version of autoconf is located (usually /usr/bin).

Both autoconf and automake are configured using configure scripts. You can use the configure script's --prefix option to change the base location for all of the files produced when compiling either package. For example, executing the

configure script with the following command will configure the autoconf or automake Makefile to install binaries in /usr/bin, architecture-independent data such as info files in /usr/share, and so on:

```
$ ./configure --prefix=/usr
```

For information on other options for the configure script, see Table 7-6, which appears in the section “Running configure Scripts” later in this chapter.

NOTE *You may want to execute the command which autoconf in order to determine which version of autoconf your system will use; if you want to use versions that you have already installed in /usr/local/bin, adjust the prefix accordingly.*

If you are using an RPM-based system, overwriting existing binaries and associated files will cause problems if you subsequently upgrade your system or an affected package. You can always use the `rpm` command’s `--force` option to force removal of packages whose binaries have been overwritten.

In general, simply modifying your path to make sure that you are executing the “right” version of an application is simpler than overwriting existing binaries, makes it easier for you to fall back to the “official” version if you discover incompatibilities in a new version, and simplifies system upgrades of package-based Linux systems. The downside of this approach is that you have to make sure every user of the system updates their path so that they execute the updated version of an application.

The examples in this section configure, build, test, and install the autoconf and automake binaries in their default locations, the appropriate subdirectories of /usr/local.

Building and Installing Autoconf

Autoconf comes preinstalled with most Linux distributions for which you have installed a software development environment for the C and C++ programming languages. However, if you are writing software that you want to distribute and you want to be able to take advantage of the latest fixes and enhancements to autoconf, it can never hurt to get the latest version of autoconf. Autoconf is a relatively stand-alone package that you can safely upgrade without risking the introduction of any incompatibilities on your system. The home page for the autoconf utility is at <http://www.gnu.org/software/autoconf>.

Autoconf depends on the GNU m4 macro preprocessor, which is installed by default on most Linux systems. The home page for GNU m4 is at <http://www.gnu.org/software/m4/>, though its primary development site, from which you can download the latest version, is at <http://savannah.gnu.org/projects/m4/>. The current version of m4 at the time of this writing is 1.4.1.

If the Perl interpreter is installed on your system when you are configuring, building, and installing autoconf from source code, the configure script will configure the Makefile so that a number of Perl scripts associated with autoconf are configured for your system. These auxiliary utilities are then installed as part of the `make install` command described later in this section. A complete list of the scripts and auxiliary utilities that can be installed as part of the autoconf package is shown in Table 7-1 later in this chapter.

You can obtain the latest version of autoconf from the download page (<http://ftp.gnu.org/gnu/autoconf>). At the time of this writing, the latest version of autoconf is 2.56—this is the version that we discuss in this chapter.

Mailing Lists for Autoconf

If you are very interested in Autoconf, want to report a bug, want to discuss its use, or have suggestions for future improvements to Autoconf, the Free Software Foundation hosts several Autoconf-related mailing lists through its gnu.org site:

- The `autoconf` list is a forum for asking questions, answering them, discussing usage, or suggesting and discussing new features for Autoconf. You can subscribe to or unsubscribe from this list at <http://mail.gnu.org/mailman/listinfo/autoconf/>.
 - The `autoconf-patches` mailing list contains the latest patches to Autoconf and is where you should submit your changes if you have found and fixed a bug or implemented a new feature in Autoconf. You can subscribe to or unsubscribe from this list at <http://mail.gnu.org/mailman/listinfo/autoconf-patches/>.
 - The `commit-autoconf` list contains commit messages from check-ins or other changes to the Autoconf source repository. You can subscribe to or unsubscribe from this list at <http://mail.gnu.org/mailman/listinfo/commit-autoconf>.
-

You can download a gzipped source file for the latest version of autoconf using your favorite browser or through a faster command-line tool such as wget. For example, you can download the archive containing the latest autoconf source using the command

```
$ wget http://ftp.gnu.org/gnu/autoconf/autoconf-2.56.tar.gz
```

Once you have downloaded the gzipped source archive, you can extract its contents using a standard tar command such as the following:

```
$ tar zxvf autoconf-2.56.tar.gz
```

This command creates the directory autoconf-2.56. To begin building autoconf, change to this directory and execute the configure script. In a textbook example of bootstrapping, the configure script was produced by autoconf when the maintainers packaged up this latest version using autoconf.

The following is an example of the output from the configure script when run on a sample Linux system:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for expr... /usr/bin/expr
checking for gm4... no
checking for gnum4... no
checking for m4... /usr/bin/m4
checking whether m4 supports frozen files... yes
checking for perl... /usr/bin/perl
checking for emacs... emacs
checking for emacs... (cached) emacs
checking where .elc files should go... ${datadir}/emacs/site-lisp
configure: creating ./config.status
bconfig.status: creating config/Makefile
config.status: creating tests/Makefile
config.status: creating tests/atlocal
config.status: creating tests/autoconf
config.status: creating tests/autoheader
config.status: creating tests/autom4te
config.status: creating tests/autoreconf
config.status: creating tests/autoscan
config.status: creating tests/autoupdate
```

```
config.status: creating tests/ifnames
config.status: creating man/Makefile
config.status: creating lib/emacs/Makefile
config.status: creating Makefile
config.status: creating doc/Makefile
config.status: creating lib/Makefile
config.status: creating lib/Autoconf/Makefile
config.status: creating lib/autoscan/Makefile
config.status: creating lib/m4sugar/Makefile
config.status: creating lib/autoconf/Makefile
config.status: creating lib/autotest/Makefile
config.status: creating bin/Makefile
config.status: executing tests/atconfig commands
```

Once you have configured the autoconf source code for your system, you can build autoconf by simply executing the `make` command in the directory `autoconf-2.56`.

```
$ make
[long, verbose output not shown]
```

Once autoconf has compiled successfully, you should execute the `make check` command to execute the tests provided with autoconf. These have the side effect of ensuring that the version of `m4` provided on your system is fully compatible with the requirements of the version of autoconf that you have just compiled:

```
$ make check
[intermediate verbose output not shown]
## -----
## GNU Autoconf 2.56 test suite.
## -----
Executables (autoheader, autoupdate...).
 1: tools.at:47      ok
 2: tools.at:92      ok
 3: tools.at:118     ok
 4: tools.at:155     ok
 5: tools.at:244     ok
 6: tools.at:275     ok
 7: tools.at:303     ok
 8: tools.at:349     ok
 9: tools.at:397     ok
10: tools.at:529     ok
11: tools.at:561     ok
12: tools.at:588     ok
```

```
M4sugar.
13: m4sugar.at:35      ok
14: m4sugar.at:77      ok
15: m4sugar.at:115     ok
[additional verbose output not shown]
## -----
## All 174 tests were successful. ##
## -----
[additional verbose output not shown]
```

As you can see from this sample output, autoconf provides a fairly exhaustive set of tests to verify its functionality. If your new version of autoconf passes all the tests, you can now proceed to install it by becoming the root user and using the standard `make install` command:

```
$ su root
Password: root-password
# make install
```

As discussed previously in the section “Deciding Whether to Upgrade or Replace Autoconf and Automake,” the autoconf package is configured to install binaries and supporting files into subdirectories of the directory `/usr/local`. For information about changing the location where autoconf installs and the pros and cons of doing so, see that section of this chapter.

If the Perl interpreter was found on your system when you configured Autoconf, the autoconf package builds and installs several related utilities. Table 7-1 lists these applications and explains what each is used for. Using these applications is discussed later in the section “Configuring Software with Autoconf and Automake.”

Table 7-1. Applications in the Autoconf Package

Utility	Description
autoconf	The shell script that invokes <code>m4</code> to generate the <code>configure</code> script from the description of your application’s configuration in the <code>configure.ac</code> file.
autoheader	A Perl script that parses the source files associated with your application and extracts information about the header files used by the application. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.

Table 7-1. Applications in the Autoconf Package (continued)

Utility	Description
autom4te	A Perl script that provides a wrapper for configuring the m4 environment, loading m4 libraries, and so on. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autoreconf	A Perl script that generates new distribution files if the files in the GNU build system have been updated. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autoscan	A Perl script that generates an initial configure.ac for you by scanning the contents of an application source directory and making a best guess about dependencies and compilation requirements. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
autoupdate	A Perl script used to update an autoconf configuration file (typically configure.ac) to reflect any changes in autoconf macro syntax or macro names. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.
ifnames	A Perl script that parses C source files and extracts a list of any conditionals (#if, #elif, #ifdef, and #ifndef) used in those files. This script is only generated and installed if Perl is found on your system when you configure the autoconf package.

You are now ready to use the new version of autoconf, as described in the section “Configuring Software with Autoconf and Automake,” or to build automake, as described in the next section.

Obtaining and Installing Automake

Like Autoconf, Automake is preinstalled with most Linux distributions in which you have installed a software development environment for the C and C++ programming languages. However, if you are writing software that you want to distribute and you want to be able to take advantage of the latest fixes and enhancements to Automake, it can never hurt to get the latest version. Automake is pretty much a stand-alone package that you can safely upgrade without risking your system. Automake has a standard page at the GNU site (<http://www.gnu.org/software/automake>), but its actual home page is at the Red Hat site (<http://sources.redhat.com/automake>), where Tom Tromey, the author of the Perl version and its maintainer, works. You can get the latest, most cutting-edge versions of Automake there.

NOTE Automake depends on the existence of a recent version of the Perl interpreter on your system. It is unlikely that you will not have Perl installed on a system you are using for development, since many development-related utilities are written in Perl. However, if you need or want to get a version of Perl to install on your system, you can download source code or binary versions of Perl from the official Perl home page at <http://www.perl.com>. The examples in this chapter were tested using Perl 5.6, which is the major version provided with most Linux distributions at the time of this writing. The URL for the download page on the Perl site is <http://www.perl.com/pub/a/language/info/software.html>.

You can obtain the latest version of automake from the download page at the GNU site (<ftp://ftp.gnu.org/gnu/automake>). At the time of this writing, the latest version of automake is 1.7.1—this is the version discussed in this chapter.

Mailing Lists for Automake

If you want to become involved with the Automake community or are interested in learning more about it, both Red Hat and the Free Software Foundation host Automake-related mailing lists:

- The automake-cvs and automake-prs lists provide commit messages from CVS and status messages regarding problem reports tracked in the GNATS database where automake issues are recorded. These lists are both hosted at Red Hat. You can subscribe to either of these lists by using the subscription form located on the Automake page at Red Hat, at <http://sources.redhat.com/automake>.
 - The automake list is a forum for asking questions, answering them, discussing usage, or suggesting and discussing new features for Automake. You can subscribe to or unsubscribe from this list at <http://mail.gnu.org/mailman/listinfo/automake>.
 - The bug-automake lists is the place you should report any bugs that you discover in automake. You can subscribe to or unsubscribe from this list at <http://mail.gnu.org/mailman/listinfo/bug-automake>.
 - The automake-patches mailing list contains the latest patches to Automake, and this is where you should check if you are waiting for a bug fix to the current version of Automake. You can subscribe to or unsubscribe from this list at <http://mail.gnu.org/mailman/listinfo/automake-patches>.
-

You can download the archive containing the latest automake source using the command

```
$ wget ftp://ftp.gnu.org/gnu/automake/automake-1.7.1.tar.gz
```

Once you have downloaded the gzipped source archive, extract the contents using a standard tar command:

```
$ tar zxvf automake-1.7.1.tar.gz
```

This command creates the directory automake-1.7.1. To begin building automake, change directory to this directory and execute the configure script found there.

The following is an example of the output from the configure script when run on a sample Linux system:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets ${MAKE}... yes
checking for perl... /usr/bin/perl
checking for tex... tex
checking whether ln works... yes
checking for egrep... grep -E
checking for fgrep... grep -F
configure: creating ./config.status
config.status: creating Makefile
config.status: creating lib/Automake/Makefile
config.status: creating lib/Makefile
config.status: creating lib/am/Makefile
config.status: creating m4/Makefile
config.status: creating m4/amversion.m4
config.status: creating tests/Makefile
config.status: creating tests/defs
```

Once configuration has completed, you can build an installable version of Automake by simply typing **make** at the command prompt. Because Automake is written in Perl, its build phase is essentially a query-replace operation that inserts the location of the Perl binary on your system into the automake and aclocal scripts that are the two executables produced and installed from the automake package. The speed of Automake's configuration and build steps is more than

made up for by the time required to run the vast number of tests that are provided with Automake.

To run the tests provided with Automake to verify its correctness, execute the command `make check`. The following is some sample output from running the Automake tests:

```
$ make check
[boring output deleted]
PASS: acinclude.test
PASS: aclibobj.test
PASS: aclocal.test
PASS: aclocal2.test
PASS: aclocali.test
PASS: aclocali1.test
PASS: aclocalii.test
PASS: acoutnoq.test
PASS: acoutput.test
PASS: acoutput2.test
PASS: acoutqln.test
PASS: acouttbs.test
PASS: acsilent.test
PASS: acsubst.test
...
```

Automake provides over 430 tests to verify its functionality in a variety of ways. Once these tests have completed, you can install Automake using the standard `make install` command. As with Autoconf, the Automake package is configured to install into subdirectories of the directory `/usr/local`. For information about changing the location where `automake` installs and the pros and cons of doing so, see the section “Deciding Whether to Upgrade or Replace Autoconf and Automake.”

Configuring Software with Autoconf and Automake

Once you have `autoconf`, `automake`, and other utilities from those packages installed on your system, you can now actually use them to configure an application. This section explains how to create the configuration and build description files used by `autoconf` and `automake` to automate the configuration, compilation, installation, and packaging of your application(s).

At a minimum, using `autoconf` and `automake` requires that you create two files: a configuration description file for `autoconf` that contains the basic `m4` macros `autoconf` needs to generate a `configure` script for your application, and a build description file that `automake` uses to generate a `Makefile`, which is then

invoked by the `configure` script to create a `Makefile`. You must also create a few auxiliary files that must exist in order for `automake` to create a packaged distribution of your application conforming to the basic requirements of an application that can be configured using the GNU build tools.

The following sections explain how to create these files and the sequence in which you must execute `autoconf`, `automake`, and the `configure` script itself to generate an appropriate `Makefile` for your system and configuration. For convenience, these sections use the same small Fibonacci application that appears in Chapter 6 to provide a real example of creating configuration files. Listings 6-1 and 6-2 show the C source code for the sample application. To review briefly this application, the file `fibonacci.c` contains the `main` routine for the application, and calls the `calc_fib()` routine that is contained in the `calc_fib.c` source file. This simple application has no external dependencies other than the `stdio.h` and `stdlib.h` include files, and has no platform-specific function or structure declarations.

Creating `configure.ac` Files

There are two standard approaches to creating the configuration file used by `autoconf`:

- Use the `autoscan` utility to generate an initial configuration file that you then fine-tune to satisfy your application's build configuration requirements.
- Create a simple `autoconf` configuration description file and then manually enhance it to reflect any additional configuration requirements that you discover.

The configuration file used by modern versions of the `autoconf` script is `configure.ac`.

NOTE Historically, the name of the configuration description file used by the `autoconf` program has always been `configure.in`. The `.in` suffix in this filename has led to some confusion with other input files used by the `configure` script, such as `config.h.in`, so the "proper" name of the configuration file used by `autoconf` is now `configure.ac`, where the `.ac` suffix indicates that the file is used by `autoconf` rather than its offspring. If you have files named `configure.ac` and `configure.in` in your working directory when you run `autoconf`, the `autoconf` program will process the file `configure.ac` rather than the file `configure.in`.

If you have Perl installed on your system, building and installing the `autoconf` package produces a Perl script called `autoscan`. By default, the `autoscan` script examines the source modules in the directory from which it is executed and generates

a file called `configure.scan`. If a `configure.ac` file is already located in the directory from which `autoscan` is executed, the `autoscan` script reads the `configure.ac` file and will identify any omissions in the file (typically, by exiting with an error message).

As an example, the source directory for the sample Fibonacci application looks like the following:

```
$ ls
calc_fib.c fibonacci.c run_tests
```

Executing the `autoscan` application and relisting the contents of this directory shows the following files:

```
$ autoscan
$ ls
autoscan.log calc_fib.c configure.scan fibonacci.c run_tests
```

If you already have a `configure.ac` file, the `autoscan.log` file contains detailed messages about any advanced macros `autoscan` believes are required by your application but that are not already present in the `configure.ac` file. If you do not already have a `configure.ac` file or there are no apparent configuration problems, the `autoscan.log` file produced by a run of `autoscan` will be empty.

The `configure.scan` file produced by running the `autoscan` program for the sample application looks like the following:

```
# -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.56)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

The `autoconf` macros in the auto-generated file have the meanings defined in Table 7-2.

Table 7-2. Entries in the Sample Autoconf File

Entry	Description
AC_PREREQ	This autoconf file was produced by (and therefore requires) version 2.56 of the autoconf package.
AC_INIT	The basic autoconf initialization macro, which requires three arguments: a user-defined name for the application package being configured by autoconf, the version of the package that is currently being configured, and an e-mail address to which bugs in the application should be reported.
AC_CONFIG_SRCDIR	A macro that takes the name of a source file that must be present in the source directory in order to configure and compile the application. This macro is essentially just a sanity check for autoconf. Its value is typically the name of the file containing the main routine for the application that you are configuring.
AC_CONFIG_HEADER	A macro that specifies that you want to use a central include file containing cross-platform or multiplatform configuration information. The default name of this configuration include file is config.h. If your application does not depend on platform-specific capabilities, you can delete this macro entry—it is inserted by default by the autoscan script so that you do not overlook this feature if you want to take advantage of it.
AC_PROG_CC	A macro that checks for the existence of a C compiler.
AC_HEADER_STDC	A macro that checks for the existence of ANSI C header files by testing for the existence of the header files stdlib.h, stdarg.h, string.h, and float.h.
AC_CHECK_HEADERS	A macro that checks for a specific C language header file other than stdio.h.
AC_CONFIG_FILES	A macro that causes autoconf to create the specified file from a template file, performing variable substitutions as needed based on the other macro statements in the autoconf configuration file. By default, the template file has the same name as the specified file, but with an .in extension; you can specify another template file by using a colon to separate the name of the template file from the name of the file that is to be generated. For example, Makefile:foo.in specifies that the Makefile should be produced from a template file named foo.in.
AC_OUTPUT	A macro that causes autoconf to generate the shell script config.status and execute it. This is typically the last macro in any autoconf configuration file.

NOTE Just as in the C programming language, the values passed to autoconf macros are enclosed within parentheses. The preprocessors associated with programming languages have their own argument-parsing routines, which compare argument lists against the prototypes defined for each function. Because autoconf uses the *m4* macro preprocessor to expand keywords into more complex entries in your *Makefile* and may need to make multiple passes over your input files, arguments to Autoconf macros are often enclosed within square brackets, which tell *m4* where each argument begins and ends. Each pass of *m4* strips off one set of square brackets. If you are trying to pass arguments to an Autoconf macro and Autoconf is not processing it correctly, you may need to enclose that argument within square brackets. This is especially true for arguments that contain whitespace. The *autoscan* program automatically inserts sets of square brackets around all of the arguments to the Autoconf macros that it generates, just to be on the safe side.

Once you have generated a template configuration file using *autoscan*, you should copy or rename this file to the default autoconf configuration file *configure.ac*. At this point, you can delete any unnecessary statements, and modify any lines that contain default values. Continuing with our example, the modified *configure.ac* file would look like the following:

```
AC_PREREQ(2.56)
AC_INIT(Fibonacci, 1.0, vonhagen@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

The modified file specifies appropriate values for the *AC_INIT* macro, removes comments, and removes unnecessary macros (in this case) such as *AC_CONFIG_HEADER*.

NOTE The autoconf macros in this sample *configure.ac* file are a small subset of all of the autoconf macros that are available. The documentation for autoconf (provided in GNU info format) is the ultimate reference for the list of currently supported autoconf macros, since it reflects changes to autoconf that may have occurred since this book was published. Using autoconf's *-v* option when processing your *configure.ac* file displays a quick list of the macros supported by your version of autoconf.

At this point, you can actually run autoconf to generate an initial configure script, which should complete successfully if no typos exist in your configure.ac file.

```
$ autoconf
$ ls
autom4te.cache  calc_fib.c  configure  configure.ac  fibonacci.c
```

Running autoconf produces a configure script and a directory named autom4te.cache. This directory contains cached information about your application that is used by the autoconf package's autom4te application, which provides a high-level layer over m4.

Having successfully created a configure script, you can then try to execute it to watch it do its magic, as in the following output:

```
$ ./configure
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for stdlib.h... (cached) yes
configure: creating ./config.status
config.status: creating Makefile
config.status: error: cannot find input file: Makefile.in
$ ls
autom4te.cache  config.log      configure     fibonacci.c
calc_fib.c      config.status  configure.ac
```

Well, that was almost magic—the autoconf program exited with an error message. As shown in the previous example, running the autoconf program produces an output file named config.log that contains information about autoconf’s processing of your configure.ac file. This file lists how autoconf handled each command that it encounters, identifies variables set by autoconf as the program generates the configuration file, and lists any errors that it encountered. Though the problem in the previous example is fairly obvious, there are times when examining this log file can help you diagnose and correct problems in your configure.ac file.

In order to eliminate this error, you could remove the call to the AC_CONFIG_FILES([Makefile]) macro, but this would still not allow the configure script to produce a Makefile. In this example, the missing link between Autoconf and the successful creation of a Makefile is the template file from which the Makefile should be created after Autoconf performs any necessary variable substitutions, as specified by the AC_CONFIG_FILES macro. Our configure.ac file specified that autoconf should create the file Makefile; because no special template filename was specified, the name of the input template file is Makefile.in, which has not been created yet.

There are two approaches to creating a Makefile template:

- Create it manually using your favorite text editor and including statements that take advantage of the several hundred variables that autoconf can replace with platform- and system-specific values.
- Use automake to generate an appropriate Makefile.in file from a simple configuration file called Makefile.am, and then add an automake initialization macro to the configure.ac file.

Entire books have been devoted to the topic of using GNU Make and creating Makefiles, and minimizing the number of files to create and maintain manually is the whole point of the GNU build tools. Therefore, the next section provides an overview of how to create a Makefile.am file. Like the configure.ac file and the configure script produced by the autoconf program, the Makefile.am file is a small text file that is automatically expanded into a robust (and complex) Makefile.in file by the automake program.

In addition to showing you how to create a simple Makefile.am file and use the automake program to create the Makefile.in file, the next section also explains a number of other files that the automake program needs to be able to find in order to package your application successfully. Once these files exist, you can then rerun your configure script and automagically create a Makefile that will compile your application, run any tests that you define for the application, and even automatically create a compressed tar file that packages your application for distribution.

Creating Makefile.am Files and Other Files Required by Automake

Whereas the `configure.ac` file used by the `autoconf` program sets values for application configuration by passing values to a number of well-known macros, the `Makefile.am` file used by the `automake` program creates a `Makefile` template based on specific make targets that you define. These targets are defined using well-known variables called *primaries* in `automake` parlance. Table 7-3 describes the most commonly used `automake` primaries.

Table 7-3. Commonly Used Automake Primaries

Primary	Description
DATA	Files that are installed verbatim by a <code>Makefile</code> . These are typically things like help files, configuration data, and so on.
HEADERS	Header files that are associated with an application.
LIBRARIES	Static libraries that must be installed with an application so that they are available to an application at runtime.
LTLIBRARIES	Shared libraries that must be installed with an application so that they are available to the application at runtime. The LT stands for Libtool, which is a GNU build package application used for building shared libraries. Using Libtool is explained in Chapter 8.
MANS	Online reference (<code>man</code>) pages.
PROGRAMS	Binary applications.
SCRIPTS	Executable command files that are associated with the application. Because these are not compiled, they cannot be considered PROGRAMS; because their protections must be specially set so that they are executable, they cannot be considered DATA.
SOURCES	Source files required in order to compile an application and/or its libraries successfully in the first place.
TESTS	Test programs used to verify that an application is installed and executes correctly.
TEXINFOS	Online reference information for the <code>info</code> program that is in the GNU Texinfo documentation format (discussed in Chapter 10).

In order to define the highest-level targets in the `Makefile` that will be generated from an `automake` `Makefile.am` file, you attach various prefixes to these

primaries. The statements in a `Makefile.am` file are organized as standard `foo = bar` statements, where the target expression (`foo`) is formed by attaching a prefix to an appropriate primary, separated by an underscore. As an example, the simplest `Makefile.am` file that you can create defines the name of a single binary and then specifies the source files required to produce it. Continuing with the simple Fibonacci program, this `Makefile.am` file would look like the following:

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c calc_fib.c
```

The `bin_` prefix for the `PROGRAMS` primary tells `automake` that the `Makefile` template should specify the binary directory for installation of the Fibonacci program, as specified by the `make` variable `bindir`. The `fibonacci_` prefix before the `SOURCES` primary tells the `automake` program that these source files are associated with the `fibonacci` binary.

These two statements tell `automake` how to create a `Makefile.in` file from which `autoconf` can generate a `Makefile` that will successfully compile and install the sample application used in this chapter. The `Makefiles` produced by `automake`, listed in Table 7-4, actually provide instructions for performing a number of tasks related to the applications that you are packaging.

Table 7-4. Makefile Targets Produced by Automake

Target	Description
all	Creates a binary version of the application and builds any related data files and executable shell scripts. As with “regular” <code>Makefiles</code> , this is the default target if you simply execute the <code>make</code> command with no arguments against a <code>Makefile</code> produced by <code>autoconf</code> and <code>automake</code> .
check	Executes any tests associated with the application.
clean	Removes all binaries and intermediate object files produced during any previous execution of the <code>Makefile</code> .
dist	Creates a redistributable tar archive (tarball) for your application.
distcheck	Verifies that the distribution produced by <code>make dist</code> is complete and correct.
distclean	Returns the application directory to the exact state it would be in if you had freshly extracted the contents of the application archive.
install	Installs the application and any related data files and executable shell scripts.
uninstall	Uninstalls the application and any related data files and executable shell scripts from the application binary, library, and data directories.

The automake program's notion of a distribution imposes additional requirements on successfully generating a template Makefile for an application. Unlike autoconf, which only requires a single input file (`configure.ac`), a successful run of the automake program expects to find various auxiliary files in the directory where you are packaging your application. These files are all text files, with the names and content listed in Table 7-5.

Table 7-5. Auxiliary Files

File	Description
AUTHORS	A file that provides information about the authors of the program that you are packaging. This is typically the names and e-mail addresses of the primary authors, as well as information about any significant contributors to the application.
ChangeLog	A text file containing the history of changes to the application. This file lists the latest changes at the top of the file so that anyone installing the application package can easily identify everything that has changed in the current distribution. The format of this file is specified in the GNU Coding Standards (http://www.gnu.org/prep/standards.html), though this is not enforced programmatically. This file contains a description of every change to the application source code, regardless of whether it is visible to the user or not.
COPYING	The GNU Public License, a copy of which is installed with the automake application.
depcomp	An auxiliary shell script that is used to compile programs and generate dependency information.
INSTALL	A template set of generic installation applications for applications that are packaged with autoconf and automake. You can customize this file to reflect any installation issues specific to your application.
install-sh	An auxiliary shell script that is used to install all of the executables, libraries, data files, and shell scripts associated with your application.
mkinstalldirs	An auxiliary shell script that is used to create any directories or directory hierarchies required by an application.

Table 7-5. Auxiliary Files (continued)

File	Description
missing	An auxiliary shell script that automatically substitutes itself for any optional applications that may have been used when creating your application, but are not actually required for successfully compiling or installing your application. For example, if your distribution includes a .c file generated from a .y file by yacc or Bison, but the .y file has not actually been modified, you do not really need yacc or Bison on the system where you are installing the package. The missing script can stand in for programs including aclocal, autoconf, autoheader, automake, bison, flex, help2man, lex, makeinfo, tar (by looking for alternate names for the tar program), and yacc.
NEWS	A record of user-visible changes to the application. This differs from the ChangeLog in that this file describes only user-visible changes. It does not need to describe bug fixes, algorithmic changes, and so on.
README	A file that provides initial information for new users of the application. By convention, this is the first file that someone will examine when building or installing an updated version of an application.

The `automake` program automatically creates the `install-sh`, `mkinstalldirs`, `missing`, `COPYING`, `INSTALL`, and `depcomp` files by copying them in from templates provided as part of the `automake` distribution. In order to create these files, you must execute `automake` at least once with the `--add-missing` command-line option, as explained in the next section. You can then create the `NEWS`, `README`, `AUTHORS`, and `ChangeLog` files using a text editor or by using the `touch` program to create empty placeholders with the correct names.

Before we show you an example of running the `automake` program and creating these files in the next section, take a look at an overview of the process of using `automake` itself:

1. Create the `Makefile.am` file for your application.

2. Execute the `automake` program with the `--add-missing` command-line option to copy in any required files currently missing from your distribution but for which templates are provided as part of your `automake` installation. This step also identifies any files that you must create manually.

3. Create the files identified in the previous step.
4. Modify your application's configure.ac file to include the `AM_INIT_AUTOMAKE(program, version)` macro, customized for your application. This macro invocation should follow the `AC_INIT` macro in your configure.ac file.
5. Because the `AM_INIT_AUTOMAKE` macro is an automake macro rather than a standard autoconf macro, you will need to run the `aclocal` script in order to create a local file of m4 macro definitions (such as the definition of the `AM_INIT_AUTOMAKE` macro) that will be used by Autoconf.
6. Rerun the `autoconf` program.
7. Run the `automake` program again to generate a complete `Makefile.in` file from your `Makefile.am` file.

At this point, all of the files necessary for successfully using Autoconf and Automake to generate a valid `configure` script for your application, which will in turn generate a valid `Makefile`, are present.

NOTE *National health regulations in the United States require that we explicitly state that attempting to read or manually modify Makefiles generated by Autoconf and Automake can be hazardous to your sanity and downright counterproductive. If you find that the automatically generated Makefile does not work correctly on your system, you should either pass options to the `configure` script to induce the generation of a correct Makefile (as explained in the section "Running `configure` Scripts" later in this chapter), or correct the problem in the `Makefile.am` file, rerun `automake`, and then send a problem report to the maintainer of your application. If you actually find a bug in `automake`, you should report that to the appropriate mailing list as catalogued earlier in the section "Obtaining and Installing Automake."*

Running Autoconf and Automake

As discussed in the previous two sections, building an application using the GNU build tools, (autoconf, automake, and libtool) requires that you first create the input files that Autoconf and Automake use as the basis of the `configure` script and `Makefile.in` files that they respectively create. The previous sections summarized how to create the input files required by Autoconf and Automake, but did not provide a complete "cradle-to-Makefile" example that explained the sequence in which various input files should be created and various commands

executed. This section summarizes the entire process of using Autoconf and Automake to configure and build an application.

The basic sequence of using Autoconf, Automake, and the `configure` script itself to configure an application is the following:

1. Create an initial application configuration file (`configure.ac`), either manually or by using the `autoscans` script to generate a prototype that you can subsequently customize for your application.
2. Create an initial build configuration file (`Makefile.am`) and add the `automake` initialization macro statement to your `configure.ac` file.
3. (Optional) If your application uses a significant number of system header files, use the `autoheader` script to identify any system-dependent values that you need to define.

NOTE *The autoheader script only executes correctly if an AC_CONFIG_HEADER statement is present in your `configure.ac` file. If no such statement is present, the autoheader program exits with an error message.*

4. (Optional) If your application contains any conditional compilation flags (`#ifdef`, `#ifndef`, and so on), use the `ifnames` script to identify these flags so that you can create appropriate compilation targets or compilation flag settings in your `Makefile.am` file.
5. Run the `aclocal` script to produce the `aclocal.m4` file. This file contains a local version of the `autoconf` macros used in the `autoconf` and `automake` configuration files to reduce the amount of work that the `automake` program needs to do when generating the `Makefile.in` file from the `Makefile.am` build description file.
6. Run the `automake` program to generate the `Makefile.in` script from your `Makefile.am` build description. This requires creating some auxiliary files that are part of a “proper” GNU application, but this is easy enough to do without significant effort.
7. Run the `autoconf` program to generate a `configure` script.
8. Execute the `configure` script to generate automatically and execute a shell script called `config.status`, which in turn creates a `Makefile` that will enable you to compile your application, run tests, package your application for distribution, and so on.

For trivial applications, this may seem a bit much, but it is really an investment in the future of your application. For complex applications, the hassle inherent in manually creating Makefiles and maintaining and tweaking them for every possible platform and system configuration makes the GNU build environment and tools such as Autoconf and Automake an impressive win. In general, applications rarely get simpler: using Autoconf and Automake from the beginning of any application development project can reduce the configuration, maintenance, and distribution effort required as your application becomes more complex, more popular, or (hopefully) both. The remainder of this section shows an example that walks through the entire process.

Earlier in the section “Creating configure.ac Files,” the autoscan program generated a template configure.ac file for a simple application that looked like the following:

```
AC_PREREQ(2.56)
AC_INIT(Fibonacci, 1.0, vonhagen@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

As shown at the end of the section “Creating configure.ac Files” and explained in the section “Creating Makefile.am Files and Other Files Required by Automake,” the only thing missing from this file is an initialization statement for the macros used by automake, and the input file for automake itself, Makefile.am. After adding an automake initialization statement to the sample configure.ac file, that file looks like the following:

```
AC_PREREQ(2.56)
AC_INIT(Fibonacci, 1.0, vonhagen@vonhagen.org)
AM_INIT_AUTOMAKE(fibonacci, 1.0)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Next, you must create a basic configuration file for the automake application. A minimal automake configuration file (automake.am) for this application looks like the following:

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c calc_fib.c
```

At this point, the working directory looks like this:

```
$ ls
calc_fib.c configure.ac fibonacci.c Makefile.am
```

Once these files have been created, you run the `aclocal` command, which collects information about the location and definition of all `autoconf` macros into the file `aclocal.m4`, as in the following example:

```
$ aclocal
$ ls
aclocal.m4 calc_fib.c configure.ac fibonacci.c Makefile.am
```

NOTE *The `aclocal` script only creates the `aclocal.m4` file if your `configure.ac` file contains `AM_INIT_AUTOMAKE`. Otherwise, it succeeds in doing nothing or fails silently, depending on your perspective. If you execute the `aclocal` command but do not see the file `aclocal.m4`, check your `configure.ac` file to make sure that you have correctly added the `AM_INIT_AUTOMAKE` macro declaration.*

Next, you must run the `automake` program, telling it to create any auxiliary files it needs by copying in templates from your `automake` installation:

```
$ automake --add-missing
configure.ac: installing './install-sh'
configure.ac: installing './mkinstalldirs'
configure.ac: installing './missing'
Makefile.am: installing './COPYING'
Makefile.am: installing './INSTALL'
Makefile.am: required file './NEWS' not found
Makefile.am: required file './README' not found
Makefile.am: required file './AUTHORS' not found
Makefile.am: required file './ChangeLog' not found
Makefile.am: installing './depcomp'
```

At this point, your working directory looks like the following:

```
$ ls
aclocal.m4      configure.ac  fibonacci.c  Makefile.am  mkinstalldirs
autom4te.cache  COPYING       INSTALL       Makefile.in
calc_fib.c     depcomp       install-sh   missing
```

Now, you must create the other files required by the GNU build process that could not be created by copying from templates. As explained in the section “Creating Makefile.am Files and Other Files Required by Automake,” these are all files that actually provide information about the history of the program, its authors, and generic information about the program that users should know before attempting to build or install it—AUTHORS, ChangeLog, NEWS, and README. The following example simply creates these files using the touch command, but in actual use, you should create these files with a text editor to enter the correct content into these files:

```
$ touch AUTHORS ChangeLog NEWS README
$ ls
aclocal.m4      ChangeLog      fibonacci.c  Makefile.in    README
AUTHORS        configure.ac   INSTALL       missing
autom4te.cache  COPYING       install-sh   mkinstalldirs
calc_fib.c     depcomp       Makefile.am  NEWS
```

At this point, you are ready to run the autoconf program to create a configure script to generate a Makefile:

```
$ autoconf
$ ls
aclocal.m4      ChangeLog      depcomp       Makefile.am    NEWS
AUTHORS        configure     fibonacci.c  Makefile.in   README
autom4te.cache  configure.ac  INSTALL       missing
calc_fib.c     COPYING       install-sh   mkinstalldirs
```

Once you have created a configure script with no errors, the rest of the process is simple. To configure your application for your hardware and operating system, simply execute the configure script. The configure script creates a config.status script that takes all of your platform-specific information into account, and then executes that script to create a Makefile, as in the following example:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
```

```

checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for stdlib.h... (cached) yes
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands

```

Listing the contents of your working directory shows the following files:

```

$ ls
aclocal.m4      config.log      depcomp      Makefile.am      README
AUTHORS        config.status    fibonacci.c  Makefile.in
autom4te.cache  configure      INSTALL      missing
calc_fib.c     configure.ac   install-sh   mkinstalldirs
ChangeLog       COPYING        Makefile     NEWS

```

At this point, you can use the generated Makefile to build the sample application, as in the following example:

```

$ make
if gcc -DPACKAGE_NAME=\"Fibonacci\" -DPACKAGE_TARNAME=\"fibonacci\" \
-DPACKAGE_VERSION=\"1.0\" -DPACKAGE_STRING=\"Fibonacci 1.0\" \
-DPACKAGE_BUGREPORT=\"vonhagen@vonhagen.org\" \

```

```

-DPACKAGE=\"fibonacci\" -DVERSION=\"1.0\" -DSTDC_HEADERS=1 \
-DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DHAVE_STDLIB_H=1 \
-DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 \
-DHAVE_INTTYPES_H=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 \
-DHAVE_STDLIB_H=1 -I. -I. -g -O2 -MT fibonacci.o -MD \
-MP -MF ".deps/fibonacci.Tpo" \
-c -o fibonacci.o 'test -f \'fibonacci.c\' || echo \'./\'fibonacci.c; \' \
then mv ".deps/fibonacci.Tpo" ".deps/fibonacci.Po"; \
else rm -f ".deps/fibonacci.Tpo"; exit 1; \
fi
if gcc -DPACKAGE_NAME=\"Fibonacci\" -DPACKAGE_TARNAME=\"fibonacci\" \
-DPACKAGE_VERSION=\"1.0\" -DPACKAGE_STRING=\"Fibonacci\ 1.0\" \
-DPACKAGE_BUGREPORT=\"vonhagen@vonhagen.org\" \
-DPACKAGE=\"fibonacci\" -DVERSION=\"1.0\" -DSTDC_HEADERS=1 \
-DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DHAVE_STDLIB_H=1 \
-DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 \
-DHAVE_INTTYPES_H=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 \
-DHAVE_STDLIB_H=1 -I. -I. -g -O2 -MT calc_fib.o -MD \
-MP -MF ".deps/calc_fib.Tpo" \
-c -o calc_fib.o 'test -f \'calc_fib.c\' || echo \'./\'calc_fib.c; \' \
then mv ".deps/calc_fib.Tpo" ".deps/calc_fib.Po"; \
else rm -f ".deps/calc_fib.Tpo"; exit 1; \
fi
gcc -g -O2 -o fibonacci fibonacci.o calc_fib.o
$ ls
aclocal.m4      config.log      fibonacci      Makefile.am
AUTHORS        config.status   fibonacci.c  Makefile.in
autom4te.cache  configure     fibonacci.o  missing
calc_fib.c      configure.ac   INSTALL       mkinstalldirs
calc_fib.o      COPYING       install-sh   NEWS
ChangeLog       depcomp       Makefile     README

```

You can then execute your sample application:

```

$ ./fibonacci
Usage: fibonacci num-of-sequence-values-to-print
$ ./fibonacci 11
0 1 1 2 3 5 8 13 21 34 55

```

That is all there is to it! As explained in the section “Creating Makefile.am Files and Other Files Required by Automake,” the generated Makefile contains default commands for installing your program and even for running tests if you have defined some using the TESTS and CHECK primaries.

For complete information about the latest versions of Autoconf and Automake, download the latest versions as explained in the section “Installing and Configuring Autoconf and Automake,” and check the info files for the appropriate application or print their reference manuals. For a user-oriented discussion of the GNU build tools, refer to the book *GNU AUTOCONF, AUTOMAKE, and LIBTOOL*, Gary V. Vaughan et al. (New Riders, 2000. ISBN: 1-57870-190-2.). Though this book discusses substantially older versions of the GNU build tools and is therefore somewhat dated, it is an excellent and friendly book that provides detailed insights into using autoconf, automake, and libtool. (Using libtool is discussed in the next chapter of this book.)

This section explained how to create the input and auxiliary files required by Autoconf and Automake and how to execute these applications and the scripts that they produce in order to compile and install your application. You may have noticed that nothing in the Autoconf or Automake input files specified details such as the location of the resulting binaries and other application files. These values are set by passing them to the configure script produced by Autoconf, which then generates a Makefile that encapsulates this information. The next section discusses the arguments you can pass to the configure script to customize the Makefile that it creates.

Running configure Scripts

As mentioned at the end of the previous section, the configure script generated by autoconf produces a Makefile that conforms to the instructions you provided in the Makefile.am file, but uses various defaults. For example, the default installed location of any application for any Autoconf/Automake application is in subdirectories of /usr/local. This is useful for user applications and for testing new versions of system applications, but at some point, you will want to configure and build a “standard” Linux application from source code and install it into some system directory. The Autoconf and Automake applications themselves are good examples, as discussed in the section “Deciding Whether to Upgrade or Replace Autoconf and Automake.” By default, new versions of these applications that you download and build are installed in /usr/local/bin, but eventually, you will want to purge your system of older versions by overwriting the versions of these programs that are installed in /usr/bin with most Linux distributions, and then deleting your “test” versions in /usr/local/bin.

The configure script has a number of command-line options that enable you to customize the behavior of the Makefile that it produces. Some of these command-line options are obscure, while others are quite useful. Table 7-6 shows the most commonly used configure options and the effect that they have on the Makefile generate by the configure script.

Table 7-6. Summary of configure Script Options

Option	Description
<code>-h, --help</code>	Display this help and exit.
<code>-V, --version</code>	Display version information and exit.
<code>-n, --no-create</code>	Do not create output files. Similar to <code>make -n</code> , these options show you what the configure script would do, but do not actually perform those operations.
<code>--prefix=PREFIX</code>	Installs files in appropriate subdirectories of PREFIX.

The most commonly used option with the configure script is the `--prefix` option. The default value for PREFIX is `/usr/local`. When you execute the configure script without the `--prefix` option, `make install` will install all the executable files for your application in `/usr/local/bin`, all libraries in `/usr/local/lib`, and so on. You can specify an installation prefix other than `/usr/local` by using the `--prefix` option—for instance, `--prefix=$HOME` to install the application in the appropriate subdirectories of your home directory, or `--prefix=/usr` to install the application in the appropriate subdirectories of `/usr`, such as `/usr/bin`, `/usr/lib`, `/usr/man`, and so on.

The configure scripts created by the Autoconf application provide a number of options that give you detailed control over various aspects of the Makefiles that are created by these scripts. For complete information about the options provided by any configure script, execute the command `./configure --help`.

CHAPTER 8

Using Libtool

THIS CHAPTER DISCUSSES LIBTOOL, an open source application that simplifies creating and maintaining precompiled collections of functions (for languages such as C) or classes and associated methods (in object-oriented languages such as C++), generally referred to as *libraries*, that can subsequently be used by and linked to applications. Throughout this chapter, the term *code libraries* is used as a general, language-independent term for a precompiled collection of functions or classes that is external to an application and linked into the application during final compilation to enable the application to execute the functions or methods present in the library.

This chapter begins by explaining the basic principles of libraries, the different types of libraries that are used in application development, and how each is used by applications. It then introduces Libtool and explains how to download, build, and use Libtool when building applications. The last sections explain potential problems that you may encounter when using Libtool, outline how to correct or work around those problems, and summarize additional sources of information about Libtool and its use.

Introduction to Libraries

A library is a collection of precompiled code that can be used by any application that is linked to it. Linking typically occurs during the final stage of compiling an application. Source code modules that use functions located in libraries are typically compiled with compiler options such as GCC's `-c` option, which tells the compiler to compile the source module but not to attempt to resolve references to functions that are not located in the source module that is currently being compiled. These references are resolved during the final stage of compilation when an executable version of the application is actually produced. How these references are resolved depends on the type of library that the executable is being linked with.

Three basic types of libraries are available on most modern computer systems: static, shared, and dynamically loaded libraries. The next sections define each of these and discuss their advantages and disadvantages.

Static Libraries

Static libraries are the oldest and simplest form of code libraries. Each application that is linked with a static library essentially includes a copy of the library in the final executable version of the application. Using static libraries is fairly simple because references to function calls in the code library can be resolved during compilation. When linking a static library to an application, the address of any function in the library can be determined and preresolved in the resulting executable. Static libraries typically use the extension .a (for archive) or .sa (for static archive).

Aside from their relative simplicity during compilation, a significant advantage of using static libraries is that applications that are statically linked with the libraries that they use are completely self-sufficient. The resulting executable can be transferred to any compatible computer system and will execute correctly there because it has no external dependencies. Applications that are statically linked to the libraries that they use can be executed more quickly than applications that use shared or dynamically loaded libraries because statically linked applications require no runtime or startup overhead for resolving function calls or class references. Similarly, statically linked applications require somewhat less memory than shared or dynamically loaded applications because they require a smaller runtime environment with less overhead than is required to search for and call functions in a shared or dynamically loaded library.

While static libraries are easy to use and have some advantages for portability, they also have some distinct disadvantages, the most significant of which are the following:

- *Increased application size:* Bundling a copy of a static library into each executable that uses it substantially increases the size of each resulting executable. The portions of the library used in the application must be bundled into each executable because it is not possible to extract specific functions from the library.
- *Library updates require recompiling applications:* Any bug fixes or enhancements made to libraries that have been statically linked into applications are not available to those applications unless the applications are recompiled with the new version of the library.

To address these problems, many Unix and Unix-like systems support shared and dynamically loaded libraries, which are discussed in the next sections.

Shared Libraries

First introduced by Sun Microsystems in the early 1990s, shared libraries are centralized code libraries that an application loads and links to at runtime,

rather than at compile time. Programs that use shared libraries only contain references to library routines. These references are resolved by the runtime link editor when the programs are executed. Shared libraries typically have the file extension .so (for shared object), though they may use other extensions such as the .dylib extension (for dynamic shared library) used on Mac OS X systems. Mac OS X is something of an anomaly in terms of how it names and references shared libraries; the remainder of this section primarily focuses on systems that use “classic” shared library implementations such as Linux and Solaris.

Shared libraries provide a number of advantages over static libraries, most notably the following:

- *Simplified application maintenance:* Shared libraries can be updated at any time, and they use a version numbering scheme so that applications can load the “right” version of a shared library. Applications compiled against a shared library can automatically take advantage of bug fixes or enhanced functionality provided in newer versions of those libraries. Applications compiled against specific or previous-generation shared libraries can continue to use those libraries without disruption. Shared library version numbering is explained later in this section.
- *Reduced system disk space requirements:* Applications that use shared libraries are smaller than applications that are statically linked with libraries because the library is not bundled into the application. This is especially significant in graphical applications, such as X Window System applications, that use functions contained in a relatively large number of graphics libraries. Compiling these types of applications with static libraries would increase their size three or four times. Similarly, running a graphical desktop environment such as GNOME or KDE or X Window manager and some number of X Window system applications that use shared libraries can significantly reduce overall system memory consumption, since all of these applications share access to a single copy of each shared library rather than including their own copy.
- *Reduced system memory requirements:* Applications that access shared libraries still need to load the shared libraries into memory when these applications are executed. However, shared libraries need only be loaded into system memory once in order to be used by any application that references the library. Because multiple applications can simultaneously share access to a single copy of a centralized shared library, overall system memory consumption is reduced when more than one application that references a shared library is being executed.

Using shared libraries has some negative aspects as well as positive ones. First, using shared libraries makes it more complex to copy single executables

between compatible computer systems, because it increases the external requirement for the application. Each computer system must provide the shared libraries, as well as appropriate versions of these shared libraries. For example, consider the X Window system `xterm` application on a modern Linux system. The Linux `ldd` (list dynamic dependencies) command shows that the `xterm` application requires the following shared libraries:

```
$ ldd 'which xterm'
    libXft.so.1 => /usr/X11R6/lib/libXft.so.1 (0x4002f000)
    libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0x4003e000)
    libXrender.so.1 => /usr/X11R6/lib/libXrender.so.1 (0x40087000)
    libXaw.so.7 => /usr/X11R6/lib/libXaw.so.7 (0x4008c000)
    libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6 (0x400e6000)
    libXt.so.6 => /usr/X11R6/lib/libXt.so.6 (0x400fc000)
    libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x4014e000)
    libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x40158000)
    libXpm.so.4 => /usr/X11R6/lib/libXpm.so.4 (0x4016f000)
    libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x4017e000)
    libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x4018c000)
    libncurses.so.5 => /usr/lib/libncurses.so.5 (0x4026a000)
    libutempter.so.0 => /usr/lib/libutempter.so.0 (0x402a9000)
    libc.so.6 => /lib/i686/libc.so.6 (0x42000000)
    libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0x402ac000)
    libdl.so.2 => /lib/libdl.so.2 (0x402d1000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
    libexpat.so.0 => /usr/lib/libexpat.so.0 (0x402d4000)
```

The second downside of using shared libraries is that their use imposes a slight performance penalty when any application using the shared library is executed, because references to shared library routines must be resolved by finding the libraries containing those routines at runtime.

At runtime, resolving references to shared libraries is done by a shared library loader. Systems that use shared libraries typically maintain a cache that identifies the locations of shared libraries on a computer system in order to minimize the amount of time the system spends trying to locate shared libraries whenever an application that uses shared libraries is executed. For example, Linux systems use the shared library loader `/lib/ld.so` and maintain a cache of the locations of shared libraries on your system in the file `/etc/ld.so.cache`. The cache file is created and maintained using the `ldconfig` application, based on the contents of the file `/etc/ld.so.conf`, which contains a list of directories to search for shared libraries.

In order for systems and applications that use shared libraries to support multiple versions of those libraries and to distinguish between them, most shared library implementations use a common set of naming conventions for

those libraries. The actual name of a shared library is composed of the prefix “lib”, the name of the library, and the basic shared library file extension .so, followed by a period and the true version number of the shared library. The version number is either two or three digits. On Solaris systems, the version number is composed of a major and minor number, separated by a period, such as libc.so.2.9. Linux systems use the same naming scheme, but also add a release number to the end, such as libxmms.so.1.2.1.

Shared libraries also have a special name called the *soname*. The soname is composed of the prefix “lib”, the name of the library, and the basic shared library file extension .so, followed by a period and the major version number of the shared library. The soname is typically a symbolic link to the appropriate shared library file. Continuing with the examples in the previous paragraph, the soname of the Solaris libc.so.2.9 library would be libc.so.2, whereas the soname of the Linux libxmms.so.1.2.1 library would be libxmms.so.1.

The final name used to refer to shared libraries is often called the *linker name*. This name, which a compiler uses when referring to a library, is simply the soname without any version number. Continuing with the examples in the preceding paragraphs, the linker name of the Solaris libc.so.2.9 library would be libc.so, whereas the linker name of the Linux libxmms.so.1.2.1 library would be libxmms.so.

The version numbering scheme used in shared library names makes it possible to maintain several compatible and incompatible versions of a single shared library on the same system. The major, minor, and release version numbers of a shared library are updated based on the type of changes made between different versions of the shared library. When a new version of a shared library is released in order to fix bugs in a previous version, only the release number is incremented. When new functions are added to a shared library but existing functions in the library still provide the same interfaces, the minor version number is incremented and the release number is reset. When interfaces in functions in a shared library change in such a way that existing applications cannot call those functions, the major number is updated and the minor and release numbers are reset. Applications that depend on specific interfaces to functions can then still load the shared library with the correct major and minor number at runtime, while applications compiled against a new version of the same shared library (featuring different parameters or a different parameter sequence) to existing functions can link against the version of the shared library with the new major number.

Dynamically Loaded Libraries

The final type of libraries in common use are dynamically loaded (DL) libraries. Dynamically loaded libraries are code libraries that an application can load and

reference at any time while it is running, rather than arbitrarily loading them all at runtime. Unlike static and shared libraries, which have different formats and are automatically loaded in different ways, dynamically loaded libraries differ from other types of libraries in terms of how an application uses them, rather than how the runtime environment or compiler uses them. Both static and shared libraries can be used as dynamically loaded libraries.

The most common use for DL libraries is in applications that use plug-ins or modules. A good example of a common use of dynamically loaded libraries are the Pluggable Authentication Modules (PAM) employed by the authentication mechanism on many Linux, FreeBSD, Solaris, and HP-UX systems. The primary advantage of DL libraries is that they enable you to defer the overhead of loading specific functionality unless it is actually required. DL libraries also make it easy to separately develop, maintain, and distribute plug-ins independent of the executable that actually invokes and loads them. They are also useful in CORBA and Com environments where you may not actually know which library to load until runtime.

Because they are loaded “as required,” an application that wishes to use DL libraries does not need any prior knowledge of the libraries other than their names or, more commonly, the locations to search for them. Most applications that use dynamically loaded modules use some sort of indirection to determine the DL libraries that are available—hardwiring a list of available plug-ins into an application largely defeats the flexibility that they provide. Instead, applications that use DL libraries typically use a well-known list of directories to search for DL libraries, or read a list of relevant directories from an environment variable or configuration file.

Applications that use DL libraries employ a standard API for opening DL libraries, examining their contents, handling errors, and so on. In C language applications on Linux systems, this API is defined and made available by including the system header file `<dlfcn.h>`. The actual functions in the API (unfortunately) differ across platforms—for example, Linux, Solaris, and FreeBSD systems use the `dlopen()` function to open libraries, HP-UX systems use `shl_load()`, and Windows systems use the radically different (of course) `LoadLibrary` interface for their dynamic link libraries (DLLs). The GNU Glibc library provides a more generic interface to DL libraries that is intended to hide most platform-specific differences in DL library implementations. The Libtool library tool, the subject of this chapter, also provides a general interface for DL libraries through its `ltdl.so` library.

What Is Libtool?

Even creating a simple static library and using it in your applications requires special compilation and linking options. The various types of libraries discussed in the previous section, the differences in using them in applications, and the

differences in how or if applications need to be able to resolve symbols in those libraries each brings its own complexity and caveats to developing applications that use libraries. If you are writing an application that uses libraries on multiple platforms, multiplying these considerations across target systems and library implementations is the stuff of nightmares.

The GNU Library Tool (Libtool) consists of a shell script and auxiliary libraries that are designed to encapsulate platform- and library-format-specific differences across platforms. Libtool does this by providing a consistent user interface that is supported on a wide variety of platforms. Libtool is most commonly invoked by the configure scripts and Makefiles produced by Autoconf and Automake. However, like all good GNU utilities, Libtool does not depend on Autoconf or Automake, and vice versa—Libtool is just another tool that configure scripts and Makefiles can use if it is available on the system where you are building your application. Manually created Makefiles can invoke Libtool just as easily as can those created by Automake.

Libtool introduces some new conventions in library extensions that are designed to make differences between the types of libraries supported on different types of systems. Though these are really internal to Libtool, it is useful to discuss them up front to minimize confusion if you are watching the build process and wondering what is going on when you see new types of files being created. Most notably, most C language compilers use the extension convention .o for intermediate object files produced during the compilation process. To simplify things internally across systems that may or may not support shared libraries, Libtool creates files with the extensions .la, which are files that contain platform-specific information about your libraries, and .lo (library object), which are optional files produced during the build process. When generated by Libtool on systems that do not support shared libraries, these .lo files contain timestamps rather than actual object code. When actually building libraries, Libtool uses the information about your system's library support to determine how (and if) to use the library object files, and employs the information in the .la files to identify the characteristics of your system's support for libraries and the actual libraries that should be used.

Aside from masking platform-specific differences in library construction and use, the most interesting problem that Libtool solves is the problem of linking applications against shared libraries that are being developed at the same time, and are therefore not preinstalled in one of the locations where your system searched for shared libraries. Though you could manually resolve this problem by setting an environment variable such as `LD_LIBRARY_PATH` to include the directories where the shared libraries you are developing are located, this would truly be a pain in complex applications that use multiple shared libraries. Libtool handles this for you automatically, which is well worth the cost of admission all by itself (especially when that cost is zero). Libtool is an elegant but easy-to-use solution to a variety of complex and/or irritating problems.

Downloading and Installing Libtool

In order to provide a consistent interface across many different types of systems, each of which may use and support different libraries, you must explicitly build Libtool on each platform on which you plan to use it. Even though it is a shell script, the script contains a number of platform-specific variable settings that are required in order for Libtool to identify and take advantage of the capabilities of your system.

Most systems that use GCC also provide auxiliary tools such as Libtool, Automake, and Autoconf. However, if you are building GCC for your system or simply installing the latest and greatest version of GCC on your system, you will also want to obtain, build, and install the latest version of Libtool on your system. A link to a downloadable archive of the source code for the latest version of Libtool is available from the Libtool home page at <http://www.gnu.org/software/libtool/libtool.html>. However, we recommend that you use Libtool 1.4.2 because installing a newer version may cause conflicts with portions of any version of Libtool that is already installed on your system. Newer versions are fine if you are absolutely sure that Libtool is not currently installed on your system. Though not used in this section, Libtool 1.4.3 was the newest version available at the time of this writing. You can download Libtool 1.4.2 from the Libtool page at any GNU mirror, such as <ftp://mirrors.kernel.org/gnu/libtool>.

The remainder of this section will use an archive file named libtool-1.4.2.tar.gz and source code directory named libtool-1.4.2 as examples—when following the instructions in this section on your system, you should substitute the name of the archive file you downloaded and the name of the source code directory that you extracted from it.

Installing Libtool

Once you have downloaded the archive, you can extract its contents using a command like the following:

```
# tar zxf libtool-1.4.2.tar.gz
```

This will create a directory with the basename of the tar file that contains the Libtool source code. In this example, the name of the extracted directory would be libtool-1.4.2.

Make the extracted directory your working directory and execute the configure script provided as part of the Libtool source distribution.

```
# cd libtool-1.4.2
# ./configure
[standard configure output deleted]
```

NOTE If you want to install the new version of Libtool in the appropriate subdirectories of a directory other than the default directory /usr/local, you can use the configure script's --prefix option to specify the new directory hierarchy. For example, if you want to install Libtool in the subdirectories of /usr, you would execute the command ./configure --prefix=/usr. You may want to do this if you are updating a system, such as a default Red Hat Linux distribution, on which Libtool is already installed. If you are using a package-based computer system, you may want to use your system's package management application to delete the package that provides Libtool so that your system does not think that another version is still installed. For example, on systems that use the Red Hat Package Manager (RPM), you can identify and remove the package that provides Libtool by using commands such as the following:

```
# rpm -qf 'which libtool'
libtool-1.4.2-12
# rpm -e libtool-1.4.2-12
```

There is really no need to remove previous versions of Libtool, except that it avoids potential confusion should you encounter a problem and need to provide someone with specific information about the version of Libtool and associated files that you are using.

Next, use the make command to build Libtool (output included due to its brevity):

```
# make
Making all in .
make[1]: Entering directory '/home/wvh/src/libtool-1.4.2'
CONFIG_FILES=libtoolize CONFIG_HEADERS= /bin/sh ./config.status
creating libtoolize
chmod +x libtoolize
make[1]: Leaving directory '/home/wvh/src/libtool-1.4.2'
Making all in libltdl
make[1]: Entering directory '/home/wvh/src/libtool-1.4.2/libltdl'
/bin/sh ./libtool --mode=compile gcc -DHAVE_CONFIG_H -I. -I. -g -O2 -c l ltdl.c
rm -f .libs/ltdl.lo
gcc -DHAVE_CONFIG_H -I. -I. -g -O2 -c ltdl.c -fPIC -DPIC -o .libs/ltdl.lo
gcc -DHAVE_CONFIG_H -I. -I. -g -O2 -c ltdl.c -o ltdl.o >/dev/null 2>&1
mv -f .libs/ltdl.lo ltdl.lo
/bin/sh ./libtool --mode=link gcc -g -O2 -o libltdl.la -rpath /usr/local/lib \
-no-undefined -version-info 4:0:1 ltdl.lo -ldl
rm -fr .libs/libltdl.la .libs/libltdl.* .libs/libltdl.*
gcc -shared ltdl.lo -ldl -Wl,-soname -Wl,libltdl.so.3 -o .libs/libltdl.so.3.1.0
```

```
(cd .libs && rm -f libltdl.so.3 && ln -s libltdl.so.3.1.0 libltdl.so.3)
(cd .libs && rm -f libltdl.so && ln -s libltdl.so.3.1.0 libltdl.so)
ar cru .libs/libltdl.a ltdl.o
ranlib .libs/libltdl.a
creating libltdl.la
(cd .libs && rm -f libltdl.la && ln -s ..//libltdl.la libltdl.la)
make[1]: Leaving directory '/home/wvh/src/libtool-1.4.2/libltdl'
Making all in doc
make[1]: Entering directory '/home/wvh/src/libtool-1.4.2/doc'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/wvh/src/libtool-1.4.2/doc'
Making all in tests
make[1]: Entering directory '/home/wvh/src/libtool-1.4.2/tests'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/wvh/src/libtool-1.4.2/tests'
#
#
```

You can then install this version of Libtool and its associated scripts and libraries using the `make install` command. The previous examples showed the extraction and compilation process being done as root; although this is not necessary to download and compile Libtool, you will almost certainly have to be the root user in order to install it.

Congratulations! You have built and installed Libtool!

Files Installed by Libtool

The files and directories listed in this section are installed in subdirectories of `/usr/local` by default. Many Linux distributions install these files and directories as subdirectories of `/usr` instead. See the note in the previous section for information on modifying the default location where Libtool and related files and directories will be installed. This section discusses the location of files installed by a default installation of Libtool.

The files that are installed as part of Libtool are the following:

- *config.guess*: A shell script used when Libtool is employed with Autoconf in order to guess the system type. This script is installed in `/usr/local/share/libtool` by default.
- *config.sub*: A shell script used when Libtool is employed with Autoconf, primarily containing routines to validate your system's operating system and configuration. This script is installed in `/usr/local/share/libtool` by default.

- *libltdl.a, libltdl.la, libltdl.so, libltdl.so.3, libltdl.so.3.1.0*: The actual library files and related symlinks, soname, etc., required to use Libtool's DL library interface (ltdl). These files are installed in /usr/local/lib by default.
- *libtool*: The primary Libtool shell script, installed by default in /usr/local/bin.
- *libtool.m4*: A collection of macro definitions used when Libtool is employed with Autoconf and Automake. This file is installed in /usr/local/share/aclocal by default.
- *libtool/libltdl*: A directory containing the files necessary to build Libtool's DL library interface (ltdl) using Autoconf and Automake. This directory contains the files README, stamp-h.in, COPYING.LIB, Makefile.am, Makefile.in, acinclude.m4, aclocal.m4, config-h.in, configure, configure.in, ltdl.cm, ltdl.c, and ltdl.h. This directory is installed in /usr/local/share by default.
- *libtoolize*: A shell script that creates the files required to use Libtool with Autoconf and Automake. This script is installed by default in /usr/local/bin.
- *ltdl.h*: The include file necessary to use Libtool's DL library interface (ltdl) in an application. This file is installed in /usr/local/include by default.
- *ltdl.m4*: A collection of macro definitions used when Libtool is employed with Autoconf and Automake, and then only when Libtool's DL library interface (ltdl) is being used. This file is installed in /usr/local/share/aclocal by default.
- *libtool.info, libtool.info-1 through libtool.info-5*: The files used by the info program to display the up-to-date documentation for Libtool. Like most GNU programs, the documentation provided for Libtool in info format is more extensive and more up to date than that provided in its main page. These files are installed in /usr/local/info by default.
- *ltmain.sh*: A collection of shell functions that are used by Libtool. This file is installed in /usr/local/share/libtool by default.

Using Libtool

The easiest way to explain how to use Libtool is to actually provide some examples. If you are writing an application that uses libraries, we suggest that you also use Autoconf and Automake, and simply incorporate the appropriate Libtool macros in the `Makefile.am` file that you created for use by Automake. A large part of the amazing beauty of the GNU configuration and compilation tools is how well they work together, and how trivial it is to use them together automatically. Today's developers truly have the opportunity to stand on the shoulders of GNU giants.

The previous paragraph aside, Libtool has an interesting interface that you may want to invoke manually at times. The next section, “Using Libtool from the Command Line,” explains the commands available within Libtool and how to use them manually. The section “Using Libtool with Autoconf and Automake” explains the statements you can add to your `Makefile.am` file in order to automatically invoke the Libtool script.

NOTE *Though Libtool commands can be manually integrated into Makefiles, there is not much point in doing that unless you are a true Luddite or are masochistic. We therefore do not explicitly discuss that here. If you are interested in doing this, you can always examine a Makefile that incorporates Libtool support that was produced by the Autotools, and reverse-engineer manual integration of Libtool into your Makefiles.*

Using Libtool from the Command Line

When used from the command line, Libtool enables you to execute various development-related commands by specifying particular command-line options and/or a particular mode of operation. These modes of operation use Libtool as a wrapper to encapsulate the functionality of development and diagnostic tools such as GCC and GDB, while still taking advantage of the functionality provided by Libtool.

The next section discusses Libtool’s command-line options and their meaning. The section that follows the discussion of command-line options discusses the modes of operation available in Libtool and any mode-specific command-line options that are available. It also provides specific examples of using Libtool in each mode.

Command-Line Options for Libtool

Libtool provides a number of command-line options and associated parameters that enable you to specify a mode of operation, debug its execution, display

information about its configuration and associated variables, and so on.

Following are the command-line options provided by Libtool:

- **config:** Displays information about the characteristics of the host and operating system on which Libtool was configured and associated Libtool variable settings.
- **debug:** Activates the shell's verbose mode (-x) to provide tracing information. Using this option shows each line in the Libtool script as it executes, enabling you to monitor and debug its operation by seeing which branches are taken by Libtool, the command and associated options that Libtool executes, and so on.
- **dry-run:** Displays the commands that would be executed by Libtool based on the other options that are specified, but does not actually execute those commands. This option is analogous to the `make` command's -n option. To reinforce this parallelism, the -n option can also be used as an equivalent to the --dry-run option.
- **features:** Displays information about the type of host on which Libtool was compiled, and whether that platform supports shared and/or static libraries.
- **finish:** A short-cut for specifying `finish` mode by using the --mode option.
- **help:** Displays information about the options supported by Libtool and then exits.
- **mode=MODE:** Enables you to specify a mode of operation for Libtool. Modes are provided as shorthand for the commands required to perform a specified type of conceptual operation. For example, using the --mode=compile option invokes GCC and any related portions of Libtool. Libtool provides a number of supported modes, many of which have additional command-line options that are specific to a given mode. Libtool's modes are discussed in the next section.
- **quiet:** Suppresses the display of status messages during Libtool execution.
- **silent:** Identical to the --quiet option in that it suppresses the display of status messages during Libtool execution.
- **version:** Displays the version of Libtool that is being executed, and exits.

The next section discusses the various modes of operation that can be specified using the --mode command-line option, and any additional command-line

options that are specific to each mode of operation. Where possible, this section uses examples from the sample Fibonacci program that was introduced in Chapter 6.

Command-Line Modes for Libtool Operation

As discussed in the previous section, a variety of conceptual tasks can be performed by Libtool by specifying them using Libtool's `--mode` command-line argument. Some of these modes provide additional command-line options that are specific to a given mode.

The following modes of operation and additional, mode-specific command-line options are available in Libtool:

- `clean`: Removes files from the build directory. This mode requires that you specify the name of the command used to remove files, any options that you want to pass to the command, and the name(s) of the file(s) that you want to remove. If any of the files that you specify for removal are Libtool objects or libraries, all of the files associated with it are removed. For example, the following command only removes a single object file:

```
$ libtool --mode=clean rm fibonacci.o
rm fibonacci.o
```

The following command, where the file to be removed is a Libtool library, removes all files associated with that library:

```
$ libtool --mode=clean rm libfib.la
rm libfib.la .libs/libfib.so.1.0.0 .libs/libfib.so.1 .libs/libfib.so \
.libs/libfib.a .libs/libfib.la .libs/libfib.lai
```

- `compile`: Compiles a source file into a Libtool object. This mode requires that you specify the name of the compiler and any mandatory compiler options. For example, the following command compiles the main module of the sample application into a standard object file and a Libtool library object:

```
$ libtool --mode=compile gcc -c fibonacci.c
rm -f .libs/fibonacci.lo
gcc -c fibonacci.c -fPIC -DPIC -o .libs/fibonacci.lo
gcc -c fibonacci.c -o fibonacci.o >/dev/null 2>&1
mv -f .libs/fibonacci.lo fibonacci.lo
```

Some additional command-line options are available when executing Libtool in compile mode. These are the `-o` option, which must be followed by the name of the object file whose creation you want to force; the `-prefer-pic` option, which tells Libtool to attempt to build position-independent code

(PIC) objects (required for shared libraries produced by Libtool); the `-prefer-non-pic` option, which tells Libtool to try to build non-PIC library objects (required for static libraries produced by Libtool) and overrides the default settings for your platform if it supports shared libraries; and the `-static` option, which forces Libtool to produce only a standard object file (`.o` file) that can be used to produce a standard static library.

- `execute`: Automatically sets the environment variable used to identify the location of required libraries (`LD_LIBRARY_PATH`), and then attempts to execute the specified program. This will work only if the executable and all shared or static libraries that it requires have successfully been compiled.

For example:

```
$ libtool --mode=execute ./fibonacci
lt-fibonacci: error while loading shared libraries: \
    libfib.so.1: cannot open shared object file: No such file or directory
$ make >/dev/null 2>&1
$ libtool --mode=execute ./fibonacci
Usage: fibonacci num-of-sequence-values-to-print
```

As shown in the last of these examples, in order to completely execute the specified argument, you must also specify any command-line arguments that the application itself requires. Because Libtool does a certain amount of quoting when passing arguments to an application executed in this fashion, this actually exposes a bug in our sample Fibonacci application, which does not check whether the argument being passed is actually an integer. Physician, heal thyself!

One additional command-line option is available when executing Libtool in execute mode. This is the `-dlopen` option, which takes one argument, the full pathname of a shared library that you want to add to your `LD_LIBRARY_PATH` environment variable prior to executing the command you specified.

- `finish`: Completes the installation of installed libraries by executing the system `ldconfig` command to add a specified directory to the system's working `LD_LIBRARY_PATH`. To permanently add a directory to the system's shared library search path, you should add the name of that directory to the configuration file for your loader. On most Linux systems, this is the file `/etc/ld.so.conf`. You must then rerun the `ldconfig` command to add the new directory to your system's library cache.

- **install:** Installs libraries or executables. This Libtool command provides a manual way of forcing the execution of specified files. This is typically more easily done using the `make install` command from the command line, but can be useful if you want to force the execution of a single component of your application into a specified directory for testing purposes. One advantage of executing this from the libtool command line is that Libtool understands the format of its library objects files, and installs all of the files associated with a specified library. For example:

```
# libtool --mode=install install libfib.la /usr/local/lib
install .libs/libfib.so.1.0.0 /usr/local/lib/libfib.so.1.0.0
(cd /usr/local/lib && rm -f libfib.so.1 \
    && ln -s libfib.so.1.0.0 libfib.so.1)
(cd /usr/local/lib && rm -f libfib.so \
    && ln -s libfib.so.1.0.0 libfib.so)
install .libs/libfib.lai /usr/local/lib/libfib.la
install .libs/libfib.a /usr/local/lib/libfib.a
ranlib /usr/local/lib/libfib.a
chmod 644 /usr/local/lib/libfib.a
PATH="$PATH:/sbin" ldconfig -n /usr/local/lib
-----
Libraries have been installed in:           /usr/local/lib
```

If you ever happen to want to link against installed libraries in a given directory, `LIBDIR`, you must either use libtool, and specify the full pathname of the library, or use the '`-L`' flag during linking and do at least one of the following:

- add `LIBDIR` to the '`LD_LIBRARY_PATH`' environment variable during execution
- add `LIBDIR` to the '`LD_RUN_PATH`' environment variable during linking
- use the '`-Wl,--rpath -Wl,LIBDIR`' linker flag
- have your system administrator add `LIBDIR` to '`/etc/ld.so.conf`'

See any operating system documentation about shared libraries for more information, such as the `ld(1)` and `ld.so(8)` manual pages.

-
- **link:** Creates a library or an executable. This Libtool mode requires that you specify a complete link command. Neglecting to do so is the most common first-time error when using Libtool in this mode, as in the following example:

```
$ libtool --mode=link fibonacci
libtool: link: you must specify an output file
Try 'libtool --help --mode=link' for more information.
```

The next most common error when using this Libtool mode is neglecting to specify libraries required by the application, as in the following example:

```
$ libtool --mode=link gcc fibonacci.c -o fibonacci
gcc fibonacci.c -o fibonacci
/tmp/ccWaF9hG.o: In function 'main':
/tmp/ccWaF9hG.o(.text+0x5b): undefined reference to 'calc_fib'
collect2: ld returned 1 exit status
```

Using this Libtool mode in the correct way looks something like the following:

```
$ libtool --mode=link gcc fibonacci.c -o fibonacci -lfib
gcc fibonacci.c -o .libs/fibonacci \
  /home/wvh/writing/gcc/src/.libs/libfib.so -Wl,--rpath \
-Wl,/usr/local/lib
creating fibonacci
```

Libtool's link mode has more mode-specific command-line options, as you can see by the list that follows, than any other mode, all of which can be specified as part of your link command.

- **-all-static:** Specifies to not do any dynamic linking. If the target linked executable is a library, a static library will be created. If it is an executable, it will be statically linked.
- **-avoid-version:** If possible, specifies to not add a version suffix to libraries.
- **-dlopen *file*:** Causes the resulting executable to try to preopen the specified object file or library and add its symbols to the linked executable if it cannot be opened at runtime.
- **-dlpreopen *file*:** Links the specified object file or library and adds its symbols to the linked executable.
- **-export-dynamic:** Enables symbols from the linked executable to be resolved with `dlsym(3)` rather than adding them to the symbol table for the linked executable at link time.
- **-export-symbols *symbol_file*:** Limits exported symbols to those specified in the file `SYMBOL_FILE`.

- **-export-symbols-regex *regex*:** Limits exported symbols to those that match the regular expression specified in *regex*.
- **-L*libdir*:** Searches the specified *libdir* directory for required libraries. This can be useful if your application uses static libraries that are already installed in a nonstandard location that is not part of your library load path.
- **-l*name*:** Specifies that the linked executable requires symbols that are located in the installed library with a base or soname of *libname*.
- **-module:** Specifies that the link command builds a dynamically linked library that can be opened using the `dlopen(3)` function.
- **-no-fast-install:** Disables fast-install mode, which means that executables built with this flag will not need relinking in order to be executed from your development area.
- **-no-install:** Links an executable that is designed only to be executed from your development area.
- **-no-undefined:** Verifies that a library that you are linking does not refer to any external, unresolved symbols.
- **-o *output-file*:** Specifies the name of the output file created by the linker.
- **-release *release*:** Specifies that the library was generated by a specific release (*release*) of your software. If you want your linked executable to be binary compatible with versions compiled using other values of *release*, you should use the **-version-info** option instead of this one.
- **-rpath *libdir*:** Specifies that a library that you are linking will eventually be installed in the directory *libdir*.
- **-R *libdir*:** Adds the specified directory (*libdir*) to the runtime path of the programs and libraries that you are linking.
- **-static:** Specifies to not do any dynamic linking of libraries.
- **-version-info *current[:rrevision[:age]]*:** Enables you to specify version information for a library that you are creating. Unlike the **-release** option, libraries and executables linked with different values for the **-version** option maintain binary compatibility with each other.

- **uninstall:** Removes libraries from an installed location. This Libtool mode requires that you specify the name of the command used to uninstall these libraries (typically `/bin/rm`) and any options that you want to provide to that command, as well as the name of the library or libraries that you want to remove. For example:

```
# libtool --mode=uninstall /bin/rm /usr/local/lib/libfib.la
/bin/rm /usr/local/lib/libfib.la /usr/local/lib/libfib.so.1.0.0 \
/usr/local/lib/libfib.so.1 /usr/local/lib/libfib.so \
/usr/local/lib/libfib.a
```

Using Libtool with Autoconf and Automake

Chapter 7 of this book explained how to create the `configure.ac` project configuration file used by Autoconf and the `Makefile.am` project configuration file used by Automake. These sections used the sample application introduced in Chapter 6, consisting of the two source code modules shown in Listings 6-1 and 6-2. For convenience' sake, the sample `configure.ac` file for this application, discussed in detail in the section “Creating `configure.ac` Files” in Chapter 7, is shown again in Listing 8-1. The sample `Makefile.am` file for this application, discussed in detail in the section “Creating `Makefile.am` Files and Other Files Required by Automake” in Chapter 7, is shown again in Listing 8-3.

This section explains how to integrate Libtool into the configuration files used by Autoconf and Automake. In order to demonstrate this, we will demonstrate how to modify the sample Fibonacci application so that the calculation routine provided in the file `calc_fib.c` (shown in Listing 6-2) is actually compiled into a library rather than a standalone object module as it was in Chapter 6. This highlights an impressive advantage of the integration of Libtool and the Autoconf and Automake configuration tools—a change such as this can be done by simply modifying the configuration files used by these applications, and “the right thing happens.”

Listing 8-1 shows the initial `configure.ac` file for the sample application. Listing 8-2 shows this file after the addition of the `AC_PROG_LIBTOOL` macro, which is required in order to tell Autoconf that Libtool is available for use during configuration. In this example, this is the only change that needs to be made to the `configure.ac` file, because most of the work required to change the `calc_fib.c` routine from a stand-alone object file into a library is done by adding Libtool information to the configuration file used by Automake, the file `Makefile.am`.

Listing 8-1. A configure.ac File for the Sample Fibonacci Application

```
AC_PREREQ(2.56)
AC_INIT(Fibonacci, 1.0, wvh@vonhagen.org)
AM_INIT_AUTOMAKE(fibonacci, 1.0, wvh@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Listing 8-2. A configure.ac File with Libtool Integration

```
AC_PREREQ(2.56)
AC_INIT(Fibonacci, 1.0, wvh@vonhagen.org)
AM_INIT_AUTOMAKE(fibonacci, 1.0, wvh@vonhagen.org)
AC_CONFIG_SRCDIR([fibonacci.c])
AC_PROG_CC
AC_PROG_LIBTOOL
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Listing 8-3 shows the simple Automake configuration file that was initially used in the sample application. Because no libraries are being used at this point, this file simply defines the name of the binary that will be generated by the resulting Makefile, defines the source code modules required in order to produce this binary, and identifies a script that can be executed in order to verify that the Fibonacci application was successfully compiled.

Listing 8-3. A Makefile.am File for the Sample Fibonacci Application

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c calc_fib.c
TESTS = run_tests
```

Listing 8-4 shows the Makefile.am file from Listing 8-3 after the addition of the macros and arguments necessary to create a library named libfib from the source module calc_fib.c instead of compiling the latter as a simple object file. This is being done solely to illustrate how to use Libtool to create a simple

library as part of the build process. Creating a library from existing functions is a common occurrence when developing complex applications, and is essentially a mandatory step when you want to be able to call these functions from other applications. Using libraries in complex applications also simplifies deployment and maintenance because the libraries can be updated without requiring recompilation of the main executable.

Listing 8-4. A Makefile.am File with Libtool Integration

```
bin_PROGRAMS = fibonacci
fibonacci_SOURCES = fibonacci.c
fibonacci_DEPENDENCIES = libfib.la
lib_LT_LIBRARIES = libfib.la
libfib_la_SOURCES = calc_fib.c
fibonacci_LDADD = "-dlopen" libfib.la
libfib_la_LDFLAGS = -version-info 1:0:0
```

Comparing Listings 8-3 and 8-4 shows that a relatively small number of changes are required to convert one or more existing code modules from static objects that are directly linked to the resulting executable into a library. For this example, the following is a summary of the changes required:

- Remove the name of the source file from the existing *program_name_SOURCES* macro entry. The sources for the new library are identified in the subsequent *library_name_SOURCES* entry.
- Add the *program_name_DEPENDENCIES* macro entry to indicate that the sample Fibonacci program also depends on the existence of the new library, *libfib.la*. The name used here is the name of the Libtool summary file, not the eventual name of the library that will be used on the platform where you configure and compile your application. As discussed earlier, the Libtool summary file provides platform-specific information about the library that will actually be used on the system where the application is being built. This type of indirection enables you to create platform-independent Makefile.am files.
- Define the library required by the sample application using the *lib_LT_LIBRARIES* macro entry.
- Define the name of any source files required to create the Libtool library using the *library_name_SOURCES* macro entry.

- Use the *program_name_LDADD* macro entry to define any additional arguments that need to be identified to the linker/loader when compiling the sample application. Enclosing the *-dlopen* flag within double quotes and including it here is something of a hack, but most versions of Automake do not correctly support the *program_name_LDFLAGS* variable, where you would think such specifications actually belonged. An upward-compatible solution is to specify them as additional loader arguments and to enclose the option within double quotation marks in order to protect the additional arguments from being processed. Future versions of Automake will enable you to specify linker flags in the more appropriate *program_name_LDFLAGS* variable.
- Add an example of using the *library_name_LDFLAGS* option to specify arguments passed to the linker/loader when building the library. In this case, the option and its associated value specifies the version number of the library.

Once you have modified the *configure.ac* and *Makefile.am* files for your application, you can execute the Autoconf application and run the resulting *configure* script to generate a *Makefile* for your application. For your convenience, the sidebar entitled “Overview of Using Libtool with Autoconf and Automake” provides a summary of the entire Autoconf/Automake process for applications that use libraries and want to quickly and easily generate them using Libtool as part of the auto-configuration process.

Overview of Using Libtool with Autoconf and Automake

The general steps required to create Autoconf and Automake configuration files that automatically invoke Libtool to build the libraries necessary for an application are as follows:

1. Create the *configure.ac* file for your application as described in the Chapter 7 section “Creating *configure.ac* Files,” adding the *AM_INIT_AUTOMAKE* macro, as discussed in the Chapter 7 section “Creating *Makefile.am* Files and Other Files Required by Automake.” Add the *AC_PROG_LIBTOOL* macro after the *AC_PROG_CC* macro. For simple applications, the final *configure.ac* file should look something like the one shown in Listing 8-2.
2. Run the *aclocal* script to create a local file of *m4* macro definitions (such as the definition of the *AM_INIT_AUTOMAKE* macro) for use by Autoconf.
3. Create the *Makefile.am* file for your application as described in the Chapter 7 section “Creating *Makefile.am* Files and Other Files Required by Automake.” For simple applications that use a single library, the final *Makefile.am* file should look something like the one shown in Listing 8-4.

4. Execute the automake program with the `--add-missing` command-line option to copy in any required files that are currently missing from your distribution but for which templates are provided as part of your automake installation. This step also identifies any files that you must create manually. If you do not need or want to create these auxiliary information files, you can use the `touch` program to create empty placeholders for these files.
 5. Execute the libtoolize script with the `--add-missing` command-line option to copy in any files required by Libtool that are currently missing from your distribution but for which templates are provided as part of your Libtool installation.
 6. Run the autoconf program to generate the `Makefile.in` file used by Automake and automatically run Automake to generate the `Makefile` for your application.
 7. Run the resulting `configure` script to generate the `Makefile` for your application and then use the `make` or `make install` command to actually build or build and install your application.
-

Troubleshooting Libtool Problems

Libtool has been used in thousands of development projects and is really quite robust at this point. The section “Command-Line Modes for Libtool Operation” showed some of the more common errors that are easy to make when using Libtool from the command line. This section discusses a few others.

One very common error is trying to link object files and Libtool shared library objects, as in the following example:

```
$ libtool --mode=link gcc -g -O -o libfib.la calc_fib.o
*** Warning: Linking the shared library libfib.la against the non-libtool
*** objects calc_fib.o is not portable!
```

While not fatal on systems that support shared libraries, this is indeed a fatal error on systems that do not support shared libraries. A more appropriate (and correct) Libtool command would be the following:

```
$ libtool --mode=link gcc -g -O -o libfib.o calc_fib.o
/usr/bin/ld -r -o libfib.o calc_fib.o
```

Another common error results from trying to link existing, external libraries in an unknown format with Libtool libraries that may be in another format. This is commonly the case if you are linking your application with libraries that are provided in object or archive format by a software vendor. The theory is that you

can simply link these libraries into your application, but this may conflict with your system's default values for using shared or static libraries. In most cases, the easiest way around this problem is to explicitly create your application using static libraries, since this is the "lowest common denominator" of library file formats.

The Libtool documentation, discussed in the next section, describes some other common problems and provides suggestions for working around or eliminating them.

Getting More Information About Libtool

As discussed earlier in this chapter, Libtool is a GNU project that is hosted at [gnu.org](http://www.gnu.org). For this reason, the central source of information about Libtool, new versions, and so on is the Libtool Home Page, at <http://www.gnu.org/software/libtool/libtool.html>.

If you are very interested in the GNU auto-configuration tools Autoconf and Automake, an excellent book on them is available that also contains a fair amount of information about Libtool: *GNU Autoconf, Automake, and Libtool*, Gary Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor (New Riders, 2000. ISBN: 1-57870-190-2.). The authors and publisher were kind enough to make this book available under the Open Publication License, so you can browse and download the book online, if you wish. The URL for the book's home page is <http://sources.redhat.com/autobook>. You can download it in a variety of formats from the URL <http://sources.redhat.com/autobook/download.html>. Although online documentation is convenient, we strongly encourage you to buy a copy of this book if you are doing serious work with any of the GNU auto-configuration tools; the authors deserve your support.

CHAPTER 9

Troubleshooting GCC

AS THE WORLD'S MOST popular C and C++ compiler, GCC has been compiled and configured for almost every modern computer system with the resources required to execute it. GCC's ability to function as a cross-compiler makes it easy to build a version of GCC that can run on one platform but generate applications targeted for another, as explained elsewhere in this book. This capability even opens up GCC to being used to generate code for systems that may not themselves have the memory or disk-space resources required to run it natively. We have used GCC to build applications for everything from minicomputers to PDAs, and have thus experienced a variety of interesting occurrences that some would call bugs or broken installations, but we prefer to think of as "learning experiences."

In this chapter, we will pass on the fruits of our experience to you. This chapter is designed to help you deal with known problems that exist in GCC when used on various platforms, platform- or implementation-specific details of GCC that may be confusing, and common usage or installation problems. Luckily, the last of these is by far the most common cause of GCC problems, and usage or installation problems are the easiest to correct.

You may never need to read this chapter, which is fine with us and, we are sure, with the GNU GCC team at the Free Software Foundation and elsewhere. However, preventative medicine is good medicine, and you may find it useful to skim through this chapter before you attempt to build or install GCC. Knowing what not to do is often as important as knowing what to do in the first place, though knowing what not to do may be a larger topic. If you experience problems installing or using GCC that fall outside the information provided in this chapter, please let us know by dropping a line to one of our e-mail addresses (as given in the Introduction). We will be happy to help you however we can, and we can then add your experience and its solution to the pool of common knowledge. Be a good GCC Samaritan—perhaps your experience can save other GCC travelers from their own train wreck.

Coping with Known Bugs and Misfeatures

The term *misfeatures* is one of the computer industry's favorite ways of referring to what more cynical people would describe as bugs, broken code, or broken scripts. In computer terms, a *feature* is an attractive capability that is provided by

a software package; a misfeature is therefore a humorous way of identifying an unattractive capability (i.e., a problem).

The release notes and any online version-specific documentation associated with any GCC release are always the best documentation for a list of known problems with a specific release of GCC. Change information about any GCC release is found at the URL <http://gcc.gnu.org/gcc-VERSION/changes.html>, where *VERSION* is the version of GCC that you are using. Information about truly ancient releases of GCC may no longer be available online. If you cannot find information about the version of GCC that you are using, perhaps it is time for an upgrade.

Aside from release notes and change information, the various GCC-related newsgroups (discussed in Chapter 12) generally contain up-to-date problem reports, suggestions, questions, and general entertainment. Within a short while after a new GCC release, the Web becomes an excellent source of information because most of the GCC newsgroups show up on archive sites that are then indexed by Web spiders and made available through popular search engines.

A few specific problems identified in the release notes for GCC 3.3.1 (the latest version available at the time this book was written) are the following:

- The `fixincludes` script that generates GCC-local versions of system header files interacts badly with automounters. If the directory containing your system header files is automounted, it may be unmounted while the `fixincludes` script is running. The easiest way to work around this problem is to make a local copy of `/usr/include` under another name, temporarily modify your automounter maps (configuration files) not to automount `/usr/include`, and then restart `autofs` (on a Linux system) or whatever other automounter daemon and control scripts you are using. If your entire `/usr` directory structure is automounted, such as on some of our Solaris systems, simply put together a scratch machine with a local `/usr` directory, build GCC there, and then install it. You can then use the `tar` application to archive the directory and install it on any system regardless of its automount configuration.

NOTE *The scripts discussed in this section are only relevant if you are building GCC, and they are provided as part of the GCC source code. If you are installing a newer version of GCC from an archive or are simply using the version of GCC that came with your system, the problems noted in these scripts are irrelevant to you.*

- The `fixproto` script may sometimes add prototypes for the `sigsetjmp()` and `siglongjmp()` functions that reference the `jmp_buf` data type before that type is defined. You can resolve this problem by manually editing the file containing the prototypes, placing the `typedef` in front of the prototypes.
- If you specify the `-pedantic-errors` option, GCC may incorrectly give an error message when a function name is specified in an expression involving the C comma operator. This spurious error message can safely be ignored.

Resolving Common Problems

One of the biggest advantages of open source software is that everyone has access to the source code and can therefore put it under a microscope whenever necessary to identify and resolve problems. For GCC, quite probably the most popular open source package of all time, a parallel advantage is that you are rarely the first person to encounter a specific problem. The Web and the GCC documentation are excellent sources for descriptions of common problems encountered when using GCC and, more importantly, how to solve them.

This section lists some common problems you may run into when using GCC on a variety of platforms and in a variety of ways, and provides solutions or workarounds. It is important to recognize that the fact that one can identify common problems does not mean that GCC itself is laden with problems. As you can see from the number of command-line options discussed in Chapter 11 and the advanced usage models discussed in Chapter 4, GCC is one of the world's most flexible programs. The number of knobs that you can turn on GCC sometime leads to problems by accidentally invoking conflicting options, environment variables settings, and so on.

Problems Executing GCC or Programs Compiled with It

It is very rare that a version of GCC is installed in a way that all users of a system cannot execute it. Installing GCC incorrectly could actually cause this problem, in which case you would see a message like the following:

```
bash: gcc: Permission denied
```

If you get this message, you will have to contact your system administrator to see if GCC was intentionally installed with restrictive file permissions or ACLs.

More commonly, problems executing GCC on a system are related to making sure that you are executing the right version of GCC, as explained in the following two sections.

Using Multiple Versions of GCC on a Single System

If you are running an open source operating system such as Linux, FreeBSD, NetBSD, or one of the others, there is a good chance that your system came with a version of GCC that is included as part of your operating system's default installation. This is incredibly useful for most day-to-day purposes, but you may eventually want to upgrade the version of GCC available on your system, often because a new version adds features or capabilities that were not present in an earlier version.

Having multiple versions of GCC installed on a single system is not usually a problem. As explained in Chapters 1 and 2, each version of GCC is consistent and contains internal references to the location where it was intended to be installed. The GCC binary (`gcc`) is primarily a driver program, using specification files (known as *spec files*) to determine the options with which to run the programs associated with each stage of the compilation process (the C preprocessor, linker, loader, and so on). However, the GCC binary itself (`gcc`) currently contains hard-coded strings that identify the directory where that version of GCC expects to find these programs.

Unfortunately, GCC's internal consistency cannot guarantee that you are executing the “right” version of GCC. If you have multiple versions of GCC installed on your system and they were all installed using the default names for GCC, make sure that your PATH environment variable is set so that the directory where the version you are trying to execute is listed before any directory containing another version of GCC. You can also execute GCC by using its full pathname, as in `/usr/local/gcc331/bin/gcc`. If installed correctly, each GCC binary will execute the versions of all of the other programs used during the compilation process that were built at the same time as that version of GCC.

Setting the value of your PATH environment variable is done differently depending on the command shell that you are running. If you are using the Bourne-Again shell (`bash`), you can set the value of this variable by using a command like the following:

```
$ export PATH=new-directory:${PATH}
```

Replace *new-directory* with the full pathname of the directory containing the version of GCC that you actually mean to run, as in the following example, which puts the directory `/usr/local/bin` at the beginning of your existing search PATH:

```
$ export PATH=/usr/local/bin:${PATH}
```

If you are using the C shell (`csh`) or the TOPS-20/TENEX C shell (`tcsh`), the syntax for updating the PATH environment variable is slightly different.

```
% setenv PATH /usr/local/bin:${PATH}
```

TIP *On most Linux and other Unix-like systems, you can find all versions of GCC that are currently present in any directory in your PATH by executing the whereis gcc command, which will return something like the following:*

```
$ whereis gcc
gcc: /usr/bin/gcc /usr/local/tivo-mips/bin/gcc /usr/local/bin/gcc
```

You can then use the which gcc command to identify which one is located first in your PATH, and adjust your PATH as necessary.

TIP *If you have built or simply installed multiple versions of GCC and encounter problems after installing a new one, you should make sure that the GCC spec file for the new compiler does not accidentally reference the files associated with an old one. The spec file for each version of GCC is typically located in the lib/gcc-lib subdirectory of the same directory tree where you installed the binaries. Use a text editor to examine the spec file in this directory and verify that any paths that it contains refer to the same directory hierarchy where the new binaries were installed.*

Problems Executing Files Compiled with GCC

After successfully compiling a program using GCC, the message “cannot execute binary file” is depressing in its simplicity. Luckily, its cause is usually quite simple. This message typically means that the binaries produced by the version of GCC that you executed are actually targeted for a different type of system than the one that you are running on. This means that you are running a version of GCC that was configured as a cross-compiler, which runs on one type of system but produces binaries designed to run on another.

Most cross-compilers are built and installed using prefixes that identify the type of system on which they are designed to run. For example, the version of GCC named ppc8xx-linux-gcc is designed to run on an x86 system but produces binaries that are intended to execute on systems with PowerPC 8xx-family processors. However, for convenience’ sake, people often create aliases or symbolic links named GCC that point to a specific cross-compiler. If you see the message “cannot execute binary file”, try the following:

- Check that you are not picking up a bad alias by using the alias gcc command to see if your gcc command is an alias for another version of GCC. If you see a message like “alias: gcc: not found”, this is not the problem.

- Verify that the binary produced by GCC is actually compiled for the architecture on which you are running by executing the file *filename* command. If its output does not match the type of system on which you are running, you may have accidentally invoked a version of GCC that was built as a cross-compiler for another type of system.
- Verify which version of GCC you are actually running by executing the `which gcc` command. This will return the first version of GCC that is currently found in your path. Make sure that this is the version of GCC that you actually want to run. If it is not, you can adjust the value of your PATH environment variable so that the directory containing the version that you wanted to run is found first, as described in the previous section.

Running Out of Memory When Using GCC

This problem is uncommon, and is usually only seen when using GCC on System V-based systems with broken versions of `malloc()`, such as SCO Unix. In general, we hope that you are not using SCO Unix for anything, but if you are stuck with it for some reason, the easiest solution is to provide a working (i.e., nonsystem) version of `malloc`, the Linux/Unix memory allocation function. A list of popular `malloc()` replacements is available at the URL <http://www.cs.colorado.edu/~zorn/Malloc.html>. Mike Haertel's GNU Malloc is a good choice. After building and installing whichever `malloc()` you have chosen, you can relink GCC, specifying the name of your new version of `malloc()` using a command-line argument such as the following:

```
MALLOC=/usr/local/lib/libgmalloc.a
```

TIP *If you have installed and built GCC on your system, you can simply recompile the file gmalloc.c and pass this as an argument when relinking GCC:*

```
MALLOC=gmalloc.o
```

Moving GCC After Installation

To put it succinctly, *do not attempt to move GCC and associated programs to another directory after building and installing them*. It is always better to rebuild GCC than to try to trick it into running from a directory other than the one for which it was built.

NOTE This section generally applies to using GCC on actual Linux systems. If you are using Cygwin, GCC is built so that it can be moved to other directories (i.e., it is relocatable).

As mentioned previously, the GCC binary (`gcc`) is primarily a driver program, using spec files to determine how to run the programs associated with each stage of the compilation process (the C preprocessor, linker, loader, and so on). However, `gcc` itself currently contains hard-coded strings that identify the directory where that version of GCC expects to find these programs.

If the `gcc` binary cannot find one or more of the programs or libraries that it requires, you will see messages such as the following whenever you try to compile a program using GCC:

```
gcc: installation problem, cannot exec 'cc1': No such file or directory
```

This specific message means that the `gcc` binary was unable to find the program `cc1`, which is the first stage of the C preprocessor, and is located in GCC's library directory, `gcc-lib`. If you are truly out of space somewhere, you can move GCC's library directory to another location and then create a symbolic link from the correct location to the new one. For newer versions of GCC, this directory consumes around 35MB of disk space, which is not a huge amount of space by today's standards.

The worst thing about moving something and using symbolic links to glue things back together is that you will almost certainly forget that you have done this at some point. This makes it easy to delete a random directory somewhere, forgetting that symbolic links elsewhere point to it. Unless your backup strategy is a lot more robust than ours, you will almost certainly mess yourself up by doing this. The unfortunate thing about breaking your current GCC installation is that this prevents you from building another version of GCC, forcing you to resort to downloading and installing precompiled binaries in package or gzipped tar file formats.

General Issues in Mixing GNU and Other Toolchains

Most GCC users, including the authors of this book, find that GCC's toolchain is a complete solution for all of their compilation needs. However, there may be times when you need to mix GCC with components of the compilation food chain that are provided by third parties. This is often the case when using optimized tools from third parties, or when you yourself are developing an enhanced preprocessor or other part of the compiler toolchain. The following are general

suggestions for resolving problems when trying to integrate third-party tools with GCC:

- Check the release notes and documents for both the foreign toolchain and the version of GCC that you are using, in order to identify any known interoperability issues. The next section of this chapter discusses some specific known problems.
- Search the Web for information about known interoperability problems.
- Verify that the problem is actually where you think it is. If you are using a version of the `make` program other than GNU `make`, ensure that your version of `make` supports integrating other tools and passing mandatory command-line options to them correctly. If you are not using GNU `make` and your version of the `make` program does not do this correctly, perhaps you should consider performing a free upgrade for your existing `make` program by downloading and installing GNU `make`.
- If you are replacing portions of the GCC toolchain with commercial or other tools and experience problems, first make sure that you are correctly integrating the two. All commercial toolchains support their own sets of command-line options, and may invoke subcomponents (such as an assembler or preprocessor) with options that you may not be aware of because they are supplied internally.
- Use verbose modes for both GCC and whatever tools you are integrating with them. You may be able to spot missing command-line options that you can then pass to the appropriate component of the toolchain by using the appropriate version of GCC's `-X` option. GCC provides the `-Xa` option to pass specific arguments to the assembler, the `-Xl` option to pass arguments to the linker, and the `-Xp` option to pass options to the preprocessor. Once you have found a combination that works for you, you can encapsulate these options by adding them to the contents of your Makefile or shell's `CCFLAGS` environment variable.

- If you cannot use primary GCC command-line options to “do the right thing” when invoking commercial or other non-GCC toolchain components, you may be able to modify environment variables in your Makefile to add new command-line options to those passed to the appropriate tool during the compilation process. If you are using GNU make, each of the components in the GCC toolchain (preprocessor, assembler, linker) has a corresponding `FLAGS` environment variable (`ASFLAGS`, `LDFLAGS`, `CPPFLAGS`) that can be set in your Makefiles in order to specify command-line options that are always supplied to the appropriate tool. If you are not using GNU make, your version of the `make` program does not support these flags, and if you are using GCC, perhaps you should consider upgrading your existing `make` program to GNU `make`.

TIP *If you modify your Makefile, you can execute the `make -n` command to show the commands that would be executed to recompile your application, without actually executing them.*

- If the version of the `make` program that you are using does not support modifying the flags that you are passing to each phase of the compilation process, it may still enable you to set Makefile variables for the tools themselves. If so, you can often include specific sets of options in those application definitions. For example, GNU `make` supports Makefile environment variables that identify the assembler (`AS`) and the preprocessor (`CXX`). You may be able to set these variables to the name of the binary for your third-party software, plus whatever command-line options are necessary to cause your third-party software to “play nice” with GCC.
- If you are still experiencing problems, talk to the vendor. They may not be happy to know that you want to use their expensive software in conjunction with a free tool, but you have paid for support. Depending on where the problem seems to appear, you can use some combination of GCC’s many debugging options (`-x`, `-c`, `-save-temp`, and `-E` come to mind) to preserve temporary files or halt the compilation process at the point at which the problem materializes. For example, if your third-party assembler cannot correctly assemble code generated by GCC, you can halt GCC’s compilation process just before entering the assembler or simply preserve the temporary input files used by the GNU assembler, and provide that to your vendor so they can attempt to identify the problem.

- Make sure that the third-party tool that you are using conforms to the same C or C++ standards as the GCC defaults. See Chapter 11 for a discussion of using the `-std` command-line option to specify a different standard for GCC to conform to during compilation.
- Similarly, make sure you are not using GNU C or C++ extensions that other tools may not recognize.

The preceding is a general list of interoperability problems. The next section provides specific examples of interoperability and integration problems that are discussed to varying degrees in the GCC documentation itself.

Specific Compatibility Problems in Mixing GCC with Other Tools

This section lists various difficulties encountered in using GCC together with other compilers or with the assemblers, linkers, libraries, and debuggers on certain systems. The items discussed in this section were all listed in the documentation provided with GCC 3.3.1. If you are experiencing problems using GCC on one platform but not another, make sure that you check GCC's info file (by using the command `info gcc` after installing GCC). Using `info` is explained in Chapter 10.

Following are some known interoperability problems on specific platforms:

- On AIX systems, when using the IBM assembler, you may occasionally receive errors from the AIX assembler complaining about displacements that are too large. You can usually eliminate these by making your function smaller or by using the GNU Assembler.
- On AIX systems, when using the IBM assembler, you cannot use the dollar sign in identifiers even if you specify the `-fdollars-in-identifiers` option due to limitations in the assembler. You can resolve this by not using this symbol in identifiers, or by using the GNU Assembler.
- On AIX systems, when compiling C++ applications, shared libraries and dynamic linking do not merge global symbols between libraries and applications when you are linking with `libstdc++.a`. A C++ application linked with AIX system libraries must include the `-Wl,-brtl` option on the linker command line.

- On AIX systems, when compiling C++ applications, an application can interpose its own definition of functions for functions invoked by `libstdc++.a` with “runtime linking” enabled. To enable this to work correctly, applications that depend on this feature must be linked with the runtime-linking option discussed in the previous bullet, and must also export the function. The correct set of link options to use in this case is `-Wl,-brtl,-bE:exportfile`.
- On AIX systems, when compiling national language support (NLS) enabled applications, you may need to set the `LANG` environment variable to `C` or `En_US` in order to get applications compiled with GCC to link with system libraries or assemble correctly.
- G++ does not do name mangling in the same way as other C++ compilers. For this reason, object files compiled with another C++ compiler cannot be used with object files compiled using G++. You must recompile your entire application using one or the other. Though this may cause you a bit of work, it was done intentionally to protect you from more subtle problems related to the internal details of your C++ compiler’s implementation. If G++ used standard name encoding, programs would link against libraries built using or provided with other compilers, but would randomly crash when executed. Using a unique name-encoding mechanism enables incompatible libraries to be detected at link time, rather than at runtime.
- Older GDB versions sometimes fail to read the output of GCC versions 2 and later. If you have trouble, make sure that you are using an up-to-date version of GDB (version 5.3 at the time that this book was written).
- If you experience problems using older debuggers such as `dbx` with programs compiled using GCC, switch to GDB. The pain of the GDB learning curve is much less than the pain of random incompatibilities—plus it is a good career investment.
- If you experience problems using profiling on BSD systems, including some versions of Ultrix, the static variable destructors used in G++ may not execute correctly or at all.
- If you are using a system that requires position-independent code (PIC), the GNU Assembler does not support PIC. To generate PIC code, you must use some other assembler (usually `/bin/as` on a Unix-like system).

- If you are linking newly compiled code with object files produced by older versions of GCC, you may encounter problems due to different binary formats. You may need to recompile your old object files in order to link successfully.
- The C compilers on some SGI systems automatically expand the `-lg1_s` option into `-lg1_s -lX11_s -lc_s`. This may cause some IRIX programs that happened to build correctly to fail when you first try to recompile them under GCC. GCC does not do this expansion—you must specify all three options explicitly.
- On SPARC systems, GCC aligns all values of type `double` on an 8-byte boundary, and expects every `double` to be aligned in the same fashion. The Sun compiler usually aligns `double` values on 8-byte boundaries, with the exception of function arguments of type `double`. The easiest way around this problem without extensive code changes is to recompile your entire application with GCC. The GCC documentation provides sample code to work around this problem, but patching a torn pair of pants does not make them new—plus you have to maintain the patches.
- On Solaris systems, the `malloc` function in the `libmalloc.a` library may allocate memory that is only aligned along 4-byte boundaries. Unfortunately, GCC on SPARCs assumes that doubles are aligned along 8-byte boundaries, which may cause a fatal signal if doubles are stored in memory allocated by Sun's `libmalloc.a` library. The only easy solution to this problem is to use `malloc` and related functions from `libc.a` rather than using the `libmalloc.a` library.

Problems When Using Optimization

The key problem with optimization is that sometimes it works too well. When using optimization, there will always be a certain amount of disagreement between an optimized executable and your source code. For example, you may find that local variables cannot be traced or examined when you are debugging an optimized application. This is often caused by the fact that GCC may have optimized the variable out of existence.

In general, you rarely need to use optimization when you are still in the debugging phase of application development. If an application works correctly before optimization but no longer works correctly when you specify optimization levels such as `-O2` or `-O3`, you should make sure that you are actually `malloc`'ing every data structure that you access through a pointer. Nonoptimized programs may accidentally work if they access memory that is actually associated with other data structures that are not currently being used. By minimizing

allocation and eliminating unnecessary or unused structures, optimization may cause “working” programs to fail in this circumstance.

Problems with Include Files or Libraries

As part of the process of building GCC, GCC runs shell scripts that make local copies of system header files that GCC believes exhibit various types of problems. Most target systems have some number of header files that will not work with GCC because they have bugs, are incompatible with ISO C, or were designed to depend on special features provided by other compilers. The best known of the scripts run by GCC is called `fixincludes`, for fairly obvious reasons.

After making local copies of these files and updating them, GCC uses these updated system header files instead of the generic system header files. If you subsequently install updated versions of your system header files, an existing GCC installation does not know that the original files have changed. GCC will therefore continue to use its own copies of system header files that may no longer be appropriate to the kernel that your system is running or the libraries that the operating system is currently using.

To resolve this sort of problem, you must reinstall GCC and cause it to regenerate its local copies of the system header files. To do this, change to the directory in which you built GCC and delete the files named `stmp-fixinc` and `stmp-headers` and the entire include subdirectory. You can then rerun the `make install` command, and GCC will re-create its local copies of problematic header files. If this process generates errors, you will need to obtain a newer copy of GCC. Your operating system vendor may have made changes that are extensive enough to make them incompatible with the shell scripts provided with your current version of GCC.

NOTE *On some older operating systems, such as SunOS 4, header file directories contain machine-specific symbolic links in certain places. This makes it possible to share most of the header files among hosts running the same version of SunOS 4 on different machine models. However, this also means that the directory of fixed header files is good only for the machine model on which it was built. For the most part, this is not an issue because only kernel-level programs should care about the differences between different types of machines. If absolutely necessary, you can build separate sets of fixed header files for various types of machines, but you will have to do this manually. Check the Web for information about scripts that various old-time Sun users have created to do this for you.*

Working with existing system libraries is very different from working with text files such as include files. GCC attempts to be a conforming, freestanding

compiler suite. However, beyond the library facilities required by GCC internally, the operating system vendor typically supplies the rest of the C library. If a vendor's C library does not conform to the C standards, programs compiled with GCC that use system libraries may display strange warnings, especially if you use an option such as `-Wall` to make GCC more sensitive. If you see multiple warnings and existing applications compiled with the vendor's C compiler exhibit strange behavior when compiled with GCC, you should strongly consider using the GNU C library (commonly called Glibc), which is available as a separate package from the GNU Web site (<http://www.gnu.org/software/libc/libc.html>). Glibc provides ISO C, POSIX, BSD, System V, and X/Open compatibility for Linux systems (and for the GNU project's own HURD-based operating system). If you do not want to switch to this or are not running Linux or HURD, your only other alternative is to petition your operating system vendor for newer libraries. You have our sympathy.

Mysterious Warning and Error Messages

The GNU compiler produces two kinds of diagnostics: errors and warnings. Each of these has a different purpose:

- Errors report problems that make it impossible to compile your program. GCC reports errors with the source filename and line number where the problem is apparent.
- Warnings report other unusual conditions in your code that may indicate a problem, although compilation can (and will) proceed. Warning messages also report the source filename and line number, but include the leading text string “warning:” to distinguish them from error messages.

Warnings indicate points in your application that you should verify. This may be due to using questionable syntax, obsolete features, or nonstandard features of GNU C or C++. Many warnings are only issued if you specify the “right” one of GCC’s `-W` command-line options (or the `-Wall` option, which turns on a popular selection of warnings).

GCC always tries to compile your program if this is at all possible. However, in some cases, the C and C++ standards specify that certain extensions are forbidden. A conforming compiler such as GCC must issue a diagnostic when these extensions are encountered. GCC’s `-pedantic` option causes GCC to issue warnings in such cases. Using the stricter `-pedantic-errors` option converts such diagnostic warnings into errors that will cause compilation to fail at such points. Only those non-ISO constructs that are required to be flagged by a conforming compiler will generate warnings or errors.

To increase the GCC’s tolerance for outdated or older features in existing applications, you can always use the `-traditional` option, which attempts to

make GCC behave like a traditional C compiler, following the C language syntax described in the book *The C Programming Language*, Brian Kernighan and Dennis Ritchie (Prentice Hall, 1988. ISBN: 0-131-10362-8.). (Kernighan and Ritchie represent the K and R in K&R.) The next section discusses differences and subtleties between the GCC and K&R C compilers.

Incompatibilities Between GNU C and K&R C

This section discusses some of the more significant incompatibilities between GNU C and K&R C. In real life, the chances of encountering these problems are small unless you are recompiling a huge legacy application that was written to work with a K&R C compiler or if, like one of the authors of this book, you still automatically think of ANSI C as “that new thing.”

NOTE *As mentioned in the previous section, using GCC's -traditional command-line option causes GCC to masquerade as a traditional Kernighan and Ritchie C compiler. This emulation is necessarily incomplete, but it is as close as possible to the original.*

Some of the more significant incompatibilities between GNU C (the default standard used by GCC) and K&R (non-ISO) versions of C are the following:

- GCC normally makes string constants read-only. If several identical string constants are used at various points in an application, GCC only stores one copy of the string. One consequence of this is that, by default, you cannot call the `mktemp()` function with an argument that is a string constant, because `mktemp()` always modifies the string that its argument points to. Another consequence is that functions such as `fscanf()`, `scanf()`, and `sscanf()` cannot be used with string constants as their format strings, because these functions also attempt to write to the format string.
- The best solution to these situations is to change the program to use arrays of character variables that are then initialized from string constants. To be kind, GCC provides the `-fwritable-strings` option, which causes GCC to handle string constants in the traditional, writable fashion. This option is automatically activated if you supply the `-traditional` option.
- To GCC, the value `-2147483648` is positive because `2147483648` cannot fit in an `int` data type. This value is therefore stored in an `unsigned long int`, as per the ISO C rules.

- GCC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GCC:

```
#define foo(a) "a"
```

will produce the actual string "*a*" regardless of the value of *a*. Using the `-traditional` option causes GCC to do macro argument substitution in the traditional (old-fashioned) non-ISO way.

- When you use the `setjmp()` and `longjmp()` functions, the only automatic variables guaranteed to remain valid are those declared as `volatile`. This is a consequence of automatic register allocation. If you use the `-W` option with the `-O` option, GCC will display a warning when it thinks you may be depending on nonvolatile data in conjunction with the use of these functions. Specifying GCC's `-traditional` option causes GCC to put variables on the stack in functions that call `setjmp()`, rather than in registers. This results in the behavior found in traditional C compilers.
- Programs that use preprocessing directives in the middle of macro arguments do not work with GCC. ISO C does not permit such a construct, and is poor form anyway. The `-traditional` option will not help you here.
- Declarations of external variables and functions within a block apply only to the block containing the declaration—they have the same scope as any other declaration in the same place. Using the `-traditional` option causes GCC to treat all external declarations as globals, as in traditional C compilers.
- In traditional C, you can combine type modifiers with preexisting `typedef`'ed names, as in the following example:

```
typedef int foo;  
typedef long foo bar;
```

- In ISO C, this is not allowed. The `-traditional` option cannot change this type of GCC behavior because this grammar rule is expressed in high-level language (Bison, in this case) rather than in C code.
- GCC treats all characters of an identifier as significant, rather than only the first eight characters used by K&R C. This is true even when the `-traditional` option is used.
- GCC does not allow whitespace in the middle of compound assignment operators such as `+=`.

- GCC complains about unterminated character constants inside of a pre-processing conditional that fails if that conditional contains an English comment that uses an apostrophe. The `-traditional` option suppresses these error messages, though simply enclosing the comment inside comment delimiters will cause the error to disappear. The following is an example:

```
#if 0
    You cannot expect this to work.
#endif
```

- Many older C programs contain declarations such as `long time();`, which was fine when system header files did not declare this function. However, systems with ISO C headers declare `time()` to return `time_t`. If this is not the same size as `long`, you will receive an error message. To solve this problem, include an appropriate system header file (`<time.h>`) and either remove the local definitions of `time()` or change them to use `time_t` as the return type of the `time()` function.
- When compiling functions that return structures or unions, GCC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GCC cannot call a structure-returning function compiled with an older compiler such as PCC, and vice versa. You can tell GCC to use a compatible convention for all functions that return structures or unions by specifying the `-fpcc-struct-return` option.

Abuse of the `_STDC_` Definition

Though more a stylistic or conceptual problem than a GCC issue, the meaning of `_STDC_` is abused frequently enough that it deserves a short discussion here.

Programmers normally use conditionals on whether `_STDC_` is defined in order to determine whether it is safe to use features of ISO C such as function prototypes or ISO token concatenation. Because vanilla GCC supports all the features of ISO C, all such conditionals should test as true, even when the `-ansi` option is not specified. Therefore, GCC currently defines `_STDC_` as long as you do not specify the `-traditional` option. GCC defines `_STRICT_ANSI_` if you specify the `-ansi` option or a `-std` option specifying strict conformance to some version of ISO C.

Unfortunately, many people seem to assume that `_STDC_` can also be used to check for the availability of certain library facilities. This is actually incorrect in an ISO C program, because the ISO C standard says that a conforming free-standing language implementation should define `_STDC_` even though it does

not have the library facilities. The command `gcc -ansi -pedantic` is a conforming freestanding implementation, and it is therefore required to define `_STDC_`, even though it does not come with an ISO C library.

TIP You should not automatically undefine `_STDC_` in a C++ application. Many header files are written to provide prototypes in ISO C but not in traditional C. Most of these header files can be used in C++ applications without any changes if `_STDC_` is defined. If `_STDC_` is undefined in a C++ application, they will all fail, and will need to be changed to contain explicit tests for C++.

Resolving Build and Installation Problems

Having been successfully installed and used on a huge number of platforms for the last 15 years or so, GCC's build and installation processes are quite robust. GCC requires a significant amount of disk space during the build and test process, especially if you build and test all of the compilers (C, C++, ADA, Java, Fortran) that are currently included in the GNU Compiler Collection. One of the most common problems encountered when building and installing GCC is simply running out of disk space.

Before you begin to build and install GCC, think about the tools that you are using to build it. For example, it is easier and safer to build GCC using GNU `make` rather than any system-specific version of `make`. Similarly, GCC is best built with GCC. Though this seems like a paradox, it really is not. When boot-strapping GCC on Solaris boxes without the official (paid) Solaris C compiler, we usually download a prebuilt binary of GCC for Solaris (from a site like <http://www.sunfreeware.com>), download the latest GCC source code, and then use the former to build the latter. Maybe we are just paranoid . . .

NOTE Using a compiler to compile itself is an impressive feat, though conceptually an equally impressive exercise in recursion.

If you have previously built and installed GCC and then encounter problems when rebuilding it later, make sure that you did not configure GCC in your primary directory unless you intend to always build it there. The GCC build and installation documents suggest that you create a separate build directory and then execute the `configure` script from that directory as `../gcc-VERSION/configure`. This will fail if you have previously configured GCC in the primary directory.

As mentioned earlier, a common error encountered when building GCC on NFS-dependent systems is that the `fixincludes` script that generates GCC-local versions of system header files may not work correctly if the directory containing

your system header files is automounted—it may be unmounted while the fixincludes script is running. The easiest way to work around this problem is to make a local copy of /usr/include under another name, temporarily modify your automounter maps (configuration files) not to automount /usr/include, and then restart autofs (on a Linux system) or whatever other automounter daemon and control scripts you are using. If your entire /usr directory structure is automounted, such as on some of our Solaris systems, you should just put together a scratch machine with a local /usr directory and build GCC there. After installing it, archive the installation directory and install GCC onto other systems from the archive.

Building and installing GCC has been done by thousands of users everywhere. We have built GCC on everything from Apollos to our home Linux boxes. If you encounter a problem, check the release notes, the README file, and the installation documentation. If you still have problems, feel free to ask us at the e-mail addresses given in the Introduction, or visit <http://gcc.gnu.org>, or check the news archives at <http://www.google.com>. If you have experienced a problem, chances are that someone else has, too.

CHAPTER 10

Using GCC's Online Help

FOR BETTER OR WORSE, the FSF uses a nonstandard online help format, GNU Info, to document its software. Info is one of the utilities people love to hate. Even though one can credibly argue that GNU Info is a more powerful and flexible system for documenting software than the ubiquitous man page, Info is dramatically different from the beloved Unix manual page, a difference that has kept it from becoming popular. Indeed, if you want to start a flame war on your favorite mailing list, don your asbestos underwear and post a message stating that Info is better than man. Seriously, because GNU Info is not very popular, most people do not now how to use it. Unfortunately, most of GCC's excellent documentation is maintained in GNU Info format, so to get the most out of the GCC's own documentation, you have to know how to use GNU Info—this chapter teaches you how to do so. We think you will find that it is not nearly as difficult to use as it seems.

What Is GNU Info?

What is GNU Info? The simple answer is that GNU Info is a hypertext online help system, intended by the FSF to replace the traditional (and, we dare say, the standard) Unix manual (or *man*) page. GNU Info enables online help systems to use tables of contents, cross-references, and indices without requiring the overhead of a graphical HTML browser. Info was originally designed long before the explosion of the Internet and before the World Wide Web was ubiquitous. Because of its hypertext abilities, GNU Info makes it easy to jump from one reference page to another that contains related information and then to return to the original page. This hyperlinking is a clear advantage over the traditional Unix manual page. Unfortunately, the Info user interface, which bears a close resemblance to the Emacs user interface, is substantially different from and far more complicated than the interface used by traditional Unix manual page viewers, which are usually some variant of the `more` or `less` text viewers. GNU's stubborn refusal to make traditional manual pages available for most of their software (with some notable exceptions, of course—the Bash manual pages come to mind) and Info's different interface has prevented it from becoming an accepted, popular replacement for the more limited but easier to use Unix manual page. So, this short chapter will show you how to use GNU Info while staying out of the Info versus man dispute.

NOTE We must confess that we started writing this chapter from the perspective of confirmed manual page advocates and committed Info haters. However, by the time we completed the first draft, our opinion of GNU Info had changed. Although we still wish that the GNU project would make both manual and Info pages available instead of forcing users to use Info, we have found that Info gives users greater capabilities for searching and viewing related information than the current man implementations. We still use `man foo` to get quick reference information, but we no longer swear a blue streak if we have to use GNU Info.

GNU Info is more than just a hypertext online help system, though. It is really a complete document production system. The `info` command is a program that knows how to view and navigate Info files. *Info files*, in turn, are Texinfo (pronounced “teck-info”) source files that have been processed by the GNU `makeinfo` command in order to create Info files that the `info` command can read. *Texinfo source files* are plain ASCII text files containing so-called `@-commands`, characters and words prefixed with `@`, and a few other characters with special meanings. The `@-commands` and other special characters provide instructions to formatting and typesetting programs about how to format and typeset the text.

One of the attractions of Texinfo is that a single Texinfo source file can be used to produce Info files for use with GNU Info, typeset files ready for printing, and HTML output that you can browse with any Web browser. For the curious, Listing 10-1 shows a small sample of Texinfo source code taken from the `linuxthreads.texi` file. This short example of Texinfo source code describes the `pthread_exit()` function in the `linuxthreads` library. Listing 10-2 shows the same Texinfo source code after it has been rendered into an Info file and displayed using the Info browser, `info`.

Listing 10-1. Sample Texinfo Source Code

```
@comment pthread.h
@comment POSIX
@deftypefun void pthread_exit (void *@var{retval})
@code{pthread_exit} terminates the execution of the calling thread. All
cleanup handlers (@pxref{Cleanup Handlers}) that have been set for the
calling thread with @code{pthread_cleanup_push} are executed in reverse
order (the most recently pushed handler is executed first). Finalization
functions for thread-specific data are then called for all keys that
have non-@code{NULL} values associated with them in the calling thread
(@pxref{Thread-Specific Data}). Finally, execution of the calling
thread is stopped.
```

Listing 10-2. Texinfo Source Code After Rendering

- Function: void pthread_exit (void *RETVAL)

'pthread_exit' terminates the execution of the calling thread. All cleanup handlers (*note Cleanup Handlers::) that have been set for the calling thread with 'pthread_cleanup_push' are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non-'NULL' values associated with them in the calling thread (*note Thread-Specific Data::).

Finally, execution of the calling thread is stopped.

We strongly recommend that you learn how to use the Info interface because a great deal of valuable information, especially about GCC, is only accessible in Info files. The next section, “Getting Started, or Instructions for the Impatient,” should be sufficient to get you started. Subsequent sections go into greater detail and offer variations and extensions of the techniques discussed in the next section.

Getting Started, or Instructions for the Impatient

To start using GNU Info, type **info program**, replacing **program** with the name of the program that interests you. This being a book about GCC, you might try

```
$ info gcc
```

To exit info, type **q** to return to the command prompt.

If you are staring at an Info screen right now and just want to get started, Table 10-1 shows you just enough Info navigation commands to help you find your way around the Info help system. The notation Ctrl-x means to press and hold the Control key while pressing the x key. Meta-x means to press and hold the Meta key (also known as the Alt key) while pressing the x key. Finally, Ctrl-x n means to press and hold the Control key while pressing the x key, release them, and then press the n key.

Table 10-1. Basic Info Navigation Commands

Key or Key Combination	Description
Ctrl-n	Moves the cursor to the next line down
Down	Moves the cursor to the next line down
Ctrl-p	Moves the cursor to the previous line
Up	Moves the cursor to the previous line
Ctrl-b	Moves the cursor to the left one character
Left	Moves the cursor to the left one character
Ctrl-f	Moves the cursor to the right one character
Right	Moves the cursor to the right one character
Ctrl-a	Moves the cursor to the beginning of the current line
Ctrl-e	Moves the cursor to the end of the current line
Spacebar	Scrolls forward (down) one screen
Ctrl-v	Scrolls forward (down) one screen
Page Down	Scrolls forward (down) one screen
Meta-v	Scrolls backward (up) one screen
Page Up	Scrolls backward (up) one screen
b	Moves the cursor to the beginning of the current node
e	Moves the cursor to the end of the current node
l	Moves to the most recently visited node
n	Moves to the next node
p	Moves to the previous node
u	Moves up a node
s	Performs a case-insensitive search for a string
S	Performs a case-sensitive search for a string
Ctrl-x n	Repeats the last search
Ctrl-x N	Repeats the last search in the opposite direction

TIP If the Alt key combinations described in this section do not work on your system, you can press the Esc key instead. So, for example, Meta-v (or Alt-v) can also be invoked as Esc-v. Notice that if you use the Esc key, you do not have to press and hold it as you do when using the Alt key.

Those of you who are Emacs users will recognize the keystrokes in Table 10-1 as Emacs keybindings. If you need more help than the keystroke reference in Table 10-1 offers you, read the next few sections. If you get totally lost or confused, just type **q** to exit the info reader and return to the command prompt.

The Beginner's Guide to Using GNU Info

This section explains the concepts and commands that cover 80 percent of GNU Info usage. By the time you have completed this section of the chapter, you will also know more about GNU Info and how to use it than almost everyone else who uses GNU software. The other 20 percent of GNU Info commands, covered in the section titled "Stupid Info Tricks," explore advanced features and techniques whose use arguably constitute Info mastery. Okay, perhaps *mastery* is a bit over the top, but the "Stupid Info Tricks" section at the end of the chapter does describe sophisticated methods you will not use very often.

Anatomy of a GNU Info Screen

Most of the discussion that follows assumes you know what a typical Info file looks like when displayed using the `info` command. This section describes the standard components of an Info screen. Figure 10-1 shows a typical Info screen (as luck would have it, it is the top-level node of the GCC Info file), with the various elements described in the text identified by callouts.

```

File: gcc.info, Node: Top, Next: G++ and GCC, Up: (DIR)
Introduction
*****  

  This manual documents how to use the GNU compilers, as well as their
  features and incompatibilities, and how to report bugs. It corresponds
  to GCC version 3.2.3. The internals of the GNU compilers, including
  how to port them to new targets and some information about how to write
  front ends for new languages, are documented in a separate manual.
  *Note Introduction: (gccint)top.  

* Menu:  

  • G++ and GCC:: You can compile C or C++ programs.  

  • Standards:: Language standards supported by GCC.  

  • Invoking GCC:: Command options supported by 'gcc'.  

  • C Implementation:: How GCC implements the ISO C specification.  

  • C Extensions:: GNU extensions to the C language family.  

  • C++ Extensions:: GNU extensions to the C++ language.  

  • Objective-C:: GNU Objective-C runtime features.  

  • Compatibility:: Binary Compatibility  

--zz-Info: (gcc.info.gz)Top, 40 lines --Top-- Subfile: gcc.info-1.gz  

Welcome to Info version 4.5. Type ? for help, m for menu item.

```

Figure 10-1. A typical GNU info screen

In general, GNU Info uses the term *window* to describe the screen area that displays Info text. Each such window has a mode line at the bottom that describes the node being displayed. The information displayed in the mode line includes the following information, reading left to right:

- The name of the file
- The name of the node
- The number of screen lines necessary to display the node
- The amount of text above the current screen, expressed as a percentage

So, in Figure 10-1, the name of the file is `gcc.info.gz`, the name of the node is `Top`, it takes 40 lines to display the entire screen, and the screen currently is displaying the top of the file. The text “`zz`” at the beginning of the mode line indicates that the file was read from a compressed disk file. The text beginning with “`Subfile`” means that this node has been split into multiple files and that, in this case, you are viewing the subfile `gcc.info-1.gz`. This text does not appear if the Info node has been spread across several files.

The documentation for GNU Info (available, oddly enough, as an Info file—type `info info` to view it) refers to the screen real estate occupied by actual Info text as the *view area*. However, we prefer to use the term *window* because the info reader is capable of displaying multiple windows inside one screen and we do not want to have to refer to “the view area in the foo window” if “the foo window” is a clear enough reference. When the info reader displays multiple windows in the same screen, one window is separated from another by its mode line, and the mode line always marks the bottom of a given window.

An echo area appears on the last or bottom line of each Info screen. The *echo area*, also called an *echo line*, displays status information and error messages, and serves as a buffer for typing the text used for word searches or for other input you might have to provide. When you are at the top or the beginning of a node, a line of text across the top of the screen, which we call *node pointers*, identifies the current file and node and the next, previous, and parent nodes of the current node, when applicable. Node pointers and their usage are described in the next section, “Moving Around in GNU Info.”

The list of topics in the center of the screen in Figure 10-1 are known as *menu references*. If you move the cursor to any character between * and :: and press Enter, a process called *selecting a node*, you will jump to the Info file corresponding to that topic (commonly called a *node*). For example, Figure 10-2 shows the screen that appears if you select the topic “C Extensions” and then, on that screen, select the topic “Labels as Values.”



Figure 10-2. Following a GNU Info Menu reference

Notice that the screen shown in Figure 10-2 has two windows, each separated from the other by a mode line as described earlier in the chapter. The upper window contains Info text and the bottom window contains footnotes that appear in this node. If you press the spacebar, the text in the upper window scrolls forward one page or screen, but the footnote window remains static. You will learn later in this chapter how to navigate between windows. You might also notice that the mode line for the upper window indicates that you have jumped to the subfile `gcc.info-11.gz`.

NOTE *The astute reader will, of course, notice that the footnote that appears in the footnote window in Figure 10-2 also appears at the bottom of the text shown in the upper window. Feature? Mistake? Bug? You decide.*

Moving Around in GNU Info

You can get a lot done in GNU Info files just by pressing the spacebar (or Ctrl-v) to scroll through each screen in order. If you want to go backward, use Meta-v. If your keyboard has them, you can probably use the Next and Previous keys (also known as Page Down and Page Up on standard PC keyboards), instead of Ctrl-v and Meta-v. PC users will find Next and Previous more convenient and familiar. In all cases, though, the canonical sequences (Ctrl-v and Meta-v) should work. Similarly, the cursor keys allow you to scroll up and down through the screen you are currently looking at one line of text at a time (in this case, the canonical keys are Ctrl-n for up and Ctrl-p for down) or to move left and right on the current line of text using Ctrl-b and Ctrl-f (left and right). Most PC users might find the cursor movement keys the most familiar, but the Control key sequences should always work regardless of the keyboard.

The node movement keys b, e, l, n, p, and u are easiest to understand if you are looking at the top of an Info page. For example, if you type **info gcc invoking**, the first few lines should resemble Listing 10-3, the first page of the Invoking GCC node.

Listing 10-3. The First Page of the GCC Info File

```
File: gcc.info,  Node: Invoking GCC,  Next: Installation,  Prev: G++ and GCC,  \
Up: Top
```

```
GCC Command Options
*****
```

The line of text across the top shows the name of the file containing the node you are currently viewing (gcc.info), the name of the node itself (Node: Invoking GCC), the next node (Next: Installation), the previous node (Prev: G++ and GCC), and the top of the node tree (Up: Top). A node (more precisely, a node pointer) is an organizational unit in Info files that corresponds to chapters, sections, subsections, footnotes, or index entries in a printed book. It also makes it easier to navigate in an Info file. As Table 10-1 shows, n jumps to the next node, p jumps to the previous node, and u jumps to up a node or to the parent node. So, from the screen illustrated in Listing 10-3, typing **n** jumps to the Installation node, **p** jumps to the node title G++ and GCC, and **u** takes you to the Top node, which, in this case, is the main menu or table of contents for the GCC Info file. To put it another way, the Top node is the parent of the current node, which may or may not always be the main menu. However, it usually only takes a few presses of the **u** key to get to the top of the tree.

Within a node, typing **b** places the cursor at the top of the current node and typing **e** places you at the end of the current node. We think the **l** (*last node*) key performs one of the most useful jumps, because it takes you back to the most recently visited node (the last node) before the current one. The **l** key is comparable to the Back button in a Web browser. Figure 10-3 illustrates how a complete Info topic might be organized.

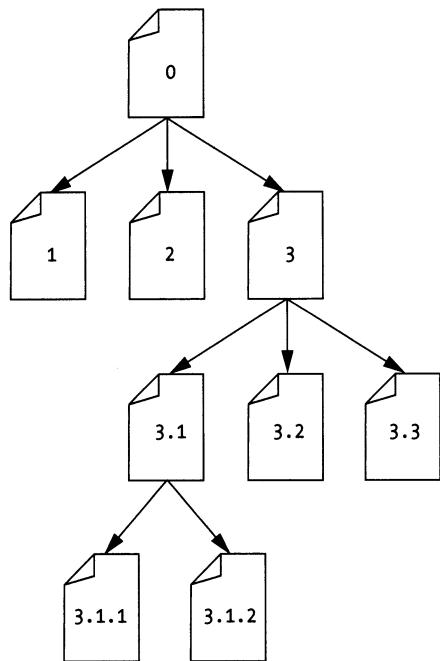


Figure 10-3. The node structure of a GNU Info file

There are three Top nodes in Figure 10-3. The Top node for 3.1.1 and 3.1.2 is 3.1. The Top node for 3.1, 3.2, and 3.3 is 3, and 3's Top node, like 1 and 2, is 0. So, if you were viewing 3.1.2, you would type **u** three times to return to the top or root node. If Figure 10-3 were arranged as a table of contents, it would resemble the following:

Chapter 1

Chapter 2

Chapter 3

Section 3.1

Subsection 3.1.1

Subsection 3.1.2

Section 3.2

Section 3.3

Finally, if you walked through the nodes using the n key or spacebar, the progression would be 0 -> 1 -> 2 -> 3 -> 3.1 -> 3.1.1 -> 3.1.2 -> 3.2 -> 3.3.

We have made such an effort to explain and illustrate the node structure of a GNU Info file and how to navigate through it because most users unfamiliar with Info complain about getting lost in it and of not knowing where they are, how to get back to where they were, and where they want to be. To boil it down to two simple rules:

1. Learn the mnemonics of the b, e, n, p, and u keys: (b)eginning of node, (e)nd of node, (n)ext node, (p)revious node, and (u)p to the Top node.
2. Keep your eye on the node pointers at the top of a node so you will know where the n, p, and u keys will take you.

Performing Searches in GNU Info

One of the most useful features of GNU Info is its ability to perform fast searches and a large variety of searches. The stand-alone info reader can search forward and backward searches and searches that are case sensitive or insensitive. You can search incrementally or by complete string, and you can also search an Info file's indices and then jump to the node containing the matching index entry. Although man page viewers have similar functionality, we have found GNU Info's search features to be faster and more capable. As always, your mileage may vary. Before jumping into the nuts and bolts of searches, Table 10-2 lists the search functions you need to know.

Table 10-2. GNU Info Search Commands

Key or Key Combination	Description
?	Searches backward for a string typed in the echo area, ignoring case
,	Jumps to the next node containing a match for a previously completed index search
i	Searches the file index for a string typed in the echo area and jumps to the node containing a match
s	Searches for a string typed in the echo area, ignoring case
S	Searches for a string typed in the echo area, paying attention to case
Ctrl-r	Searches backward for a string as you type it
Ctrl-s	Searches forward for a string as you type it
Ctrl-x n	Repeats the previous search, retaining the previous search's case sensitivity
Ctrl-x N	Repeats the previous search in the reverse direction, retaining the previous search's case sensitivity

While performing a forward search with Ctrl-s, you can press Ctrl-s to search forward for the next occurrence or Ctrl-r to search backward for the previous occurrence. Conversely, when running a backward search using Ctrl-r, pressing Ctrl-r repeats the backward search for the next (previous) occurrence, and Ctrl-s executes the same search in the forward direction.

Another way to repeat the previous search is to type **s** (or **S**) and press Enter to accept the default string, which is always the last search string embedded in square brackets (`[]`). Pressing Enter without typing a new search string uses the default value. When repeating previous searches using Ctrl-x n or Ctrl-x N, the previous search's case sensitivity option is applied. The search features Info provides do show one small bit of perversity: the **s** search is case insensitive unless the string entered contains an uppercase letter, in which case the search automatically becomes case sensitive.

TIP *In almost every mode, pressing Ctrl-g, sometimes several times, will restore Info's interface to something sane, canceling most operations.*

Following Cross-References

Another one of GNU Info's most useful features, and the source of much gnash-ing of teeth for the uninitiated, is its support for cross-references, referred to as *xrefs* in the GNU Info documentation. *xrefs* are pointers or hotlinks to other Info nodes that contain related information (and that may exist in a separate file). It is these *xrefs* that give Info its hypertext capability. It is also these *xrefs* that cause users to get lost and start hating Info. And, it is when using these *xrefs* that the `l` key becomes most useful—getting you back to where you started. A canonical *xref* has the following form:

`* label: target. comment`

An *xref* begins with `*`, followed by *label*, which is a name that you can use to refer to the *xref*, followed by a colon, which separates the label from the target, the target itself, and an optional comment. *target* is the full name of the node to which the *xref* refers. The optional *comment* is often used to describe the contents of the referenced node. Consider the following *xref*, taken from the GCC Info file's index:

`* allocate_stack instruction pattern: Standard Names.`

In this example, *label* is the text `allocate_stack instruction pattern` and *target* is the node named `Standard Names`. The period (`.`) is not part of the target, but merely indicates the end of the target name. To select an *xref*, position the cursor anywhere between the `*` and the end of the target name and then press Enter.

You will often see *xrefs* that use the following form:

`* G++ and GCC:: You can compile C or C++ programs.`

In this case, the target name has been omitted and a second colon used in its place. This means that the label and the target name have the same names. That is, the previous *xref* is equivalent to

`* G++ and GCC:G++ and GCC. You can compile C or C++ programs.`

This shorthand notation is especially common in node menus, such as the one shown in Figure 10-1. In fact, this shorthand notation is so commonly used in node menus that it is called a *menu reference* in order to distinguish from the other standard type of *xref*, a note reference. Unlike menu references, which appear in, you guessed it, node menus, *note references* appear in the body text of a node, inline with the rest of the text, much like the hyperlinks in the body text of a standard HTML Web page. Note references are very similar to menu references, except that they begin with `*Note`, rather than just a bare `*`. Listing 10-4, for example, extracted from the GCC Info file, contains a note reference.

Listing 10-4. A Typical Note Reference

element of the union should be used. For example,

```
union foo { int i; double d; };

union foo f = { d: 4 };
```

will convert 4 to a 'double' to store it in the union using the second element. By contrast, casting 4 to type 'union foo' would store it into the union as the integer 'i', since it is an integer. (*Note Cast to Union:::)

The note reference appears at the end of the listing and refers to the node named "Cast to Union," which explains the semantics of casting a type to a C union type. After selecting either type of xref, you can use the l key to return to the node you were reading before you jumped or you can follow another reference. No matter how far afield you stray, if you press l enough times, you will eventually traverse the history list the info reader maintains until you return to the node from which you started.

Printing GNU Info Nodes

GNU Info's documented method for printing an individual node is the command M-x print-node, which pipes the contents of the node to the lpr command. If the environment variable INFO_PRINT_COMMAND is set, the node's contents will be piped through the command defined in this variable. However, if you want to print a complete Info file rather than a single node, you might want to use the following command:

```
$ info --subnodes info_file -o - | lpr
```

This command prints the entire Info file named *info_file* using the default system printer. The --subnodes option tells the info reader to recurse through each node of *info_file*'s menus and display the contents of each node as it recurses through the node tree. The -o option tells the info reader to save the output in a file, and must be followed by a filename. A filename of - specifies standard output. In the command shown, output is piped into lpr, which, if your printer is properly configured, will print the entire manual. So, to print the entire GCC Info file, the command would be

```
$ info --subnodes gcc -o - | lpr
```

CAUTION *Do not try this command at home, kids! The entire GCC manual is over 500 pages long!*

Invoking GNU Info

The complete syntax for executing GNU Info from the command line is

```
info [options...] [info_file...]
```

info_file identifies the Info node(s) you want to view, separated by whitespace. For example, `info gcc` opens the GCC Info file. If you want to visit a particular menu item or subnode available from the menu (table of contents) in the GCC Info file, specify it after the top-level node. So, if you want to go straight to the Installation node of the GCC Info file, use the command `info gcc installation`. Additional arguments will take you deeper into the node tree. Thus, `info gcc installation cross-compiler` will take you to the Cross-Compiler subnode of the Installation subnode of the `gcc` node. The case of the specified nodes and subnodes is ignored. Table 10-3 lists the values for the options arguments that we find most useful—the complete list is available in the Invoking Info node of the `info` Info file (`info info invoking`).

Table 10-3. Common GNU Info Command-Line Options

Option	Description
<code>--apropos=word</code>	Starts info looking for <i>word</i>
<code>--dribble=file</code>	Logs keystrokes to <i>file</i> (see <code>--restore</code>)
<code>-f file</code>	Opens the Info file specified by <i>file</i> , bypassing the special directory file and the default search path
<code>--file=file</code>	Opens the Info file specified by <i>file</i> , bypassing the special directory file and the default search path
<code>-h</code>	Displays a short usage message
<code>--help</code>	Displays a short usage message
<code>--index-search string</code>	Searches the specified Info file's index node for <i>string</i> and jumps to the first matching node
<code>-o file</code>	Redirects Info output from the display to the file specified by <i>file</i>
<code>--output file</code>	Redirects Info output from the display to the file specified by <i>file</i>

Table 10-3. Common GNU Info Command-Line Options (continued)

Option	Description
-0	Jumps directly to an Info node that identifies how to invoke a program, if such a node exists
--raw-escapes	Leaves ANSI formatting on the page
--restore= <i>file</i>	Reads keystrokes from <i>file</i> (complements --dribble)
--show-options	Same as -0 and --usage
--subnodes	Recursively displays the child nodes of the top-level Info file (must be used with -o)
--usage	Jumps directly to an Info node that identifies how to invoke a program, if such a node exists
--vi-keys	Starts info using keybindings that work somewhat like vi

If `-f file` specifies an absolute pathname, such as `-f ./gcc.info`, Info will only look at the specified file or path, ignoring its built-in search path. If a search specified by `--index-search` fails, Info displays an error message. As explained earlier, a bare `-` specified with `-o` is interpreted as standard output, which allows you to use Info output in pipelines and with the `--subnodes` option.

Stupid Info Tricks

Despite the title “Stupid Info Tricks,” this section offers a grab bag of tips and hints for using GNU Info. Learning to use the information in this section might turn you into a Certified GNU Info Power User, allowing you to dazzle your friends and colleagues with your mastery of GNU Info.

One of our favorite tricks uses the `--subnodes` option. As you can probably imagine, we spent a lot of time studying GCC’s Info pages. When we got tired of looking at onscreen text, we used the following shell pipeline, or a variation of it, to print out all or part of the GCC manual.

```
$ info --subnodes gcc -o - | enscript -G -H -o - | psnup -2 | lpr
```

The first part of the command you have already seen. Instead of piping the output straight to the printer, we ran it through `enscript` to convert the text to PostScript, piped `enscript`’s output to `psnup` to create 2-up pages (to reduce paper consumption), and then printed the result. `enscript`’s `-G` option adds a flashy header to the top of each (virtual) page; `-H` adds highlight bars to the printed output, emulating what old computer hacks know as greenbar (although, in this case, it is probably more accurate to call it *greybar*).

Using Command Multipliers

GNU Info, like GNU Emacs, enables you to prefix most commands with a numeric multiplier, causing the commands to execute multiple times. The command that invokes the multiplier is `Ctrl-u [n]`, where n is the number of times you want the following command to execute. If you omit n , the default value of the multiplier is 4. So, for example, to scroll down four screens, you can press `Ctrl-u 4 Ctrl-v`, or `Ctrl-u Ctrl-v`. To scroll backward two screens, you can try `Ctrl-u 2 Meta-v`, or `Ctrl-u -2 Ctrl-v`. Yes, that is -2 in the second command—when you use a negative multiplier value, most operations, such as moving or scrolling, work in the opposite or reverse direction. Thus, for example, using a negative multiplier with `Ctrl-b` causes the cursor to move forward rather than backward.

Working with Multiple Windows

We mentioned earlier in the chapter that GNU Info can display multiple windows in a single screen, with mode lines serving as the visual separators between windows. Info also provides methods for moving from one window to another, changing the size of windows, and for creating and deleting windows. The unspoken question, of course, is, “Why would one want to use multiple windows?” The most common reason to have multiple windows is to keep from having to jump back and forth between nodes. Instead of jumping back and forth between two Info nodes, you can open an additional window (strictly speaking, you just split the existing window into two separate windows), display a different node in the new window, and then switch between them. This approach allows you to look at the information in two nodes more or less simultaneously. Obviously, it should be clear that you can open as many windows as you want, although too many windows open in the same screen will quickly become unwieldy and difficult to read.

To split the current window, press `Ctrl-x 2`. This command splits the *current window*, defined as the window containing the cursor, into two equally sized windows, leaving the cursor in the original window. Info commands, such as cursor movement, scrolling, and following xrefs, only apply to the current window. To move the cursor to the newly created window, press `Ctrl-x o` (the mnemonic to keep in mind is [o]ther window). Repeatedly pressing `Ctrl-x o` cycles through all open windows. Conversely, to move to the previous window, you can use `Meta-x o`. To close a window, you have two options. To delete the current window, that is, the window in which the cursor is currently located, press `Ctrl-x 0`. To close all windows except the current one, press `Ctrl-x 1`. If you want to scroll the text another window forward, press `Meta-Ctrl-v`. Finally, if you want to make the current window larger and other windows smaller, use the command `Ctrl-x ^`. The default growth step is one line, so this command is a good one to use with the command multiplier (`Ctrl-u [n]`). So, for example, to grow the current window by three lines, use the command `Ctrl-u 3 Ctrl-x ^`.

To reverse this effect, you can either switch to the other window and grow it by three lines (Ctrl-x o Ctrl-u 3 Ctrl-x ^) or you can shrink the current window by using a multiplier of -3, that is, press Ctrl-u -3 Ctrl-x ^.

Getting Help

Predictably, GNU info has a help screen, accessible by pressing Ctrl-h. Use the Info commands discussed in this chapter to navigate the help screen. When you want to close the help window, press the l key—the help window is just a specially handled node pointer, so the l key returns you to the node you were originally viewing. If you are so inclined, you can go through the Info tutorial by pressing Ctrl-h h. Remember the magic rescue sequence: if all else fails and you find yourself totally discombobulated, press the q key, possibly several times, to exit Info.

GCC Command-Line Options

As you would expect from the world's most popular (and freely available) compiler, the compilers in the GNU Compiler Collection have a tremendous number of potential command-line options. This is partially due to the flexibility that arises from having thousands of users and contributors to GCC, each of whom wants the compiler to behave in a specific way, but even more so due to the tremendous number of platforms and architectures on which GCC is used.

GCC's online help in `info` format is a great source of reference and usage information for GCC's command-line options. However, option information in GCC `info` is organized into logical groups of options, rather than providing a simple, alphabetical list that you can quickly scan to obtain information about using a specific option. It is also sometimes somewhat out of date—we all know that the documentation is often the last thing to be updated.

This chapter provides an easier reference for GCC command-line options than the documentation provided with GCC. The table that appears at the end of this chapter lists system/hardware-independent command-line options as they are organized in the GCC documentation—by conceptually related groups of options. This can be useful if you are trying to perform a specific task and wonder what options might be relevant to that task or type of task.

The bulk of this chapter provides a single, monolithic list of all of the machine-independent GCC command-line options, organized alphabetically for easy reference. This section makes it easy to look up information for any specific GCC option. In combination with the table at the end, this should make it easier for you to work with GCC and take advantage of the tremendous power that its zillions of command-line options provide.

NOTE *GCC also provides hundreds of machine-specific options that you will rarely need to use unless you are using a specific platform and need to take advantage of some of its unique characteristics. Because these are both the largest set of GCC options and the set that you will be using least frequently if you tend to work on a more standard hardware platform or operating system, we have moved the discussion of those options to Appendix B, which provides a summary and list of all the machine-specific options for GCC.*

Alphabetical List of GCC Options

Only those well-versed in character and string comparisons know offhand whether “A” is less than “a,” and so on, and it’s a pain to type “man ascii” each time you need to remember how to sort a specific letter. Since not everyone may be one with the ASCII chart (and may not even be using a Linux, *BSD, or Un*x machine, for that matter), the options described in this section are listed more-or-less alphabetically, with uppercase options preceding lowercase options involving the same letter of the alphabet. Options whose names begin with symbols are listed before the alphabetic options, followed by options beginning with numerals, and concluding with alphabetic options. The number of dashes preceding any given argument is shown, but is ignored for sorting purposes.

NOTE *Though this book focuses on using GCC for C compilation, this section includes options for other languages, identifying the appropriate language whenever possible. This is intended as much for completeness’ sake as to help you identify options that are irrelevant for the language that you are compiling.*

-###: Specifying this output option causes the GCC commands relevant to your command line to be displayed in quoted form, but not executed. This option is typically used to identify mode-specific commands that you can subsequently incorporate into shell scripts, or to verify exactly what GCC is attempting to execute in response to specific command-line options.

-ansi: For C and C++ programs, this option enforces compliance with ANSI C (ISO C89) or standard C++, disabling features such as the `asm` and `typeof` keywords; predefined, platform-specific macros such as `unix` or `vax`; the use of C++ // comments in C programs; and so on. When using this option, non-ISO programs can still be successfully compiled—to actively reject programs that attempt to use non-ANSI features, you must also specify the `-pedantic` option.

-aux-info *filename*: For C programs, specifying this option causes prototyped declarations for all referenced functions to be dumped to the specified output file *filename*, including those defined in header files. This option is silently ignored in any language other than C. The output file contains comments that identify the source file and line number for each declared function, and letters indicating whether the declaration was implicit (I), prototyped (N), or unprototyped (O), and whether the function was declared (C) or defined (F) there. Function definitions are followed by a list of arguments and their declarations.

-b*machine*: When using GCC as a cross-compiler, this option enables you to identify the target machine, and therefore the associated compiler.

machine should be the same value specified when you executed the configure script to build the cross-compiler.

-B*prefix*: When using GCC as a cross-compiler, this option enables you to specify a *prefix* that should be used to try to find the executables, libraries, and include and data files for the compiler. For each of the subprograms (cpp, cc1, as, and ld) run by the compiler, using this option causes the compiler driver to try to use *prefix* to locate each subprogram, both with and without any values specified with the -b (*machine*) and -V (*version*) options. If binaries with the specified *prefix* are not found in the directories listed in your PATH environment variable, the GCC driver also looks in the directories /usr/lib/gcc and /usr/local/lib/gcc-lib for both binaries and subdirectories with relevant names. Using this option also causes the GCC driver to attempt to locate and use include files and libraries with the specified *prefix*. This command-line option is equivalent to setting the `GCC_EXEC_PREFIX` environment variable before compilation.

-c: An output option, specifying this option causes the GCC driver to compile or assemble the source files without linking them, producing separate object files. This option is typically used to minimize recompilation when compiling and debugging multimodule applications.

-d*letters*: An internal compiler debugging option (rather than an application debugging option) that causes the GCC driver to generate debugging output files during compilation at times specified by *letters*. The names of the debugging files are created by appending a pass number (*pass*) to a word identifying the phase of compilation to the name of the source file (*file*), separated by a period, and then adding an extension that reflects the phase of compilation at which the file was generated. Values for *letters* and the names of the output files are as follows:

A: Annotates the assembler output with miscellaneous debugging information

a: Produces the rtl, flow2, addressof, stack, postreload, greg, lreg, life, cfg and jump debugging output files

B: Dumps after block reordering (*file.pass.bbro*)

b: Dumps after computing branch probabilities (*file.pass.bp*)

C: Dumps after the first if-conversion (*file.pass.ce*)

- c: Dumps after instruction combination (*file.pass.combine*)
- D: Dumps all macro definitions at the end of preprocessing
- d: Dumps after delayed branch scheduling (*file.pass.dbr*)
- E: Dumps after the second if-conversion (*file.pass.ce2*)
- e: Dumps after static single assignment (SSA) optimizations
(*file.pass.ssa* and *file.pass.ussa*)
- F: Dumps after purging ADDRESSOF codes (*file.pass.addressof*)
- f: Dumps after life analysis (*file.pass.life*)
- G: Dumps after global common subexpression elimination (GCSE)
(*file.pass.gcse*)
- g: Dumps after global register allocation (*file.pass.greg*)
- h: Dumps after finalization of exception handling (EH) code
(*file.pass.eh*)
- i: Dumps after sibling call optimizations (*file.pass.sibling*)
- j: Dumps after the first jump optimization (*file.pass.jump*)
- k: Dumps after conversion from registers to stack (*file.pass.stack*)
- l: Dumps after loop optimization (*file.pass.loop*)
- l: Dumps after local register allocation (*file.pass.lreg*)
- M: Dumps after performing the machine-dependent reorganization
(*file.pass.mach*)
- m: Prints statistics on memory usage at the end of the run
- N: Dumps after the register move pass (*file.pass.regmove*)
- n: Dumps after register renumbering (*file.pass.rnreg*)
- o: Dumps after post-reload optimizations (*file.pass.postreload*)
- P: Dumps the register transfer language (RTL) in the assembler output
as a comment before each instruction
- p: Annotates assembler output with a comment identifying the pattern,
alternative, and length of each instruction
- R: Dumps after the second scheduling pass (*file.pass.sched2*)
- r: Dumps after RTL generation (*file.pass.rtl*)

- S: Dumps after the first scheduling pass (*file.pass.sched*)
 - s: Dumps after first common subexpression elimination (CSE) and associated jump optimization pass (*file.pass.cse*)
 - t: Dumps after the second CSE and associated jump optimization pass (*file.pass.cse2*)
 - v: For each dump file, dumps a representation of the control flow graph suitable for viewing with VCG (*file.pass.vcg*)
 - w: Dumps after the second flow pass (*file.23.flow2*)
 - X: Dumps after SSA dead-code elimination pass (*file.pass.ssadce*)
 - x: Only generates RTL for each function instead of compiling it
 - y: Dumps debugging information during parsing
 - z: Dumps after the peephole pass (*file.pass.peephole2*)
- dumpmachine: Using this debugging option displays the compiler's target machine and then exits.
- dumpspecs: Using this debugging option displays the compiler's built-in specs and then exits. These are specifications stored in a file named `specs`, located in the `lib/gcc-lib` directory of the directory hierarchy in which you installed GCC. GCC's specifications tell GCC where to find various mandatory files and libraries, which tools to use at each phase of compilation, and how to invoke them.
- dumpversion: Using this debugging option displays the compiler's version and then exits.
- E: Specifying this output option causes the GCC framework to define the macros `_GNUC_`, `_GNUC_MINOR_`, and `_GNUC_PATCHLEVEL_`, and to stop after the preprocessing stage without running the compiler itself. Files that do not require preprocessing are ignored.
- falign-functions | -falign-functions=n: Specifying this optimization option causes GCC to align the start of functions to a machine-specific value (when *n* is not specified) or the next power-of-two boundary greater than *n*, skipping up to *n* bytes. Specifying `-falign-functions=1` is equivalent to specifying `-fno-align-functions`, meaning that functions will not be aligned.

-falign-jumps | -falign-jumps=n: Specifying this optimization option causes GCC to align branch targets to a machine-specific value (when *n* is not specified) or to the next power-of-two boundary, skipping up to *n* bytes. Specifying **-falign-jump=1** is equivalent to specifying **-fno-align-jumps**, meaning that jumps will not be aligned.

-falign-labels | -falign-labels=n: Specifying this optimization option causes GCC to align all branch targets to a machine-specific value (when *n* is not specified) or to the next power-of-two boundary, skipping up to *n* bytes. Specifying this option causes GCC to insert dummy options in output code to cause the requested alignment. Specifying **-falign-labels=1** is equivalent to specifying **-fno-align-labels**, meaning that labels will not be aligned. If **-falign-loops** or **-falign-jumps** are given and their machine-specific or specified values are greater than the value requested by this option, their values are used instead.

-falign-loops | -falign-loops=n: Specifying this optimization option causes GCC to align loop targets to a machine-specific value (when *n* is not specified) or to the next power-of-two boundary, skipping up to *n* bytes. Specifying **-falign-loops=1** is equivalent to specifying **-fno-align-loops**, meaning that loops will not be aligned.

-fallow-single-precision: When compiling C applications, this option specifies not to promote single precision math operations to double precision (the K&R C default), even if **-traditional** is specified. Using this option may provide performance optimizations on certain architectures. This option has no effect when compiling with ISO or GNU C conventions.

-falt-external-templates: A deprecated option for C++ compilation, this option generates template instances based on where they are first instantiated. This option is similar to **-fexternal-templates**.

-fargument-alias | -fargument-noalias | -fargument-noalias-global: These code generation options specify the possible relationships among parameters and between parameters and global data. These options rarely need to be used—in most cases, the GCC framework uses the options appropriate to the language that is being compiled. The **-fargument-alias** option specifies that arguments (parameters) can alias each other and global storage. The **-fargument-noalias** option specifies that arguments do not alias each other, but can alias global storage. The **-fargument-noalias-global** option specifies that arguments do not alias each other or global storage.

-fasynchronous-unwind-tables: This code generation option causes GCC to generate a loop unwind table in DWARF2 format (if DWARF2 is supported by the target machine) that can be used in stack unwinding by asynchronous external events such as a debugger or garbage collector.

-fbounds-check: In GCC's Java and Fortran-77 front ends, this optimization option automatically generates additional code that checks whether all array indices are within the declared size of the appropriate array. This option is on by default in Java, and false by default for FORTRAN-77. This option is ignored when compiling code in languages other than Java and FORTRAN-77.

-fbranch-probabilities: This optimization option uses the *file.da* files produced by a run of GCC with the `-fprofile-arcs` option to further optimize code based on the number of times each branch was taken. The information in the .da files is closely linked to the GCC options used during a specific run of GCC, so you must use the same source code and the same optimization options for both compilations.

-fcall-saved-reg: This code generation option tells GCC to treat the register named *reg* as an allocable register whose contents are saved by functions, and will therefore persist across function calls. Functions compiled with this option save and restore the register *reg* if they use it. This option should not be used with registers such as the frame pointer or stack pointer that are used internally by the compiler, or in which function values are returned. Registers are machine specific—those valid for each specific GCC output target are defined in the `REGISTER_NAMES` macro in the machine description macro file.

-fcall-used-reg: This code generation option tells GCC to treat the register named *reg* as an allocable register whose contents may be overwritten by function calls. Functions compiled with this option will not save and restore the register *reg*. This option should not be used with registers such as the frame pointer or stack pointer that are used internally by the compiler. Registers are machine specific—those valid for each specific GCC output target are defined in the `REGISTER_NAMES` macro in the machine description macro files.

-fcaller-saves: This optimization option tells GCC to automatically add extra instructions that save and restore the contents of each register across function calls. This enables compiled code to make global use of registers that may also be used as scratch pads within various functions. This option is active by default for systems that have no call-preserved registers. This option is automatically enabled when using optimization level 2 and higher.

-fcheck-new: When compiling C++ programs, specifying this option causes GCC to check that the pointer returned by the operator new is nonnull before attempting to use the storage that the pointer refers to. This should be unnecessary, since new should never return a null pointer. If you declare your operator new as throw(), G++ will automatically check the return value.

-fcond-mismatch: When compiling C programs, specifying this option allows the successful compilation of conditional expressions with mismatched types in the second and third arguments. The value of such expressions is void. This option is not supported for C++.

-fconserve-space: When compiling C++ programs, specifying this option causes GCC to put uninitialized or runtime-initialized global variables into the common segment, as is done when compiling C programs. This saves space in the executable but obscures duplicate definitions, and may cause problems if these variables are accidentally destroyed multiple times. This option is no longer useful on most targets because newer releases of GCC typically put variables into the BSS without making them common.

-fconstant-string-class=*classname*: When compiling Objective C programs, using this option tells GCC to use *classname* as the name of the class to instantiate for each literal string specified with the syntax @"..." . The default class name is NXConstantString.

-fcse-follow-jumps: Used during common subexpression elimination (CSE), this optimization option tells GCC to scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an if statement with an else clause, CSE will follow the jump when the condition tested is false.

-fcse-skip-blocks: Used during common subexpression elimination, this optimization option is similar to -fcse-follow-jumps, but causes CSE to follow jumps that conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, this option causes CSE to follow the jump around the body of the if statement.

-fdata-sections: For output targets that support arbitrary code sections, this optimization option causes GCC to place each data item into its own section in the output file. The name of the data item determines the name of the section in the output file. This option is typically used on systems with HPPA or Sparc processors (under HP-UX or Solaris 2, respectively), whose linkers can perform optimizations that improve the locality of reference in the instruction space. These optimizations may also be available on AIX and systems using the ELF object format. Using this option causes the assembler and linker to create larger object and executable files that may therefore be slower on some systems. Using this option also prevents the use of gprof on some systems and may cause problems if you are also compiling with the -g option.

-fdelayed-branch: If supported by the target machine, this optimization option causes GCC to attempt to reorder instructions in order to exploit instruction slots that are available after delayed branch instructions.

-fdelete-null-pointer-checks: This optimization option tells GCC to use global dataflow analysis to identify and eliminate useless checks for null pointers. The compiler assumes that dereferencing a null pointer would have halted the program, so that pointers that are checked after being dereferenced cannot be null. On systems that can safely dereference null pointers, you can use the -fno-delete-null-pointer-checks to disable this optimization.

-fdiagnostics-show-location=[once|every-line]: This option tells GCC's diagnostic message formatter how frequently diagnostic messages should display source information when diagnostic messages are wrapped across multiple lines. The default value is once; specifying every-line causes the diagnostic formatter to specify the source location on every output line, even if it is just a continuation of a previous message.

-fdollars-in-identifiers: This C++ compilation option tells GCC to accept the dollar sign symbol (\$) in identifiers. You can also explicitly prohibit use of dollar signs in identifiers by using the -fno-dollars-in-identifiers option. This is a backward-compatibility option: K&R C allows the character \$ in identifiers, but ISO C and C++ forbid the use of this character.

-fdump-class-hierarchy | -fdump-class-hierarchy-*options*: When compiling C++ programs, specifying this debugging option causes GCC to dump a representation of each class's hierarchy and virtual function table layout to a file. The filename is constructed by appending 'class' to the name of each source file. If the *options* form is used, the values specified in *options* control the details of the dump as described in the definition for the -fdump-tree option.

-fdump-translation-unit | -fdump-translation-unit-*options*: When compiling C and C++ programs, specifying this debugging option causes GCC to dump a representation of the tree structure for the entire translation unit to a file. The filename is constructed by appending .tu to the name of each source file. If the *options* form is used, the values specified in *options* control the details of the dump as described in the description of the **-fdump-tree** option.

-fdump-tree-switch | -fdump-tree-switch-*options*: When compiling C++ programs, specifying this debugging option causes GCC to dump the intermediate language tree to a file at various stages of processing. The filename is constructed by appending a *switch*-specific suffix to the source filename. The following tree dumps are possible:

inlined: Dumps to the file *file.inlined* after function inlining

optimized: Dumps to the file *file.optimized* after all tree-based optimization

original: Dumps to the file *file.original* before performing any tree-based optimization

If the *options* form is used, *options* is a list of hyphen-separated options that control the details of the dump. *options* that are irrelevant to a specific dump are ignored. The following options are available:

address: Displays the address of each node. This address changes according to the environment and each source file, and therefore is primarily used to associate a dump file with a specific debug environment.

all: Turns on all options.

slim: Inhibits dumping members of a scope or function body simply because that scope has been reached. Only items that can be directly reached by some other path are dumped.

-fdump-unnumbered: When doing debugging dumps due to the use of the **-d** option, specifying this option causes GCC to suppress instruction and line numbers. This makes it easier to use **diff** to compare debugging dumps from multiple runs of GCC with different compiler options, most specifically with and without the **-g** option.

-fexceptions: This code generation option tells GCC to enable exception handling and generates any extra code needed to propagate exceptions. This option is on by default when compiling applications written in languages such as C++ that require exception handling. It is primarily provided for use when compiling code written in languages that do not natively use exception handling, but that must interoperate with C++ exception handlers. For some targets, using this option also causes GCC to generate frame unwind information for all functions, which can substantially increase application size although it does not affect execution.

NOTE *As an optimization, you can use the -fno-exceptions option to disable exception handling in older C++ applications that do not use exception handling.*

-fexpensive-optimizations: This optimization option tells GCC to perform a number of minor optimizations that may require a substantial amount of processing time.

-fexternal-templates: Deprecated. When compiling C++ applications, using this option causes GCC to apply #pragma interface and #pragma implementation to template instantiation. Template instances are emitted or suppressed according to the location of the template definition.

-ffast-math: This optimization option causes GCC to define the preprocessor macro __FAST_MATH__ and perform internal math optimizations. This option implies the use of the -fno-math-errno, -funsafe-math-optimizations, and -fno-trapping-math options and activates them if they are not specified. This option should never be used in conjunction with GCC's standard -O optimization options, because this can result in incorrect output for programs that depend on the exact implementation of IEEE or ISO rules and specifications for math functions.

-ffixed-reg: This code generation option tells GCC to treat the register identified by *reg* as a fixed register that is never referred to by generated code other than compiler internals. Registers are machine specific—those valid for each specific GCC output target are defined in the REGISTER_NAMES macro in the machine description macro file.

-ffloat-store: This optimization option tells GCC not to store floating-point variables in registers and to inhibit other options that might change whether a floating-point value is taken from a register or memory. Using this option prevents excess precision on machines where floating registers keep more precision than a double requires, such as the 68000 (with 68881) and x86 architectures. Additional precision may be undesirable in programs that rely on the precise definition of IEEE floating point. You can compile such programs with this option only if you modify them to store relevant intermediate computations in variables rather than in registers.

-ffor-scope: When compiling C++ programs, specifying this option tells GCC to limit the scope of variables declared in a `for-init` statement to the `for` loop, as specified by the C++ standard. If the opposite `-fno-for-scope` option is used, the scope of variables declared in a `for-init` statement extends to the end of the enclosing scope. This was the default behavior of older versions of GNU C++ and many traditional implementations of C++. If neither option is specified, GCC follows the C++ standard.

-fforce-addr: This optimization option causes GCC to force memory address constants to be copied into registers before doing arithmetic on them.

-fforce-mem: This optimization option causes GCC to force memory operands to be copied into registers before doing arithmetic on them. This may produce better code because it makes all memory references potential common subexpressions that can be reduced. When they are not common subexpressions, instruction combination should eliminate the additional overhead of separate register load. This option is automatically turned on when using the `-O2` optimization option.

-ffreestanding: When compiling C applications, this option tells GCC that compilation takes place in a freestanding environment where the standard library may not be available or exist, such as when compiling an operating system kernel. Using this option implies the use of the `-fno-builtin` option, and is equivalent to using the `-fno-hosted` option.

-ffunction-sections: For output targets that support arbitrary code sections, this optimization option causes GCC to place each function into its own section in the output file. The name of the function determines the name of the section in the output file. This option is typically used on systems with HPPA or Sparc processors (under HP-UX or Solaris 2, respectively), whose linkers can perform optimizations that improve the locality of reference in the instruction space. These optimizations may also be available on AIX and systems using the ELF object format. Using this option causes the assembler and linker to create larger object and executable files that may therefore be slower on some systems. Using this option also prevents the use of gprof on some systems and may cause problems if you are also compiling with the -g option.

-fgcse: This optimization option tells GCC to perform a global common subexpression elimination pass, also performing global constant and copy propagation.

NOTE *When compiling a program using GCC's computed gotos extension, you may get better runtime performance if you disable the global common subexpression elimination pass by specifying the -fno-gcse option.*

-fgcse-lm: This optimization option causes GCC to attempt to optimize load operations during global command subexpression elimination. If a load within a loop is subsequently overwritten by a store operation, GCC will attempt to move the load outside the loop and to only use faster copy/store operations within the loop.

-fgcse-sm: This optimization causes GCC to attempt to optimize store operations after global common subexpression elimination. When used in conjunction with the **-fgcse-lm** option, loops containing a load/store sequence will be changed to a load before the loop and a store after the loop whenever possible.

-fgnu-runtime: When compiling Objective C programs, this option causes GCC to generate object code that is compatible with the standard GNU Objective C runtime. This is the default on most systems.

-fhosted: When compiling C programs, this option tells GCC that the entire standard C library is available during compilation, which is known as a *hosted environment*. Specifying this option implies the **-fbuiltin** option. This option is usually appropriate when compiling any application other than a kernel, unless you want to compile statically linked applications. Using this option is equivalent to using the **-fno-freestanding** option.

-finhibit-size-directive: This code generation option tells GCC not to output a `.size` assembler directive or any other internal information that could cause trouble if the function is split and the two portions subsequently relocated to different locations in memory. This option is typically used only when compiling the `crtstuff.c` character handling routine and should not be necessary in any other case.

-finline-functions: This optimization option tells GCC to integrate simple functions into the routines that call them if they are simple enough to do so, based on heuristics. If all calls to a given function are integrated and the function is declared static, no independent assembler code for the function is generated independent of that in the routines that call it.

-finline-limit=n: This optimization option tells GCC to modify its internal limit on the size of functions that are explicitly marked as inline with the `inline` keyword or that are defined within the class definition in C++. *n* represents the number of pseudo instructions (an internal GCC calculation) in such functions, excluding instructions related to parameter handling. The default value of *n* is 600. Increasing this value can result in more inlined code, which may cause increased compilation time and memory consumption. Decreasing this value can improve compilation faster, but may result in slower programs because less code will be inlined. This option can be quite useful for programs such as C++ programs that use recursive templates and therefore can substantially benefit from inlining.

-finstrument-functions: This code generation option tells GCC to insert instrumentation calls for function entry and exit. The following profiling functions will be called with the address of the current function and the address from which it was called immediately after entering and immediately before exiting from each function:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit (void *this_fn, void *call_site);
```

The first argument is the address of the start of the current function, which can be looked up exactly in the symbol table. You can specify the `no_instrument_function` attribute for specific functions in order to avoid making profiling calls from them. For more information about profiling and for examples of using profiling functions and attributes, see Chapter 6.

-fkeep-inline-functions: This optimization option tells GCC to generate a separate, callable version of each function marked as `inline`, even if all calls to a given function have resulted in inline code. This does not apply to functions marked as `extern inline`.

-fkeep-static-consts: This optimization option causes GCC to create and allocate all variables declared `static const`, even if the variables are not referenced. This option is active by default. To force GCC to check whether variables are referenced and to only create them if this is the case, use the `-fno-keep-static-consts` option.

-fleading-underscore: This code generation option is provided for compatibility with legacy assembly code, and forces C symbols to be created with leading underscores in object files. This is not completely supported on all GCC output targets.

-fmath-errno: This optimization option tells GCC to set the value of `ERRNO` after calling math functions that are executed with a single instruction, such as the `sqrt` function. Programs that rely on IEEE exceptions for math error handling may want to use the `-fno-math-errno` option to provide performance improvements while still maintaining IEEE arithmetic compatibility.

-fmem-report: This debugging option causes GCC to display statistics about permanent memory allocation at the end of compilation.

-fmerge-all-constants: This optimization option causes GCC to attempt to merge identical constants and variables. This increases the scope of the `-fmerge-constants` option, and also tries to merge arrays and variables that are initialized with string and floating-point constants. This may generate nonconformant C and C++ code because these languages require all nonautomatic variables to have distinct locations.

-fmerge-constants: This optimization option causes GCC to attempt to merge identical string and floating-point constants across compilation units. This option is on by default during optimization on output targets where the assembler and linker support it.

-fmessage-length=n: This diagnostic option causes GCC to format diagnostic messages so that they fit on lines of *n* characters or less, inducing line wrapping in messages that are greater than *n* characters. The default value for *n* is 72 characters for C++ messages and 0 for all other front ends supported by GCC. If *n* is zero, no line wrapping will be done.

-fms-extensions: When compiling C++ programs, this option causes GCC not to display warnings about nonconformant constructs used in the Microsoft Foundation Classes, such as messages about implicit integer return values and nonstandard syntax for getting pointers to member functions.

-fmove-all-movables: This optimization option causes GCC to move all invariant computations in loops outside the loop.

-fnext-runtime: When compiling Objective C programs, this option causes GCC to generate output that is compatible with the former NeXT computer system runtime that is used on Darwin and Mac OS X systems.

-fno-access-control: When compiling C++ programs, this option disables access checking. This option is rarely used and is primarily provided to work around problems in the access control code.

-fno-asm: This option affects the keywords that are recognized when compiling C and C++ programs. When compiling ISO C99 programs, this option disables the `asm` and `typeof` keywords. When compiling other C programs, this option also disables the `inline` keyword. When compiling C++ programs, this option only disables the `typeof` keyword. You can still use the keywords `_asm_`, `_inline_`, and `_typeof_` in any C or C++ application. This option is automatically enabled when using the `-ansi` option.

-fno-branch-count-reg: Specifying this optimization option causes GCC not to use decrement and branch instructions on a count register. Instead, GCC generates a sequence of instructions that decrement a register, compare it against zero, and then branch based upon the result. This option is only meaningful on architectures such as x86, PowerPC, IS-64, and S/390 that provide decrement and branch instructions.

-fno-builtin | -fno-builtin-function: Specifying this C and Objective C option causes GCC not to recognize built-in functions that do not begin with `_builtin_` as a prefix. This rule is always the case in C++ applications compiled with GCC; to specifically invoke GCC's internal functions, you must call them with the `_builtin_` prefix.

GCC normally generates special code to handle its built-in functions more efficiently. In some cases, built-in functions may be replaced with inline code that makes it difficult to set breakpoints during debugging or to link with external libraries that provide equivalent functions.

In C and Objective C applications, you can use the less restrictive `-fno-builtin-function` option to selectively disable specific built-in functions. This option is ignored if no such built-in function is present. Similarly, you can use the `-fno-builtin` option to disable all built-in functions and then modify your applications to selectively map function calls to the appropriate built-ins, as in the following example:

```
#define strcpy(d, s)      __builtin_strcpy ((d), (s))
```

`-fno-common`: Specifying this code generation option when compiling a C language application causes GCC to allocate all global variables in the data section of the object file, even if uninitialized, rather than generating them as common blocks. This option is provided for compatibility with existing systems, but has the side effect that if the same variable is declared in multiple, separately compiled source modules, and is not explicitly declared as `extern`, GCC will display an error message during linking.

`-fno-const-strings`: Specifying the C++ option causes GCC to assign the type `char *` to string constants rather than `const char *`, as specified in the C++ standard. Specifying this option does not allow you to write to string constants unless you also use the `-f writable-strings` option. Using this option is deprecated—writing to constants really is not good form.

`-fno-cprop-registers`: Specifying this optimization option causes GCC to perform an additional copy-propagation pass to try to improve scheduling and, where possible, eliminate the register copy entirely. This is done after register allocation and post-register allocation instruction splitting.

`-fno-default-inline`: Specifying this C++ optimization option causes GCC not to assume that functions declared within a class scope should be inlined. If you do not specify this option, GCC will automatically inline the functions if you use any optimization level (-O, -O2, or -O3) when compiling C++ applications.

`-fno-defer-pop`: Specifying this optimization option causes GCC to pop the arguments to each function call as soon as that function returns. GCC normally lets arguments accumulate on the stack for several function calls and pops them all at once on systems where a pop is necessary after a function call return.

-fno-function-cse: Specifying this optimization option causes GCC not to put function addresses in registers but to instead make each instruction that calls a constant function contain the explicit address of that function. You may need to use this option if you receive assembly errors when using other optimization options.

-fno-elide-constructors: Specifying this option when compiling C++ applications causes GCC not to omit creating temporary objects when initializing objects of the same type, as permitted by the C++ standard. Specifying this option causes GCC to explicitly call the copy constructor in all cases.

-fno-enforce-eh-specs: Specifying this option when compiling C++ applications causes GCC to skip checking for violations of exception specifications at runtime. This option violates the C++ standard, but may be useful for reducing code size. The compiler will still optimize based on the exception specifications.

-fno-for-scope: Specifying this option when compiling C++ applications causes GCC to extend the scope of variables declared inside a `for` loop to the end of the enclosing scope. This is contrary to the C++ standard but was the default in older versions of GCC and most other traditional C++ compilers. If neither this option or the opposite `-ffor-scope` option is specified, GCC adheres to the standard, but displays a warning message for code that might be invalid or assumes the older behavior.

-fno-gnu-keywords: This option affects the keywords that are recognized when compiling C and C++ programs. When compiling ISO C99 programs, this option disables the `asm` and `typeof` keywords. When compiling other C programs, this option also disables the `inline` keyword. When compiling C++ programs, this option only disables the `typeof` keyword. You can still use the keywords `_asm_`, `_inline_`, and `_typeof_` keywords in any C or C++ application. This option is automatically enabled when using the `-ansi` option.

-fno-gnu-linker: This code generation option is used when you are compiling code for linking with non-GNU linkers, and suppresses generating global initializations (such as C++ constructors and destructors) in the form used by the GNU linker. Using a non-GNU linker also requires that you use the `collect2` program during compilation to make sure that the system linker includes constructors and destructors, which should have been done automatically when configuring the GCC distribution.

-fno-guess-branch-probability: This optimization option tells GCC to use an empirical model to predict branch probabilities, which may prevent some optimizations. Normally, GCC may use a randomized model to guess branch probabilities when none are available from either profiling feedback (`-fprofile-arcs`) or `_builtin_expect`. This may cause different runs of the compiler on the same program to produce different object code, which may not be desirable for applications such as real-time systems.

-fno-ident: This code generation option causes GCC to ignore the `#ident` directive.

-fno-implicit-templates: When compiling C++ programs, this option tells GCC to only emit code for explicit instantiations, and not to generate code for noninline templates that are instantiated by use.

-fno-implicit-inline-templates: When compiling C++ programs, this option tells GCC not to generate code for implicit instantiations of inline templates. This option is typically used in combination with optimization options to minimize duplicated code.

-fno-implement-inlines: When compiling C++ programs, this option tells GCC not to generate out-of-line copies of inline functions based on the `#pragma implementation`. This saves space, but will cause linker errors if the code for these functions is not generated inline everywhere these functions are called.

-fno-inline: This optimization option prevents GCC from expanding any function inline, effectively ignoring the `inline` keyword. During normal optimization, the code for functions identified as `inline` is automatically inserted at each function call.

-fno-math-errno: This optimization option prevents GCC from setting `ERRNO` after calling math functions that can be executed as a single instruction, such as `sqrt`. Not setting `ERRNO` reduces the overhead of calling such functions, but `ERRNO` should be set in programs that depend on exact IEEE or ISO compliance for math functions. The `-fno-math-errno` option is therefore not turned on by any of the generic GCC optimization options, and must be specified manually.

-fno-nonansi-builtins: When compiling C++ programs, this option disables the use of built-in GCC functions that are not specifically part of the ANSI/ISO C specifications. Such functions include `_exit()`, `alloca()`, `bzero()`, `conjf()`, `ffs()`, `index()`, and so on.

-fno-operator-names: When compiling C++ programs, this option prevents GCC from recognizing keywords such as `and`, `bitand`, `bitor`, `compl`, `not`, `or`, and `xor` as synonyms for the C++ operators that represent those operations.

-fno-optional-diags: When compiling C++ programs, this option disables optional diagnostics that are not mandated by the C++ standard. In version 3.2.2 of GCC, the only such diagnostic is one generated when a name has multiple meanings within a class. Other such diagnostics may be added in future releases of GCC.

-fno-peephole | -fno-peephole2: These optimization options disable different types of machine-specific optimizations. Whether one, the other, or both of these options are useful on your system depends on how or if they were implemented in the GCC implementation for your system.

-fno-rtti: When compiling C++ programs, this option tells GCC not to generate information about every class with virtual functions. This information is typically used by C++ runtime type identification features such as `dynamic_cast` and `typeid`. Using this option can save space if you do not explicitly use those aspects of C++—they will still be used by internals such as exception handling, but the appropriate information will only be generated when needed.

-fno-sched-interblock: This optimization option tells GCC not to schedule instructions across basic blocks, which is normally done by default when using the `-fschedule-insns` option or optimization options `-O2` or higher.

-fno-sched-spec: This optimization option tells GCC not to move non-load instructions, which is normally done by default when using the `-fschedule-insns` option or optimization options `-O2` or higher.

-fno-signed-bitfields: When compiling C programs, this option tells GCC that bit fields are unsigned by default (in other words, when neither `signed` or `unsigned` is present in their declaration). Without specifying this option, bit fields are assumed to be signed in order to be consistent with other basic data types. This option is redundant when used in conjunction with the `-traditional` option, which causes all bit fields to be unsigned by default.

-fno-stack-limit: This code generation option causes GCC to generate code without an explicit limit on stack size.

-fno-trapping-math: This optimization option causes GCC to generate code that assumes that floating-point operations cannot generate user-visible traps, which may result in faster operation due to reduced overhead when returning from such functions. If you want to experiment with or perform this type of optimization, this option must be explicitly specified because its use may result in code that is not completely IEEE or ISO compliant. This option is never automatically turned on by any standard GCC optimization option.

-fno-unsigned-bitfields: When compiling C programs, this option tells GCC that bit fields are signed by default (in other words, when neither the signed or unsigned keyword is present in their declaration). This option should not be used in conjunction with the `-traditional` option, which causes all bit fields to be unsigned by default.

-fno-verbose-asm: This code generation option minimizes the number of comments inserted into generated assembly code, and is the GCC default.

-fno-weak: When compiling C++ programs, this option tells GCC not to use weak symbol support, even if it is provided by the linker—GCC's default action is to use weak symbols when they are available. This option is primarily used for testing at this point and generally should not be used.

-fnon-call-exceptions: When compiling languages such as C++ that support exceptions and when runtime support for exception handling is present, specifying this option causes GCC to generate code that allows trap instructions to throw exceptions. This option does not enable exceptions to be thrown from arbitrary signal handlers.

-fomit-frame-pointer: This optimization option tells GCC not to keep the frame pointer in a register for functions that do not require a frame pointer. Besides making an additional register available for other code, this option reduces both code size and the execution path by eliminating the instructions required to save, set up, and restore frame pointers. This option will have no effect on systems where the standard function calling sequence always includes frame pointer allocation and setup. When building GCC for a specific platform, the `FRAME_POINTER_REQUIRED` macro determines whether this option is meaningful on a specific target system.

-foptimize-register-move: This optimization option tells GCC to attempt to reassign register numbers in simple operations in an attempt to maximize register tying. This option is synonymous with the `-fregmove` option, and is automatically enabled when using GCC optimization levels 2 and higher.

-foptimize-sibling-calls: This optimization option attempts to optimize sibling and tail recursive calls.

-fpack-struct: This code generation option tells GCC to attempt to pack all structure members together without holes, reducing memory use in applications that allocate significant numbers of in-memory data structures. This option is rarely used in applications that make extensive use of system functions that may employ offsets based on the default offsets of fields in data structures.

-fpcc-struct-return: This code generation option causes GCC to return short (integer-sized) struct and union values in memory rather than in registers. This option is typically used when linking object code compiled with GCC with object code produced by other compilers that use this convention.

-fpermissive: When compiling C++ code, specifying this option causes GCC to downgrade the severity of messages about nonconformant code to warnings, rather than treating them as actual errors. This option has no effect if used with the `-pedantic` option.

-fPIC: This code generation option tells GCC to emit position-independent code that is suitable for dynamic linking, but eliminates limitations on the size of the global offset table. Position-independent code (PIC) uses a global offset table to hold constant addresses that are resolved when an application is executed. This option is therefore only meaningful on platforms that support dynamic linking. If you are interested in potentially reducing the size of the global offset table, you should use the `-fPIC` option instead of this one.

-fpic: This code generation option tells GCC to generate position-independent code that is suitable for use in a shared library. This option is only meaningful on target platforms that support dynamic linking and use a machine-specific value (typically 16K or 32K) for the maximum size of the global allocation table for an application. If you see an error message indicating that this option does not work, you should use the `-fPIC` option instead of this one.

NOTE *Code generated for the IBM RS/6000 is always position independent.*

-fprefetch-loop-arrays: This optimization option tells GCC to generate instructions to prefetch memory on platforms that support this. Prefetching memory improves the performance of loops that access large arrays.

-freg-struct-return: When compiling C or C++ applications, this code generation causes GCC to generate code that returns struct and union values in registers whenever possible. By default, GCC uses whichever of the `-fpcc-struct-return` or `-freg-struct-return` options is appropriate for the target system.

-freorder-functions: This optimization option causes GCC to optimize function placement using profile feedback.

-frepo: When compiling C++ applications, this option enables automatic template instantiation at link time. Using this option also implies the `-fno-implicit-templates` option.

-fpretend-float: This debugging option is often used when cross-compiling applications and tells GCC to compile code assuming that both the host and target systems use the same floating-point format. Though this option can cause actual floating constants to be displayed correctly, the instruction sequence will probably still be the same as the one GCC would make when actually running on the target machine.

-fprofile-arcs: This debugging and optimization option tells GCC to instrument arcs (potential code paths from one function or procedure to another) during compilation to generate coverage data. This coverage information can subsequently be used by `gcov` or to enable GCC to perform profile-directed block ordering. Arc coverage information is saved in files with the `.da` (directed arc) extension after each run of an instrumented application. To enable profile-directed block-ordering optimizations, you must compile your application with this option, execute it with a representative data set, and then compile the program again with the same command-line options, adding the `-fbbranch-probabilities` option. To use this option during code coverage analysis, you must also use the `-ftest-coverage` option.

-freduce-all-givs: This optimization option tells GCC to strength reduce all general-induction variables used in loops. Strength reduction is an optimization that uses previous calculations or values to eliminate more expensive calls or calculations. This option is activated by default when any GCC optimization level is used.

-fregmove: This optimization option tells GCC to reassign register numbers in order to maximize the amount of register tying, and is synonymous with the -foptimize-register-move option. This option is active by default when using GCC optimization level 2 or higher.

-frename-registers: This optimization option tells GCC to make use of any unallocated registers in order to attempt to avoid false dependencies in scheduled code. This option is therefore most frequently used on systems with large numbers of registers.

-frerun-cse-after-loop: This optimization option tells GCC to rerun common subexpression elimination after loop optimization has been performed.

-frerun-loop-opt: This optimization option tells GCC to run the loop optimizer twice, attempting to immediately capitalize on the results of the first pass.

-fsched-spec-load: This optimization option tells GCC to move some load instructions where this is predicted to improve performance, reduce the execution path, or enable subsequent optimizations. This option is typically used only when you are also using the -O2, -O3, or -fschedule-insns options to schedule before register allocation.

-fsched-spec-load-dangerous: This optimization option is slightly more aggressive than the -fsched-spec-load option, and tells GCC to be even more aggressive in moving load instructions in order to improve performance, reduce the execution path, or enable subsequent optimizations. This option is typically only used when you are also using the -O2, -O3, or -fschedule-insns options to schedule before register allocation.

-fschedule-insns: If supported on the target machine, this optimization option tells GCC to attempt to reorder instructions to eliminate execution stalls that occur when the data required for an operation is unavailable. This option can be quite useful on systems with slow floating-point or memory load instructions by enabling other instructions to execute until the result of the other instructions are available.

-fschedule-insns2: This optimization option is similar to -fschedule-ins, but tells GCC to perform an additional pass to further optimize instruction scheduling after register allocation has been performed. This option can further improve performance on systems with a relatively small number of registers or where memory load instructions take more than one cycle.

-fshared-data: This code generation option causes GCC to locate data and nonconstant variables in the code that is currently being compiled in shared, rather than private, data. This may be useful on operating systems where shared data is literally sharable between processes running the same program.

-fshort-double: This code generation option tells GCC to use the same size when storing double and float data.

-fshort-enums: This code generation option tells GCC to minimize the amount of storage allocated to enumerated data types, only allocating as many bytes as necessary for the complete range of possible values. In other words, the amount of storage associated to enum data types will be the smallest integer data type that provides sufficient space.

-fshort-wchar: This C programming language option causes GCC to override the underlying data type used for wchar_t so that it is hardwired to be a short unsigned int instead of whatever the default data type is for the target hardware.

-fsigned-bitfields: This C programming language option controls whether a bit field is signed or unsigned when this is unspecified. By default, bit fields are typically signed because this is consistent with basic integer types such as int, which are also signed unless the -traditional option is also specified on the command line.

-fsigned-char: This C programming language option causes GCC to define the char data type as signed, requiring the same amount of storage as signed char. This option is equivalent to specifying the -fno-unsigned-char command-line option.

-fsingle-precision-constant: This optimization option tells GCC to handle floating-point constants as single precision constants instead of implicitly converting them to double precision constants.

-fssa: This optimization option tells GCC to perform its optimizations using static single assignment (SSA) form. The flow graph for each function is first translated into SSA form, optimizations are done while in that form, and the SSA form is then translated back into a flow graph. This option is still under active development and should be used with caution.

-fssa ccp: This optimization option tells GCC to do Sparse Conditional Constant Propagation in SSA form. The -fssa option must also be specified in order to use this option and, like that option, this option is still actively under development and should be used with caution.

-fssa-dce: This optimization option causes GCC to perform aggressive dead-code elimination in SSA form. The `-fssa` option must also be specified in order to use this option and, like that option, this option is still actively under development and should be used with caution.

-fstack-check: This code generation option tells GCC to add extra code to force the operating system to notice whenever the stack is extended, helping ensure that applications do not accidentally exceed the stack size. (The operating system must still monitor the stack size.) This option is primarily useful in multithreaded environments, where more than one stack is in use. The automatic stack overflow detection provided by most systems in single-stack (single-process) environments is usually sufficient without using this option.

-fstack-limit-register=reg | -fstack-limit-symbol=SYM: These code generation options tell GCC to add extra code that ensures the stack does not grow beyond a certain value, where `reg` is the name of a register containing the limit or `SYM` is the address of a symbol containing the limit. A signal is raised if the stack grows beyond that limit. For most targets, the signal is raised before the stack crosses the boundary, so the signal can be caught and handled without taking special precautions. If you are using this option, you should also use the GNU linker to ensure that register names and symbol addresses are calculated and applied correctly.

-fstats: This C++ option tells GCC to display front-end processing statistics once compilation has completed. The GNU C++ (G++) development team uses this information.

-fstrength-reduce: This optimization option tells GCC to do loop strength reduction and iteration variable elimination.

-fstrict-aliasing: This optimization option tells GCC to use the strictest aliasing rules applicable to the language being compiled. For C (and C++), this performs optimizations based on expression type. Objects of two different types are assumed to be located at different addresses unless the types are structurally similar. For example, an `unsigned int` can be an alias for an `int`, but not for a `void *` or `double`. A character type can be an alias for any other type. This option can help detect aliasing errors in potentially complex data types such as unions.

-fsyntax-only: This diagnostic option tells GCC to check the code for syntax errors, without actually compiling any part of it.

-ftemplate-depth-*n*: This C++ option sets the maximum instantiation depth for template classes to *n*. Limits on template instantiation depth are used to detect infinite recursion when instantiating template classes. ANSI/ISO C++ standards limit instantiation depth to 17.

-ftest-coverage: This debugging option causes GCC to create two data files for use by the gcov code-coverage utility. The first of these files is *source.bb*, which provides a mapping of basic blocks to line numbers in the source code. The second of these files is *source.bbg*, which contains a list of all of the arcs in the program's flow graph. If this option is used with the **-fprofile-arcs** option, executing the compiled program will also create the data file *source.da*, which provides runtime execution counts used in conjunction with the information in the *source.bbg* file. Coverage data generally maps better to source files if no optimization options are used when generating code coverage information.

-fthread-jumps: This optimization option optimizes jumps that subsequently perform redundant comparisons, skipping those comparisons and redirecting the code to the appropriate point later in the code execution flow.

-ftime-report: This debugging option causes GCC to display statistics about the time spent in each compilation pass.

-ftls-model=*model*: This code generation option tells GCC to use a specific thread-local storage model. *model* should be either `global-dynamic`, `local-dynamic`, `initial-exec`, or `local-exec`. The default is `global-dynamic`, unless the **-fpic** option is used, in which case the default is `initial-exec`.

-ftracer: This code generation pass simplifies the control flow of functions, allowing other optimizations to do a better job.

-ftrapv: This optimization option causes GCC to generate traps for signed overflow on addition, subtraction, and multiplication operations.

-funroll-all-loops: This optimization causes GCC to unroll all loops, even if the number of times they are executed cannot be guaranteed when the loop is entered. Though this usually makes programs run more slowly, it provides opportunities for subsequent optimization through code elimination.

-funroll-loops: This optimization causes GCC to unroll loops, but limits the loops that will be unrolled to those where the number of times they are executed can be determined at compile time or when entering the loop. Using this option implies both the -fstrength-reduce and -frerun-cse-after-loop options. Using this option makes code larger but does not guarantee improved performance or execution speed. It does provide opportunities for subsequent optimization through other GCC optimization options.

-funsafe-math-optimizations: This optimization option enables GCC to perform optimizations for floating-point arithmetic that assume that arguments and results are valid and may violate IEEE or ANSI standards. This option should never be used in conjunction with any standard optimization (-O) option because it can result in incorrect output in programs that depend on an exact implementation of the IEEE or ISO specifications for math functions.

-funsigned-bitfields: This C language option controls whether a bit field is signed or unsigned, when neither keyword is specified. Bit fields are ordinarily signed by default because this is consistent with basic integer types such as int, which are signed types. Using the -traditional option forces all bit fields to be unsigned regardless of whether this option is specified.

-funsigned-char: This C language option forces the char data type to be unsigned. This overrides any default character data type defaults for a given system. This option is valuable when porting code between system types that have different defaults for the char data type. Note that the char type is always distinct from signed char and unsigned char even though its behavior is always the same as either of those two.

-funwind-tables: This code generation option tells GCC to enable exception handling and generates any static data used when propagating exceptions. This option is similar to the -fexceptions option, but generates static data rather than code. This option is rarely used from the command line, and is usually incorporated into language processors that require this behavior (such as C++).

-fuse-cxa-atexit: This C++ option causes GCC to register destructors for objects with static storage duration using the `_cxa_atexit()` function rather than the `atexit` function. This option is required for fully standards-compliant handling of static destructors, but only works on systems where the C library supports the `_cxa_atexit()` function.

-fverbose-asm: This code generation option inserts extra comments into generated assembly code to make it more readable. This option is generally only used during manual optimization or by people who are verifying the generated assembly code (such as the GCC maintenance and development teams).

-fvolatile: This code generation option tells GCC to consider all memory references through pointers to be volatile.

-fvolatile-global: This code generation option tells GCC to consider all memory references to extern and global data items to be volatile. This option does not cause GCC to consider static data items to be volatile.

-fvolatile-static: This code generation option tells GCC to consider all memory references to static data to be volatile.

-fvtable-gc: This C++ option tells GCC to generate special relocations for vtables and virtual function references. This enables the linker to identify unused virtual functions and zero out vtable slots that refer to them. This option is commonly used with the **-ffunction-sections** option and the linker's **-Wl,--gc-sections** option, in order to also discard the functions themselves. This optimization requires that you are also using GNU as and GNU ld, and is not supported on all system types. Note that the **-Wl,--gc-sections** option is ignored unless the **-static** option is also specified.

-fwritable-strings: This C language option tells GCC to store string constants in the writable data segment without unquifying them. This option is provided for compatibility with older programs that assume they can write into string constants (even though this is poor form). Specifying the **-traditional** option also causes this behavior.

-g: This debugging option causes GCC to include debugging and symbol table information in object files, which can subsequently be used by GDB. The format of these object files and the debugging information that they contain depends on the native binary format associated with the platform, and can be one of the following: COFF (SVR3 systems), DWARF (SVR4), STABS (Linux), or XCOFF (AIX).

On most systems that use STABS format, including the **-g** option enables the use of extra debugging information that only GDB can employ; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to ensure generation of the extra information, use **-gstabs+**, **-gstabs**, **-gcoff+**, **-gcoff**, **-gdwarf-1+**, **-gdwarf-1**, or **-gvms**. Users of the **-gvms** option have our inherent sympathy.

Unlike most other C compilers, GCC allows you to use `-g` with optimization options such as `-O`, but this is not recommended. As you would expect, optimized code may occasionally produce surprising results due to optimization in loop structure, loop control, variable elimination, statement movement, result compression, and so on.

Other than the format-related options listed previously, related debugging options include `-ggdb`, `-glevel` (and similar level options), profiling options such as `-p`, `-pg`, `-Q`, `-ftime-report`, `-fmem-report`, `-fprofile-arcs`, `-ftest-coverage`, `-dletters`, `-fdump-unnumbered`, `-fdump-translation-unit`, `-fdump-class-hierarchy`, `-fdump-tree`, `-fpretend-float`, `-print-multi-lib`, `-print-prog-name=program`, `-print-libgcc-file-name`, `-print-file-name=libgcc.a`, `-print-search-dirs`, `-dumpmachine`, `-dumpversion`, and `-dumpspecs`.

`-glevel | -ggdblevel | -gstabslevel | -gcofflevel | -gxoofflevel | -gvmslevel`: These debugging options cause GCC to produce debugging information in various formats, using the *level* value to specify the amount of information displayed. The default level is 2. Level 1 produces minimal information, sufficient for displaying backtraces in parts of the program that the user does not expect to debug. Level 3 includes information such as the macro definitions present in the program.

`-gcoff`: This debugging information causes GCC to produce debugging information in the COFF format used by SDB on most System V systems prior to System V Release 4.

`-gdwarf`: This debugging option causes GCC to produce debugging information in DWARF (version 1) format, used by the SDB debugger on most System V Release 4 systems.

`-gdwarf+`: This debugging option causes GCC to produce debugging information in DWARF version 1 format using GNU extensions understood only by the GNU debugger (GDB). Debugging applications compiled with this option using other debuggers or compiling them with toolchains that are not strictly composed on GNU tools may cause unexpected behavior.

`-gdwarf-2`: This debugging option causes GCC to produce debugging information in DWARF version 2 format, which is used by the DBX debugger on IRIX 6.

`-gen-decls`: This Objective C option causes GCC to dump interface declarations for all classes seen in each source file (*sourcefile*) to a file named *sourcename.decl*.

-ggdb: This debugging option causes GCC to produce debugging information that is targeted for use by GDB. Using this option tells GCC to employ the debugging options that provide the greatest amount of detail for the target platform.

-gstabs: This debugging option forces GCC to produce debugging information in the STABS format without using the GDB extensions. The STABS format is used by most debuggers on BSD and Linux systems. GDB can still be used to debug these applications, but will not be able to debug them as elegantly as if the **-gstabs+** option had been specified.

-gstabs+: This debugging option causes GCC to produce debugging information in STABS format, using the GDB extensions that are only meaningful to the GNU debugger, GDB. Debugging applications compiled with this option using other debuggers or compiling them with toolchains that are not strictly composed on GNU tools may cause unexpected behavior.

-gvms: This sad, lonely debugging option is only relevant to users who are writing applications intended to be executed and debugged by the DEBUG application on VMS systems. SYS\$SYSOUT, anyone?

-gxcoff: This debugging option causes GCC to produce debugging information in XCOFF format, if supported on the execution platform. XCOFF is the format used by the classic Unix DBX debugger on IBM RS/6000 systems.

-gxcoff+: This debugging option causes GCC to produce debugging information in XCOFF format, using debugging extensions that are only understood by the GNU debugger (GDB). Debugging applications compiled with this option using other debuggers or compiling them with toolchains that are not strictly composed on GNU tools may cause unexpected behavior.

--help: This output option causes GCC to display a summary list of the command-line options that can be used with GCC. Using this option in conjunction with the **-v** option causes GCC to also pass the **--help** option to all subsequent applications invoked by GCC, such as the assembler, linker, and loader, which will also cause them to display a list of many of the standard command-line options that they accept. Adding the **-W** command-line option to the **--help** and **-v** options will also cause GCC and subsequent applications in the toolchain to display command-line options that can be used, but which have no documentation except for this book.

-I-: This directory search option is used during preprocessing to identify additional directories that should be searched for the #include definition files used in C and C++ applications. Any directories specified using the -I option before the -I- option are only searched for files referenced as #include "file", not for files referenced as #include <file>. If you use the -I option to specify additional directories to search after using the -I- option, those additional directories will be searched both for files referenced as #include "file" and #include <file>.

Using the -I- option also keeps the preprocessor from examining the working directory for #include files referenced in #include statements. You can subsequently use the -I option to explicitly search the working directory if you specify it by name on the command line.

NOTE *Using the -I- option does not cause GCC to ignore the standard system directories in which #include files are typically located. To do this, specify the -nostdinc option.*

-Idir: This directory search option is used during preprocessing to identify additional directories that should be added to the beginning of the list of directories that are searched for #include files. This can be used to override #include files in an operating system's include directories that have the same name as #include files local to your application's source code. If you use more than one -I option on the GCC command line, the specified directories are scanned in the order that they were specified, from left to right, followed by the standard system directories.

NOTE *You should not use this option to add #include directories that contain vendor-supplied system header files. Such directories should be specified using the -isystem option, which puts them in the search order after directories specified using the -I option, but before the system header files.*

CAUTION *Using the -I option to specify system include directories (such as /usr/include and /usr/include/sys) is not a good idea. GCC's installation procedure often corrects bugs in system header files by copying them to GCC's include directory and then modifying its copies. Because GCC is your friend, it will display a warning whenever you specify a system include directory using this option.*

-Ldir: This directory search option causes the linker to add the specified directory *dir* to the list of directories to be searched for libraries specified using the **-l** command-line option.

-no-integrated-cpp: For C and C++ applications, this debugging option causes GCC to invoke the external C preprocessor (*cpp*) rather than the internal C preprocessor that is included with GCC. The default is to use the internal *cpp* that is provided with GCC. Specifying this option in conjunction with the **-B** option enables you to integrate a custom C preprocessor into your GNU toolchain and allows you to integrate a user-supplied *cpp* that you then specify via the **-B** option. For example, you could name your preprocessor *mycpp* and then cause GCC to use it by specifying the **-no-integrated-cpp -Bmy** option sequence.

-nostdinc: Specifying this C language directory search option prevents the preprocessor from searching the standard system directories for #include files specified using #include <file> statements.

-nostdinc++: Specifying this C++ language directory search option prevents the preprocessor from searching the C++-specific system directories for #include files specified using #include <file> statements. Standard system include directories such as /usr/include and /usr/include/sys are still searched.

-O | -O1: Specifying this optimization option causes GCC to attempt to reduce the size and improve the performance of the target application. On most systems, the **-O** option turns on the **-fthread-jumps** and **-fdelayed-branch** options.

Without optimization, GCC's primary goal is to compile applications as quickly as possible. A secondary goal is to make it easy to subsequently debug those applications, if necessary. Compiling with optimization will almost certainly take more time and will also require more memory when compiling any sizable function or module. Optimization may also combine variables or modify the execution sequence of an application, which can make it difficult to debug an optimized application. You rarely want to specify an optimization option when compiling an application for debugging unless you are debugging the optimization process itself.

-O0: Specifying this optimization option explicitly disables optimization. This option is the equivalent of not specifying any **-O** option. While seemingly meaningless, this option is often used in complex Makefiles where the optimization level is specified in an environment variable or command-line Makefile option.

-O2: Specifying this optimization option causes GCC to attempt additional optimizations beyond those performed for optimization level 1. In this optimization level, GCC attempts all supported optimizations that do not trade off between size and performance. This includes all optimization options with the exception of loop unrolling (-funroll-loops), function inlining (-finline-functions), and register renaming (-frename-registers). As you would expect, using the -O2 option increases both compilation time and the performance of compiled applications.

-O3: Specifying this optimization option causes GCC to attempt all performance optimizations, even if they may result in a larger compiled application. This includes all optimization options performed at optimization levels 1 and 2, plus loop unrolling (-funroll-loops), function inlining (-finline-functions), and register renaming (-frename-registers).

-o *file*: Specifying this output option tells GCC to write its output binary to the file *file*. This is independent of the type of output that is being produced: preprocessed C code, assembler output, and object module, or a final executable.

If the -o option is not specified, executable output will be written to the following files:

Executables: Written to a file named a.out (regardless of whether a.out is the execution format)

Object files: Written to files with the input suffix replaced with ".o" (*file.c* output is written to *file.o*.)

Assembler output: Written to files with the input suffix replaced with ".s" (for example, assembler output for the file *file.c* output is written to *file.s*.)

Preprocessed C source code: Written to standard output

-p: Specifying this debugging option causes GCC to generate extra code that will produce profiling information that is suitable for the analysis program prof. This option must be used both when compiling and linking the source file(s) that you want to obtain profiling information about.

-pass-exit-codes: Specifying this output option causes GCC to return the numerically highest error code produced during any phase of the compilation process. GCC typically exists with a standard Unix error code of 1 if an error is encountered in any phase of the compilation.

`--param name=value`: Specifying this optimization option provides control over the parameters used to control various optimization options. For example, GCC will not inline functions that contain more than a certain number of instructions. The `--param` command-line option gives you fine-grained control over limits such as this. All of these parameters are integer values.

Possible parameters that you can specify are the following:

`max-delay-slot-insn-search`: The maximum number of instructions to consider when looking for an instruction to fill a delay slot. Increasing values mean more aggressive optimization, resulting in increased compilation time with a potentially small improvement in performance.

`max-delay-slot-live-search`: The maximum number of instructions to consider while searching for a block with valid live register information when trying to fill delay slots. Increasing this value means more aggressive optimization, resulting in increased compilation time.

`max-gcse-memory`: The approximate maximum amount of memory that will be allocated in order to perform global common subexpression elimination optimization. If more memory than the specified amount is required, global common subexpression optimization will not be done.

`max-gcse-passes`: The maximum number of passes of global common subexpression elimination to run.

`max-pending-list-length`: The maximum number of pending dependencies scheduling will allow before flushing the current state and starting over. Large functions with few branches or calls can create excessively large lists that needlessly consume memory and resources.

`max-inline-insns`: Functions containing more than this number of instructions will not be inlined. This option is functionally equivalent to using the `-finline-limit` option with the same value.

`-pedantic`: Specifying this diagnostic/warning option causes GCC to display all warnings demanded for strict ISO C and ISO C++ compliance. Using this option does not verify ISO compliance, because it only issues warnings for constructs for which ISO C and ISO C++ require such a message (plus some that have been added in GCC but are not strictly mandatory for ISO C/C++). This option also causes GCC to refuse to compile any program that uses extensions and C/C++ syntax that are not ISO compliant. Valid ISO C and ISO C++ programs should compile properly with or without this option (though some may require additional restrictive options such as `-ansi` or a `-std` option specifying a specific version of ISO C).

Using the `-pedantic` option does not generate warning messages for alternate keywords whose names begin and end with `_`, and are also disabled for expressions that follow these keywords. These extensions are typically only used in system software, rather than application software.

`-pedantic-errors`: Specifying this diagnostic/warning message causes GCC to display error messages rather than warnings for non-ISO C/C++ constructs.

`-pg`: Specifying this debugging option causes GCC to generate extra code that produces profile information suitable for use by the analysis program `gprof`. This option must be used both when compiling and linking the source file(s) that you want to obtain profiling information about.

`-pipe`: Specifying this output option causes GCC to use pipes rather than temporary files when exchanging data between various stages of compilation. This can cause problems on systems where non-GNU tools (such as a custom preprocessor, assembler, and so on) are used as part of the compilation toolchain.

`-print-file-name=library`: Specifying this debugging option causes GCC to display the full pathname of the specified *library*. This option is often used when you are not linking against the standard or default system libraries, but you do want to link with a specific library, such as `libgcc.a`, as in the following example:

```
gcc -nodefaultlibs foo.c bar.c... 'gcc -print-file-name=libgcc.a'
```

When used in this form, surrounding the command `gcc -print-file-name=libgcc.a` with backquotes causes the command to be executed and its output displayed, which is then incorporated on the compilation command line as an explicit reference to the target library.

NOTE If the specified library is not found, GCC simply echoes the library name.

`-print-libgcc-file-name`: Specifying this debugging option is a shortcut for using the option `-print-file-name=libgcc.a`, and is used in the same circumstances.

-print-multi-directory: Specifying this debugging option causes GCC to print the directory name corresponding to the multilib selected by any other switches that are given on the command line. This directory is supposed to exist in the directory defined by the `GCC_EXEC_PREFIX` environment variable.

-print-multi-lib: Specifying this debugging option causes GCC to display the mapping from multilib directory names to compiler switches that enable them. This information is extracted from the specification files used by the compiler, in which the directory name is separated from the switches by a semicolon, and each switch starts with an @ symbol instead of the traditional dash/minus symbol, with no spaces between multiple switches.

What Are Multilibs?

Multilibs are libraries that are built multiple times, each with a different permutation of available machine-specific compiler flags. This makes it easy for GCC to produce output targeted for multiple platforms and to take advantage of different types of optimizations for similar but different platforms by having precompiled versions of libraries targeted for each.

Multilibs are typically built when you are using GCC with multiple targets for a given architecture, where you need to support different, machine-specific flags for various combinations of targets, architectures, subtargets, subarchitectures, CPU variants, special instructions, and so on.

You can display any multilibs available on your system by executing the `gcc -print-multi-lib` command. Multilibs are specified in entries in the compiler specification files that are stored in the `install-dir/lib/architecture/version/specs` file associated with each GCC installation.

-print-prog-name=program: Specifying this debugging option causes GCC to display the full pathname of the specified *program*, which is usually a part of the GNU toolchain.

NOTE If the specified *program* is not found, GCC simply echoes the name of the specified *program*.

-print-search-dirs: Specifying this debugging option causes GCC to print the name of its installation directory and the program and library directories that it will search for mandatory files, and then exit. This option is useful when debugging installation problems reported by GCC. To resolve installation problems, you can either rebuild GCC correctly or symlink or move any problematic components into one of the directories specified in the output of this command. You can often temporarily hack around installation problems by setting the environment variable `GCC_EXEC_PREFIX` to the full pathname (with a trailing `/`) of the directory where missing components are actually installed.

-Q: Specifying this debugging option causes GCC to print the name of each function as it is compiled and print some general statistics about each pass of the compiler.

-S: Specifying this output option causes GCC to stop after generating the assembler code for any specified input files. The assembler file for a given source file has the same name as the source file, but has an `.s` extension instead of the original extension of the input source file.

-save-temp: Specifying this debugging option causes GCC to preserve all temporary files produced during the compilation process, storing them in the working directory of the compilation process. This produces `.i` (preprocessed C input) and `.s` (assembler) files for each file specified for compilation. These files have the same basename as the original input files, but a different extension.

-std=std: This C language option tells GCC which C standard the input file is expected to conform to. You can use the features of a newer C standard even without specifying this option as long as they do not conflict with the default ISO C89 standard. Specifying a newer version of ISO C using `-std` essentially enables GCC's support for features in the specified standard to the default features found in ISO C89. Specifying a newer C standard changes the warnings that will be produced by the `-pedantic` option, which will display the warnings associated with the new base standard, even when you specify a GNU extended standard.

Possible values for `std` are the following:

`c89 | iso9899:1990`: ISO C89 (the same as using the `-ansi` switch).

`iso9899:199409`: ISO C89 as modified in amendment 1.

`c99 | iso9899:1999`: ISO C99. This standard is not yet completely supported. For additional information, see <http://gcc.gnu.org/version/c99status.html>, where *version* is a major GCC version such as `gcc-3.1`, `gcc-3.2`, `gcc-3.3`, and so on. At the time this book was written, the standard names `c9x` and `iso9899:199x` can also be specified, but are deprecated.

`gnu89`: The default value (i.e., the value used when the `-std` option is not given), this refers to ISO C89 with some GNU extensions and ISO C99 features.

`gnu99`: ISO C99 with some GNU extensions. This will become the default once ISO C99 is fully supported in GCC. The name `gnu9x` can also be specified, but is deprecated.

`--target-help`: Specifying this C language output option causes GCC to print a list of options that are specific to the compilation target. Using this option is the easiest way to get an up-to-date list of all platform-specific GCC options for the target platform.

`-time`: Specifying this debugging option causes GCC to display summary information that lists user and system CPU time (in seconds) consumed by each step in the compilation process (`cc1`, `as`, `ld`, and so on). User time is the time actually spent executing the specified phase of the compilation process. System time is the time spent executing operating system routines on behalf of that phase of the compilation process.

`-traditional`: This C language option tells GCC to attempt to support some aspects of traditional C compilers and non-ANSI C code, and also automatically invokes the parallel `-traditional-cpp` option for GCC's internal C preprocessor. This option can only be used if the application being compiled does not reference header (`#include`) files that do not contain ISO C constructs. Though much beloved by K&R C fans, this option is deprecated and may disappear in a future release of GCC.

TIP When using this option, you may also want to specify the `-fno-builtin` option if your application implements functions with the same names as built-in GCC functions.

Some of the backwards-compatibility features activated by this option are the following:

- Older function call sequence is acceptable; parameter types can be defined outside the parentheses that delimit the parameters but before the initial bracket for the function body.
- All automatic variables not declared using the `register` keyword are preserved by the `longjmp` function. In ISO C, any automatic variables not declared as `volatile` have undefined values after a return.
- All `extern` declarations are global even if they occur inside a function definition. This includes implicit function declarations.
- Newer keywords such as `typeof`, `inline`, `signed`, `const`, and `volatile` are not recognized. Alternative keywords such as `_typeof_`, `_inline_`, and so on can still be used.
- Comparisons between pointers and integers are always allowed.
- Integer types `unsigned short` and `unsigned char` are always promoted to `unsigned int`.
- Floating-point literals that are out of range for that data type are not an error.
- String constants are stored in writable space and are therefore not necessarily constant.
- The character escape sequences `\x` and `\a` evaluate as the literal characters x and a, rather than being a prefix for the hexadecimal representation of a character and a bell, respectively.

-`traditional-cpp`: Specifying this option when compiling C applications causes GCC to modify the behavior of its internal preprocessor to make it more similar to the behavior of traditional C preprocessors. See the GNU CPP manual for details (<http://gcc.gnu.org/onlinedocs/cpp>).

-trigraphs: Specifying this option while compiling a C application causes GCC to support ISO C trigraphs. The character set used in C source code is the 7-bit ASCII character set, which is a superset of a superset of the ISO 646-1983 Invariant Code Set. Trigraphs are sequences of three characters (introduced by two question marks) that the compiler replaces with their corresponding punctuation characters. Specifying the **-trigraphs** option enables C source files to be written using only characters in the ISO Invariant Code Set by providing an ISO-compliant way of representing punctuation or international characters for which there is no convenient graphical representation on the development system. The **-ansi** option implies the **-trigraphs** option because it enforces strict conformance to the ISO C standard.

The nine standard trigraphs and their replacements are the following:

Trigraph:	??(??)	??<	??>	??=	??/	??'	??!	??-
Replacement:	[]	{	}	#	\	^		~

-V *version*: Specifying this target-related option causes GCC to attempt to run the specified version of GCC if multiple versions are installed on your system. Using this option can be handy if you have multiple versions of GCC installed on your system.

-v: Specifying this output option causes GCC to print the commands executed during each stage of compilation, along with the version number of each command.

-W: Specifying this diagnostic/warning option causes GCC to display extra warning messages when it detects any of the following events in the code that is being compiled:

- A function can return either with or without a value. If a function returns a value in one case, both a return statement with no value (such as `return;`) or an implicit return after reaching the end of a function will trigger a warning.
- The left side of a comma expression has no side effects. (A *comma expression* is an expression that contains two operands separated by a comma. Although GCC evaluates both operands, the value of the expression is the value of the right operand. The left operand of a comma expression is used to do an assignment or produce other side effects—if it produces a value, it is discarded by GCC.) To suppress the warning, cast the left-side expression to `void`.

- An unsigned value is compared against zero using `<` or `<=`.
 - A comparison like `x <= y <= z` appears. GCC interprets this as `((x <= y) < z)`, which compares the return value of the comparison of `x` and `y` against the value of `z`, which is usually not what is intended.
 - Storage-class specifiers like `static` are not the first things in a declaration, which is suggested by modern C standards.
 - A return type of a function has a type qualifier. This has no effect because the return value of a function is not an assigned lvalue.
 - Unused arguments to a function call are present. This warning will only be displayed if the `-Wall` or `-Wunused` options are also specified.
 - A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. To eliminate this warning, specify the `-Wno-sign-compare` command-line option.
 - An aggregate has a partly bracketed initializer, which is usually seen when initializing data structures that contain other data structures. The values passed to an internal data structure must also be enclosed within brackets.
 - An aggregate has an initializer that does not initialize all members of the structure.
- w:** Specifying this diagnostic/warning option causes GCC not to display any warning messages.
- Wabi:** Specifying this option when compiling C++ applications causes GCC to display a warning when it generates code that may not be compatible with the generic C++ application binary interface (ABI). Eliminating warnings of this type typically requires that you modify your code. The most common causes of these warnings are padding related, either when using bit fields or when making assumptions about the length of words or data structures.
- Waggregate-return:** Diagnostic/warning. Not implied by `-Wall`. Warns if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)
- Wall:** Specifying this diagnostic/warning option activates the majority of GCC's warnings. This option is not a reference to one of the authors of this book.

NOTE Though you would assume that using GCC's `-Wall` option turns on all warnings, that is not the case. The following warning-related options are not automatically activated when you specify `-Wall`, and must be separately specified if desired: `-W`, `-Waggregate-return`, `-Wbad-function-cast`, `-Wcast-align`, `-Wcast-qual`, `-Wconversion`, `-Wdisabled-optimization`, `-Werror`, `-Wfloat-equal`, `-Wformat-nonliteral`, `-Wformat-security`, `-Wformat=2`, `-Winline`, `-Wlarger-than-len`, `-Wlong-long`, `-Wmissing-declarations`, `-Wmissing-format-attribute`, `-Wmissing-noreturn`, `-Wmissing-prototypes`, `-Wnested-externs`, `-Wno-deprecated-declarations`, `-Wno-format-y2k`, `-Wno-format-extra-args`, `-Wpadded`, `-Wpointer-arith`, `-Wredundant-decls`, `-Wshadow`, `-Wsign-compare`, `-Wstrict-prototypes`, `-Wtraditional`, `-Wundef`, and `-Wunused-code`. See the explanation of these options in the chapter for details on exactly what additional warnings they will generate.

`-Wbad-function-cast`: Specifying this diagnostic/warning option when compiling a C application causes GCC to display a warning message when a function call is cast to a type that does not match the function declaration. Though this is something that many C programmers traditionally do, using this option can be useful to help you detect casts to data types of different sizes.

`-Wcast-align`: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a pointer is cast such that the target will need to change the alignment of the specified data structure. For example, casting a `char *` to an `int *` on machines where integers can only be accessed at 2- or 4-byte boundaries would generate this warning.

`-Wcast-qual`: Specifying this diagnostic/warning option causes GCC to display a warning message whenever casting a pointer removes a type qualifier from the target type. For example, casting a `const char *` to an ordinary `char *` would generate this warning.

`-Wchar-subscripts`: Specifying this diagnostic/warning option causes GCC to display a warning message whenever an array subscript has type `char`.

`-Wcomment`: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a comment-start sequence `(/*)` appears within another `(/*)` comment or whenever a newline is escaped within a `//` comment. This is an incredibly helpful option to detect most cases of “comment overflow.”

-Wconversion: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This helps identify conversions that would change the width or signedness of a variable, which is a common cause of error on machines with alignment requirements. This option will not generate warnings for explicit casts like `(unsigned) -1` because these will be preresolved during compilation.

-Wctor-dtor-privacy: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning whenever a class seems unusable, because all the constructors or destructors are private and the class has no friends (i.e., grants no access to other classes or functions) or public static member functions.

-Wdisabled-optimization: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a requested optimization pass is disabled or skipped. This rarely indicates a problem with your GCC installation or code, but instead usually means that GCC's optimizers were simply unable to handle the code because of its size, complexity, or the amount of time that the requested optimization pass would require.

-Wdiv-by-zero: Specifying this diagnostic/warning option causes GCC to display a warning message whenever the compiler detects and attempts to divide an integer by zero. You can disable this warning by using the `-Wno-div-by-zero` option. This warning is not generated when an application attempts floating-point division by zero, because this can occasionally be used in applications to generate values for infinity and NaN (Not a Number).

-Weffc++: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever application code violates various guidelines from Scott Meyers' *Effective C++, Second Edition* (Addison-Wesley, 1997. ISBN: 0-201-92488-9.) and *More Effective C++* (Addison-Wesley, 1995. ISBN: 0-201-63371-X.), such as the following:

- Define a copy constructor and an assignment operator for classes with dynamically allocated memory (Item 11, *Effective C++*).
- Prefer initialization to assignment in constructors (Item 12, *Effective C++*).
- Make destructors virtual in base classes (Item 14, *Effective C++*).

- Have operator= return a reference to `*this` (Item 15, *Effective C++*).
- Do not try to return a reference when you must return an object (Item 23, *Effective C++*).
- Distinguish between prefix and postfix forms of increment and decrement operators (Item 6, *More Effective C++*).
- Never overload the operators `&&`, `||`, or `,` (Item 7, *More Effective C++*).

NOTE *Ironically, some of the standard header files used by GCC do not follow these guidelines, so activating this warning option may generate unexpected messages about system files. You can ignore warnings from outside your code base or use a utility such as grep -v to filter out those warnings.*

- Werror: Specifying this diagnostic/warning option causes GCC to make all warnings into errors.
- Werror-implicit-function-declaration: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a function is used before it has been declared.
- Wfloat-equal: Specifying this diagnostic/warning option causes GCC to display a warning message whenever floating-point values are compared for equality. Because floating-point values are often used as approximations for infinitely precise real numbers, it is not always possible to precisely compare such approximation. If you are doing this, a better suggestion is to compare floating-point values by determining if they fall within an acceptable range of values by using relational operators.
- Wformat: Specifying this diagnostic/warning option causes GCC to display a warning message whenever the arguments to calls to `printf`, `scanf`, `strftime` (X11), `strfmon` (X11), and similar functions do not have types appropriate to the specified format string. If the `-pedantic` option is used with this option, warnings will be generated for any use of format strings that are not consistent with the programming language standard being used.
- Wformat=2: Specifying this diagnostic/warning option is the same as explicitly invoking the `-Wformat`, `-Wformat-nonliteral`, and `-Wformat-security` options.

- Wformat-nonliteral: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a format string is not a string literal and therefore cannot be checked, unless the format function takes its format arguments as a variable argument list (`va_list`).
- Wformat-security: Specifying this diagnostic/warning option causes GCC to display a warning message whenever calls to the `printf()` and `scanf()` function use a format string that is not a string literal and there are no format arguments, as in `printf (foo);`. At the time this book was written, this option is a subset of the warnings generated by the -Wformat-nonliteral option, but is provided to detect explicitly format strings that may be security holes.
- Wimplicit: Specifying this diagnostic/warning option is the same as explicitly invoking the -Wimplicit-int and -Wimplicit-function-declaration options.
- Wimplicit-function-declaration: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a function is used before being declared.
- Wimplicit-int: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a declaration does not specify a type, which therefore causes the declared function or variable to default to being an integer.
- Winline: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a function that was declared as inline cannot be inlined.
- Wlarger-than-len: Specifying this diagnostic/warning option causes GCC to display a warning message whenever an object of larger than `len` bytes is defined.
- Wlong-long: Specifying this diagnostic/warning option causes GCC to display a warning message whenever the `long long` type is used. This warning option is automatically enabled when the -pedantic option is specified. You can inhibit these warning messages in this case by using the -Wno-long-long option.
- Wmain: Specifying this diagnostic/warning option causes GCC to display a warning message whenever the type of `main()` or the number of arguments passed to it is suspicious. A program's main routine should always be an externally linked function that returns an integer value and takes either zero, two, or three arguments of the appropriate types.

-Wmissing-braces: Specifying this diagnostic/warning option causes GCC to display a warning message whenever an aggregate or union initializer is not correctly bracketed so that it explicitly follows the conventions of the aggregate or union. As an example, the following expression would generate this warning:

```
int a[2][2] = { 0, 1, 2, 3 };
```

-Wmissing-declarations: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a global function is defined without a previous declaration, even if the definition itself provides a prototype. Using this option detects global functions that are not declared in header files.

-Wmissing-format-attribute: Specifying this diagnostic/warning option for C programs causes GCC to display a warning message whenever a function such as printf() or scanf() contains a format string that contains more attributes than are provided in subsequent arguments to the call. If the -Wformat option is also specified, GCC will also generate warnings about similar occurrences in other functions that appear to take format strings.

-Wmissing-noreturn: Specifying this diagnostic/warning option causes GCC to display a warning message whenever functions are used that might be candidates for the noreturn attribute (`_attribute_((noreturn)) prototype;`).

-Wmissing-prototypes: Specifying this diagnostic/warning option causes GCC to display a warning message whenever compiling a C application in which a global function is defined without a previous prototype declaration, and is intended to detect global functions that are not declared in header files. This warning is issued even if the definition itself provides a prototype.

-Wmultichar: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a multicharacter constant (e.g., "FOO") is used. This option is enabled by default, but can be disabled by specifying the -Wno-multichar option. Multicharacter constants should not be used in portable code because their internal representation is platform specific.

-Wnested-externs: Specifying this diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever an extern declaration is encountered within a function.

- Wno-deprecated: Specifying this diagnostic/warning option causes GCC not to display a warning message whenever deprecated features are used.
- Wno-deprecated-declarations: Specifying this diagnostic/warning option causes GCC not to display a warning message whenever functions, variables, and types marked as deprecated (through the `deprecated` attribute) are used.
- Wno-format-y2k: Specifying this diagnostic/warning option causes GCC not to display a warning message whenever `strftime()` formats are used that may yield only a two-digit year.
- Wno-format-extra-args: Specifying this diagnostic/warning option when -Wformat is also specified causes GCC not to display a warning message whenever excess arguments are supplied to a `printf()` or `scanf()` function. Extra arguments are ignored, as specified in the C standard. Warnings will still be displayed if the unused arguments are not all pointers and lie between used arguments that are specified with \$ operand number specifications.
- Wno-import: Specifying this diagnostic/warning option causes GCC not to display a warning message whenever `#import` statements are encountered in an Objective C application. (The `#import` statement is identical to C's `#include` statement, but will not include the same include file multiple times.)
- Wno-non-template-friend: Specifying this diagnostic/warning option when compiling a C++ application causes GCC not to display a warning message whenever nontemplatized friend functions are declared within a template. The C++ language specification requires that friends with unqualified IDs declare or define an ordinary, nontemplate function. Because unqualified IDs could be interpreted as a particular specialization of a templated function in earlier versions of GCC, GCC now checks for instances of this in C++ code by using the -Wno-non-template-friend option as a default. The -Wno-non-template-friend option can be used to disable this check but keep the conformant compiler code.
- Wno-pmf-conversions: Specifying this diagnostic/warning option causes GCC not to display a warning message whenever C++ disables the diagnostic for converting a bound pointer of a member function to a plain pointer.
- Wno-protocol: Specifying this diagnostic/warning option when compiling an Objective C application causes GCC not to display a warning message if methods required by a protocol are not implemented in the class that adopts it.

-Wno-return-type: Specifying this diagnostic/warning option causes GCC to suppress warning messages whenever a function is defined with a return type that defaults to int, or when a return without a value is encountered in a function whose return type is not void. This option does not suppress warning messages when compiling C++ applications that contain nonsystem, non-main functions without a return type.

-Wno-sign-compare: Specifying this diagnostic/warning option causes GCC to suppress displaying a warning message whenever a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.

-Wnon-template-friend: When compiling C++ applications under earlier versions of GCC, unqualified IDs could be interpreted as a particular specialization of a templatized function. The C++ language specification requires that friends with unqualified IDs declare or define an ordinary, nontemplate function. Specifying this option causes GCC to check for this situation and display an error message. This option is enabled by default.

-Wnon-virtual-dtor: Specifying this diagnostic/warning option when compiling a C++ application causes GCC not to display a warning message whenever a class declares a nonvirtual destructor that should probably be virtual because the class may be used polymorphically.

-Wpacked: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a structure is specified as packed, but the packed attribute has no effect on the layout or size of the structure.

-Wpadded: Specifying this diagnostic/warning option causes GCC to display a warning message whenever padding is included in a data structure, regardless of whether it is used to align a single element or the entire data structure. This warning is displayed because it is often possible to reduce the size of the structure simply by rearranging its components.

-Wold-style-cast: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever a C-style cast to a non-void type is used within a C++ program. Newer cast statements such as const_cast, reinterpret_cast, and static_cast should be used instead because they are less vulnerable to unintended side effects.

- Woverloaded-virtual: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever a function declaration hides virtual functions from a base class, typically because a virtual function with the same name is already present in a base class.
- Wparentheses: Specifying this diagnostic/warning option causes GCC to display a warning message whenever parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, when operators are nested whose precedence is commonly confused, or when there may be confusion about the if statement to which an else branch belongs. Warnings caused by the latter two problems can easily be corrected by adding brackets to explicitly identify nesting.
- Wpointer-arith: Specifying this diagnostic/warning option causes GCC to display a warning message whenever anything depends on the size of a function type or of void. GNU C assigns these types a size of 1 for convenience in calculations and pointer comparisons.
- Wredundant-decls: Specifying this diagnostic/warning option causes GCC to display a warning message whenever anything is declared more than once in the same scope.
- Wreorder: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever the order of member initializers given in the code does not match the order in which they were declared. A warning is displayed to inform you that GCC is reordering member initializers to match the declaration.
- Wreturn-type: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a function is defined with a return type that defaults to int, or when a return without a value is encountered in a function whose return type is not void. When compiling C++ applications, nonsystem functions without a return type (and which are not main()) always produce this error message, even when the -Wno-return-type option is encountered. This option is active by default.
- Wselector: Specifying this diagnostic/warning option when compiling an Objective C application causes GCC to display a warning message whenever a selector defines multiple methods of different types.

-Wsequence-point: Specifying this diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever the compiler detects code that may have undefined semantics because of violations of sequence point rules in the C standard. Sequence point rules help the compiler order the execution of different parts of the program, and can be violated by code sequences with undefined behavior such as `a = a++, a[n] = b[n++], and a[i++] = i;`. Some more complicated cases may not be identified by this option, which may also occasionally give a false positive.

-Wshadow: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a local variable shadows another local variable, parameter, or global variable, or whenever a built-in function is shadowed.

-Wsign-compare: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This option is automatically invoked when you specify the -W option.

-Wsign-promo: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever overload resolution chooses a promotion from an unsigned or enumerated type to a signed type over a conversion to an unsigned type of the same size. Earlier versions of GCC would try to preserve unsignedness.

-Wstrict-prototypes: Specifying this diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever a function is declared or defined without specifying the types of its arguments.

-Wswitch: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a switch statement has an index of an enumerated type but lacks a case statement for one or more of the named values of the type. (Adding a default statement eliminates this warning.) This warning will also be produced if case statements are present whose values are not found in the enumerated type.

-Wsynth: Specifying this diagnostic/warning option when compiling a C++ application causes GCC to display a warning message whenever GCC's synthesis behavior does not match that of Cfront. Cfront is a C++ compiler product family from Software Solutions Products.

-Wsystem-headers: Specifying this diagnostic/warning option causes GCC to display a warning message whenever potentially invalid constructs are found in system header files. Using this command-line option tells GCC to display warnings about system headers as if they occurred in application code. To display warnings about unknown pragmas found in system headers, you must also specify the -Wunknown-pragmas option.

-Wtraditional: Specifying this diagnostic/warning option when compiling a C application causes GCC to display a warning message whenever constructs are encountered that behave differently in traditional and ISO C or are only found in ISO C, and for problematic constructs that should generally be avoided. Some examples of these are the following:

- Macro parameters that appear within string literals in the macro body. Traditional C supports macro replacement within string literals, but ISO C does not.
- Preprocessor directives that do not begin with the hash symbol (#) as the first character on a line. Preprocessor directives such as #pragma that are not supported by traditional C can thus be “hidden” by indenting them. For true portability, you may want to generally avoid preprocessor directives such as #elif that are not supported by traditional C.
- Function-like macros that appear without arguments.
- The U integer constant suffix, or the F or L floating-point constant suffixes.
- Functions that are declared external in one block and subsequently used after the end of the block.
- A switch statement that has an operand of type long.
- Nonstatic function declarations that follow static ones.
- Integer constants. The ISO type of a decimal integer constant has a different width or signedness from its traditional type.
- ISO string concatenation.
- Initialization of automatic aggregates, which are nonstatic local arrays and structures.
- Conflicts between identifiers and labels.
- Union initialization of nonzero unions.
- Prototype conversions between fixed and floating-point values and vice versa. You can use the -Wconversion option to display additional warnings related to possible conversion problems.

- Wtrigraphs: Specifying this diagnostic/warning option causes GCC to display a warning message whenever trigraphs without comments are encountered that might change the meaning of the program.
- Wundef: Specifying this diagnostic/warning option causes GCC to display a warning message whenever an undefined identifier is evaluated in an #if preprocessing directive.
- Wuninitialized: Specifying this diagnostic/warning option causes GCC to display a warning message whenever an automatic variable is used without first being initialized or if an existing nonvolatile variable may be changed by a setjmp call. These warnings are only generated when using the -O, -O1, -O2, or -O3 optimization options, and then only for nonvolatile variables that are candidates for register allocation.
- Wunknown-pragmas: Specifying this diagnostic/warning option causes GCC to display a warning message whenever it encounters an unknown #pragma preprocessing directive.
- Wunreachable-code: Specifying this diagnostic/warning option causes GCC to display a warning message whenever GCC detects code that will never be executed.
- Wunused: Specifying this diagnostic/warning option provides a convenient shortcut for specifying all of the -Wunused-function, -Wunused-label, -Wunused-value, and -Wunused-variable options. In order to get a warning about an unused function parameter, you must either specify the -W and -Wunused options or separately specify the -Wunused-parameter option.
- Wunused-function: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a static function is declared but not defined, or when a noninline static function is not used.
- Wunused-label: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a label is declared but not used.
- Wunused-parameter: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a function parameter is unused aside from its declaration.
- Wunused-value: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a statement computes a result that is not used.
- Wunused-variable: Specifying this diagnostic/warning option causes GCC to display a warning message whenever a local variable or nonconstant static variable is unused aside from its declaration.

-x [language|none]: Specifying the output option identifies the output *language* to be generated rather than letting the compiler choose a default based on the extension of the input file. This option applies to all following input files until the next **-x** option. Possible values for *language* are ada, assembler, assembler-with-cpp, c, c-header, c++, c++-cpp-output, cpp-output, f77, f77-cpp-output, java, objc-cpp-output, objective-c, and ratfor. Specifying none turns off the language specification, reverting to GCC's defaults based on the extension of the input file.

Alphabetical List of GCC Assembler Options

The options discussed in this section are GCC command-line options that affect the GNU assembler, as:

-Wa,option: Causes GCC to pass *option* as an option to the assembler. If *option* contains commas, each comma is interpreted as a separator for multiple *options*.

Alphabetical List of GCC Linker Options

The options discussed in this section are GCC command-line options that affect the default GCC linker, ld.

NOTE *If you want to use GCC as the front end instead of ld, you should use the -Wl,<options> option to pass linker options to ld.*

-llibrary | -l library: Causes the linker to search the library named *library* (*liblibrary.a*) when linking. Object files and libraries are searched based on the order in which they are specified on the linker command line. The only difference between these options and explicitly specifying the name of the library is that these options surround *library* with lib and .a, and search for the specified library in multiple directories.

-lobjc: Required to link Objective C programs.

-nodefaultlibs: Causes the linker not to use the standard system libraries when linking, instead linking only against libraries that are explicitly specified on the command line. The standard startup files will still be used unless you also specify the **-nostartfiles** option.

-nostartfiles: Causes the linker not to use its standard system startup files when linking. The standard system libraries are still used if you also specify the **-nostdlib** or **-nodefaultlibs** options.

-nostdlib: Causes the linker not to use either the standard system startup files or libraries when linking, instead linking only against libraries that are explicitly specified on the command line. The libraries bypassed by this command include `libgcc.a`, a library of internal subroutines that GCC uses to overcome shortcomings of particular machines and to provide special functions that are required by some languages. To use the **-nostdlibs** option but include this library, you can specify the **-lgcc** option.

-s: Causes the linker to remove all symbol table and relocation information from the final executable.

-shared: Causes the linker to produce a shared object that can then be linked with other objects to form an executable. When using this option, you should make sure that all of the shared objects that you will eventually link together were compiled with the same set of **-fpic**, **-fPIC**, or **model** suboption compiler options.

-shared-libgcc | -static-libgcc: Cause the linker to force the use of the shared or static version on `libgcc.a` on systems that provide both. These options have no effect if a shared version of `libgcc.a` is not available. The shared version is generally preferable because this makes it easier to do things like throwing and catching exceptions across different shared libraries.

-static: Causes the linker to prevent linking against shared libraries on systems that support them.

-symbolic: Causes the link to bind references to global symbols when building a shared object and to warn about any unresolved references.

-u *symbol*: Causes the linker to pretend that the symbol *symbol* is undefined in order to force linking of library modules that define it. You can use this option multiple times with different symbols in order to force the loading of additional library modules.

-Wl,*option*: Causes GCC to pass *option* as an option to the linker. If *option* contains commas, each is interpreted as a separator for multiple *options*.

-Xlinker *option*: Causes GCC to pass *option* as an option to the linker, and is often used to supply system-specific linker options that GCC does not recognize. Each *option* is a single token. If you need to pass an option that takes an argument, you must use the **-Xlinker** option twice, once for the option and once for the argument.

Alphabetical List of GCC Preprocessor Options

The options discussed in this section are primarily GCC command-line options that affect GCC's integrated preprocessor, which is run on each C source file before actual compilation. This section also discusses standard GCC options that can be used when invoking the preprocessor directly as `cpp`.

- \$: Specifying this option forbids the use of \$ in identifiers.
- A predicate=answer*: Specifying this option makes an assertion with the predicate *predicate* and answer *answer*, enabling you to force information about the system and computer type to the preprocessor. For example, -A `system-gnu` would tell the preprocessor that the value of the system predicate should be asserted as `gnu`.
- A-predicate=answer*: Specifying this option cancels any assertion with the predicate *predicate* and answer *answer* that you may have previously forced.
- A-: Specifying this option cancels all predefined assertions and all assertions that precede it on the command line, and undefines all predefined macros and all macros that precede it on the command line.
- ansi: Specifying this option when invoking `cpp` from the command line tells the preprocessor which standard the input code should conform to. The GNU preprocessor currently only understands the various C standards.
- C: Specifying this option causes the preprocessor to retain comments from the input files, with the exception of comments within preprocessor directives, which are deleted along with the directive. This option can be useful if you want to examine the output of the preprocessor (preserved by the `-save-temp` option) and need a convenient way for delimiting regions of code with comments so that you can see exactly what the preprocessor is doing with your input code.
- D*name*: Specifying the option causes the preprocessor to define *name* as a macro with a definition of 1.
- D*name=definition*: Specifying this option causes the preprocessor to define *name* as a macro with the definition *definition*. If, for some reason, you wish to define a function-like macro on the command line, you must write its argument list with surrounding parentheses before any equals sign.

NOTE Macro definition (-D) and undefined (-U) options are processed in the order they are specified on the command line. All -imacros file and -include file options are processed after all -D and -U options.

-dchars: Specifying this option produces a variety of different behavior based on the specified *chars*, which must be a single string composed of one or more of the following characters, and must not contain or be preceded by a space:

D: Generates a list of all preprocessor macros defined in your code and their preprocessed expansion, sending the result to stdout.

I: Embeds all #include directives in the preprocessor output stream.

M: Generates a list of the #define directives for all of the macros defined for the preprocessor, including predefined macros. You can generate a list of all predefined macros simply by passing an empty file to the preprocessor using a command such as `cpp -dM emptyfile.c`.

N: Generates a list of the names of all preprocessor macros and the results of their preprocessing, sending the result to stdout.

-fno-show-column: Specifying this option causes the preprocessor to suppress printing column numbers in diagnostics. This option is typically used when you are using another program (such as dejagnu) to process the results of your diagnostics, but that program does not need or want the column numbers.

-fpreprocessed: Specifying this option tells the preprocessor that the input file has already been preprocessed, and therefore suppresses macro expansion, trigraph conversion, escaped newline splicing, processing of most directives, and so on. The preprocessor will still recognize and remove comments. This option is assumed by default if the input has the extensions .i, .ii, or .mi, which are the default extensions produced when running GCC using the `-save-temp`s option.

-ftabstop=*width*: Specifying this option sets the distance between tab stops in the input file, which helps the preprocessor report correct column numbers in warnings or errors. The default value, used when this option is not supplied, is 8. This option is ignored if *width* is less than 1 or greater than 100.

-gcc: Specifying this option defines the `_GNUC_`, `_GNUC_MINOR_`, and `_GNUC_PATCHLEVEL_` macros, which are also defined automatically if you use GCC's `-E` option.

-H: Specifying this option causes the preprocessor to print the name of each header file used during preprocessing. Each name is indented to show how deeply nested that `#include` statement is.

--help: Specifying this option when directly invoking the preprocessor causes it to display a usage message and then exit.

-idirafter dir: Specifying this option causes `cpp` to search `dir` for header files, but only after all directories specified with the `-I` option and the standard system directories have been searched. The specified directory is treated as though it were a system include directory.

-imacros file: Specifying this option causes `cpp` to load all macros defined in the specified `file`, but to discard any other output produced by scanning the file. All files specified by the `-imacros` options are processed before any files specified by using the `-include` option. This enables `cpp` to acquire all of the macros defined in the specified file without including any other definitions that it contains.

-include file: Specifying this option causes `cpp` to process the specified `file` as if it were specified using a `#include "file"` statement in the first line of the primary source file. The first directory searched for `file` is the preprocessor's working directory, which may differ from the directory containing the main source file. If the specified `file` is not found there, `cpp` searches for it through the remainder of the normal include file search chain. If the `-include` option is specified multiple times, the specified files are included in the order that they appear on the command line.

-iprefix prefix: Specifying this option causes `cpp` to use `prefix` as the prefix for subsequent `-iwithprefix` options. If the specified `prefix` is a directory, it should end with a trailing `/`.

-isystem dir: Specifying this option causes `cpp` to search `dir` for header files after all directories specified by the `-I` option have been searched, but before the standard system directories. The specified directory is also treated as a system include directory.

-iwithprefix *dir* | -iwithprefixbefore *dir*: Specifying this option causes `cpp` to append the specified *dir* to any prefix previously specified with the **-iprefix** option, and to add the resulting directory to `cpp`'s search path for include directories. Using the **-iwithprefix** option adds this directory to the beginning of `cpp`'s search path (just as the **-I** option would). Using the **-iwithprefixbefore** option adds this directory to the end of `cpp`'s search path (just as the **-idirafter** option would).

-M: Specifying this option causes `cpp` to generate a rule suitable for use by the `make` program that describes the dependencies of the main source file, rather than actually preprocessing the source files. This `make` rule contains the name of that source file's corresponding output file, a colon, and the names of all included files, including those coming from all **-include** or **-imacros** command-line options. Unless explicitly specified using the **-MT** or **-MQ** command-line options, the name of the object file is derived from the name of the source file in the standard fashion, by replacing any existing extension with the **.o** extension associated with object files. If the rule is extremely long, it is broken into multiple lines whose new lines are escaped using the backslash character (/).

You can use the **-M** option with the **-MF** option to specify the output file, which is recommended if you are also using debugging options such as **-dM** to generate debugging output. Specifying this option also invokes the **-E** option, automatically defining the **_GNUC_**, **_GNUC_MINOR_**, and **_GNUC_PATCHLEVEL_** macros and causing the compilation process to stop after the preprocessing phase.

-MD: Specifying the **-MD** option is equivalent to specifying the **-M -MF *file*** options, except that the **_GNUC_**, **_GNUC_MINOR_**, and **_GNUC_PATCHLEVEL_** macros are not defined and that compilation continues after the preprocessing stage. Since this option does not take an output *file* argument, `cpp` first checks if the name of an output file has been specified using the **-o** option. If so, the output file is created using the basename of that file and replacing any existing extension with the **.d** extension. If not, the name of the output file is derived from the name of the input file, again replacing any existing suffix with the **.d** suffix. Because the **-MD** option does not imply the **-E** option, this option can be used to generate a dependency `make` rule output file as part of the complete compilation process.

NOTE *If the -MD option is used in conjunction with the -E option, any -o option specifies the name of the dependency output file. If used without the -E option, the -o option specifies the name of the final object file.*

-MF *file*: Specifying this option along with the **-M** or **-MM** options identifies the name of a file to which `cpp` should write dependency information. If the **-MF** option is not specified, the dependency rules are sent to the place to which preprocessor output would have been sent.

-MG: Specifying this option along with the **-M** or **-MM** options causes `cpp` to treat missing header files as generated files that should be located in the same directory as the source file. This option also suppresses generating preprocessed output, because a missing header file is considered an error. This option is often used when automatically updating Makefiles.

-MM: Specifying this option causes GCC to generate the same output rule as that produced by the **-M** option, the difference being that the generated rules do not list include files that are found in system include directories or include files that are included from system include files.

-MMD: Specifying this option causes GCC to generate the same output rule as that produced by the **-M** option, the difference being that the generated rules do not list user include files that are found in or other user include files that are included from user include files.

-MP: Specifying this option causes `cpp` to add a fake target for each dependency other than the main file, causing each to depend on nothing through a dummy rule. These rules work around errors that may be generated by the `make` program if you remove header files without making corresponding Makefile updates.

-MQ *target*: Specifying this option causes `cpp` to change the output target in the rule emitted by dependency-rule generation. Instead of following the standard extension substitution naming convention, using this option sets the name of the output file to the name that you specify. Any characters that have special meaning to the `make` program are automatically quoted.

-MT *target*: Specifying this option causes `cpp` to change the output target in the rule emitted by dependency-rule generation. Instead of following the standard extension substitution naming convention, using this option sets the name of the output file to the exact filename that you specify.

-o *file*: When invoking `cpp` directly, specifying this option causes `cpp` to write its output to the specified *file*. This is the same as specifying *file* as the second nonoption argument to `cpp`.

-P: Specifying this option causes the preprocessor to inhibit the generation of line markers in its output, and is usually used when the output from the preprocessor will be used with a program that may not understand the line markers.

-remap: Specifying this command-line option enables special code in the preprocessor that is designed to work around filename limitations in file systems, such as the MS-DOS FAT file system, that only permit short, silly filenames.

-U *name*: Specifying this option causes `cpp` to cancel any previous definition of *name*, regardless of whether it is built in or was explicitly defined using the **-D** option.

-undef: Specifying this option causes `cpp` not to redefine any system-specific macros. Standard system-independent macros are still defined.

-v: Specifying this option when invoking `cpp` from the command line causes `cpp` to display its version number and the search path used for header files before proceeding with preprocessing.

-version | --version: Specifying this option when invoking `cpp` from the command line causes the preprocessor to display its version number. If only one dash is used with the option, preprocessing proceeds normally. If two dashes precede the option, `cpp` exits immediately after displaying its version information.

-w: When invoking `cpp` directly, this option can be used to suppress all warnings.

-W*option*: When invoking `cpp` directly, you can use various standard GCC warning options to activate specific warnings. These have the same definitions as the equivalent standard GCC options. See the definitions of these warnings in the first section of this chapter. Warning options that can be used with `cpp` when it is invoked from the command line are **-Wall**, **-Wcomment**, **-Wcomments**, **-Werror**, **-Wimport**, **-Wsystem-headers**, **-Wtraditional**, **-Wtrigraphs**, and **-Wundef**.

-Wp, *option*: Causes GCC to pass *option* as an option to the preprocessor. If *option* contains commas, each comma is interpreted as a separator for multiple options.

-x *lang*: Specifying this option when invoking `cpp` directly enables you to manually specify the syntax that `cpp` should expect the input file to contain. Supported languages are `assembler-with-c`, `c`, `c++`, and `objective-c`. If you do not specify this option, `cpp` will infer the input language from the extension of the input file (`.c`, `.cc`, `.m`, or `.S`). If the file extension is unrecognized, `cpp` will assume that the input file contains C source code.

GCC Option Reference

The following table groups options together by type and function, providing a convenient way of seeing related options.

Option Type	Options
Assembler options	<code>-Wa,<i>option</i></code>
C language options	<code>-ansi, -aux-info <i>filename</i>, -fallow-single-precision, -fcond-mismatch, -ffreestanding, -fhosted, -fno-asm, -fno-builtin, -fno-builtin-<i>function</i>, -fshort-wchar, -fsigned-bitfields, -fsigned-char, -funsigned-bitfields, -funsigned-char, -fwritable-strings, -no-integrated-cpp, -nostdinc, -std=<i>standard</i>, -traditional, -traditional-cpp, -trigraphs</code>
C++ language options	<code>-falt-external-templates, -fcheck-new, -fconserve-space, -fdollars-in-identifiers, -fexternal-templates, -ffor-scope, -fms-extensions, -fno-access-control, -fno-const-strings, -fno-default-inline, -fno-elide-constructors, -fno-enforce-eh-specs, -fno-for-scope, -fno-gnu-keywords, -fno-implicit-templates, -fno-implicit-inline-templates, -fno-implement-inlines, -fno-nonansi-builtins, -fno-operator-names, -fno-optional-diags, -fno-rtti, -fno-weak, -fpermissive, -frepo, -fstats, -ftemplate-depth-<i>n</i>, -fuse-cxa-atexit, -fvtable-gc, -nostdinc++, -Wabi, -Wctor-dtor-privacy, -Weffc++, -Wno-deprecated, -Wno-non-template-friend, -Wno-pmf-conversions, -Wno-return-type, -Wno-sign-compare, -Wnon-template-friend, -Wnon-virtual-dtor, -Wold-style-cast, -Woverloaded-virtual, -Wreorder, -Wreturn-type, -Wsign-promo, -Wsynth</code>
Code generation options	<code>-fargument-alias, -fargument-noalias, -fargument-noalias-global, -fasynchronous-unwind-tables, -fcall-saved-reg, -fcall-used-reg, -ffixed-reg, -fexceptions, -finhibit-size-directive, -finstrument-functions, -fleading-underscore, -fno-common, -fno-ident, -fno-gnu-linker, -fnon-call-exceptions, -fpack-struct, -fpcc-struct-return, -fpic, -FPIC, -freg-struct-return, -fshared-data, -fshortEnums, -fshort-double, -fstack-check, -fstack-limit-register=<i>reg</i>, -fstack-limit-symbol=SYM, -ftls-model=<i>model</i>, -funwind-tables, -fverbose-asm, -fvolatile, -fvolatile-global, -fvolatile-static</code>
Debugging unnumbered options	<code>-d<i>letters</i>, -dumpsspecs, -dumpmachine, -dumpversion, -fdump-translation-unit[-<i>n</i>], -fdump-class-hierarchy[-<i>n</i>], -fdump-tree-original[-<i>n</i>], -fdump-tree-optimized[-<i>n</i>], -fdump-tree-inlined[-<i>n</i>], -fmem-report, -fpretend-float, -fprofile-arcs, -ftest-coverage, -ftime-report, -g, -glevel, -gcoff, -gdwarf, -gdwarf-1, -gdwarf-1+, -gdwarf-2, -ggdb, -gstabs, -gstabs+, -gvms, -gcoff, -gxcoff+, -p, -pg, -print-file-name=<i>library</i>, -print-libgcc-file-name, -print-multi-directory, -print-multi-lib, -print-prog-name=<i>program</i>, -print-search-dirs, -Q, -save-temps, -time</code>

Diagnostic options	<code>-fdiagnostics-show-location=[once every-line], -fmessage-length=n</code>
Directory search options	<code>-Bprefix, -I-, -Idir, -Ldir, -nostdinc, -nostdinc++, -specs=file</code>
Linker options	<code>-library, -nostartfiles, -nodefaultlibs, -nostdlib, -s, -static, -static-libgcc, -shared, -shared-libgcc, -symbolic, -u symbol, -Wl,option, -Xlinker option</code>
Objective C language options	<code>-fconstant-string-class=class-name, -fgnu-runtime, -fnext-runtime, -gen-decls, -Wno-protocol, -Wselector</code>
Optimization options	<code>-falign-functions=n, -falign-jumps=n, -falign-labels=n, -falign-loops=n, -fbranch-probabilities, -fcaller-saves, -fcprop-registers, -fcse-follow-jumps, -fcse-skip-blocks, -fdata-sections, -fdelayed-branch, -fdelete-null-pointer-checks, -fexpensive-optimizations, -ffast-math, -ffloat-store, -fforce-addr, -fforce-mem, -ffunction-sections, -fgcse, -fgcse-lm, -fgcse-sm, -finline-functions, -finline-limit=n, -fkeep-inline-functions, -fkeep-static-consts, -fmerge-constants, -fmerge-all-constants, -fmove-all-movables, -fno-default-inline, -fno-defer-pop, -fno-function-cse, -fno-guess-branch-probability, -fno-inline, -fno-math-errno, -fno-peephole, -fno-peephole2, -funsafe-math-optimizations, -fno-trapping-math, -fomit-frame-pointer, -foptimize-register-move, -foptimize-sibling-calls, -fprefetch-loop-arrays, -freduce-all-givs, -fregmove, -frename-registers, -freorder-functions, -frerun-cse-after-loop, -frerun-loop-opt, -fschedule-insns, -fschedule-insns2, -fsingle-precision-constant, -fssa, -fssa ccp, -fssa-dce, -fstrength-reduce, -fstrict-aliasing, -fthread-jumps, -ftracer, -ftrapv, -funroll-all-loops, -funroll-loops, --param name=value, -O, -O0, -O1, -O2, -O3, -Os</code>
Output and processing options	<code>-###, -c, -E, --help, -o file, -pass-exit-codes, -pipe, -S, --target-help, -v, -x language</code>
Preprocessor options	<code>-\$, -A predicate=answer, -A predicate[=answer], -A-, -C, -dD, -dI, -dM, -dN, -D macro[=defn], -fno-show-column, -fpreprocessed, -ftabstop=width, -gcc, -H, -idirafter dir, -imacros file, -include file, -iprefix file, -isystem dir, -iwithprefix dir, -iwithprefixbefore dir, -M, -MF, -MG, -MM, -MP, -MQ, -MT, -P, -remap, -U macro, -undef, -v, -version, --version, -Wp,option, -x lang</code>
Target options	<code>-b machine, -V version</code>

Warning options	<code>-fsyntax-only, -pedantic, -pedantic-errors, -w, -W, -Waggregate-return, -Wall, -Wbad-function-cast (C only), -Wcast-align, -Wcast-qual, -Wchar-subscripts, -Wcomment, -Wconversion, -Wno-deprecated-declarations, -Wdisabled-optimization, -Wdiv-by-zero, -Werror, -Wfloat-equal, -Wformat, -Wformat=2, -Wformat-nonliteral, -Wformat-security, -Wimplicit, -Wimplicit-int, -Wimplicit-function-declaration, -Werror-implicit-function-declaration, -Wimport, -Winline, -Wlarger-than-len, -Wlong-long, -Wmain, -Wmissing-braces, -Wmissing-declarations, -Wmissing-format-attribute, -Wmissing-noreturn, -Wmissing-prototypes (C only), -Wmultichar, -Wnested-externs (C only), -Wno-format-extra-args, -Wno-format-y2k, -Wno-import, -Wpacked, -Wpadded, -Wparentheses, -Wpointer-arith, -Wredundant-decls, -Wreturn-type, -Wsequence-point, -Wshadow, -Wsign-compare, -Wstrict-prototypes (C only), -Wswitch, -Wsystem-headers, -Wtraditional (C only), -Wtrigraphs, -Wundef, -Wuninitialized, -Wunknown-pragmas, -Wunreachable-code, -Wunused, -Wunused-function, -Wunused-label, -Wunused-parameter, -Wunused-value, -Wunused-variable, -Wwrite-strings</code>
-----------------	---

CHAPTER 12

Additional GCC Resources

As the most widely used compiler on computer systems today, GCC has a tremendous number of users. It is therefore not surprising that there are a similarly large number of online resources where you can obtain detailed information about GCC, ask questions, read others' questions, share your solutions and expertise, and so on. Many Web sites, mailing lists, and Usenet newsgroups are dedicated to sharing information about GCC, its development status, known problems, bug fixes, and the like.

One caveat that you should remember about the information that you obtain through any free resource (such as the Web, Usenet, etc.) is that you should take it with a grain of salt. Luckily, if you are having a problem using GCC and someone proposes a workaround or solution, it is usually easy enough to test the suggestion and determine whether it resolves the problem that you are having.

This chapter provides an overview of the most popular online sources of information about GCC, explaining how to find them, how to access them, and so on. The Web is a living, breathing place—between the time that this chapter was written and the time you are reading it, many new, information-packed sites may have appeared. Regardless, the sites in this chapter are a good starting point; many of them (such as the GCC-related Usenet newsgroups) have been available for years and will continue to be excellent sources of information in the future.

Usenet Resources for GCC

Long before the commercial availability of the Internet, ARPANET users developed a world-wide distributed discussion system known as Usenet. Usenet consists of a flexible set of newsgroups with names that are classified hierarchically by subject, and is often also referred to as *netnews*. Messages are read from and posted to these newsgroups by people using software generally known as newsreaders, though Web browsers such as Mozilla, Netscape, and Internet Explorer contain built-in software for reading and sending messages to Usenet newsgroups. Sending a message to a newsgroup is generally referred to as *posting a message* to that newsgroup.

In order to access Usenet news, you must specify a news server in your newsreader software or Web browser. A *news server* is a computer system you

have access to that serves as a repository for Usenet newsgroups. Most news servers restrict access to their copies of various newsgroups based on your network address, host name, or some similar mechanism.

After specifying a news server in your browser or newsreader software, you can display a list of the newsgroups that the server keeps a copy of, and can then subscribe to any of the newsgroups that you are interested in. When you subsequently use your browser or newsreader software to view the newsgroup, a list of all messages posted to each newsgroup since the last time you checked them is displayed.

Posting messages to a Usenet newsgroup works much like e-mail, which is why e-mail and newsgroup software is often bundled together in Web browsers. There are two basic types of newsgroups, known as *moderated* and *unmoderated*. Posts made to moderated newsgroups are screened by individuals known as *moderators*, who determine if a message is appropriate to a given newsgroup and submit it to the newsgroup if this is the case. Unmoderated newsgroups are open—they can be posted to by anyone, and it is the responsibility of the person posting a message to only post messages on relevant subjects to these newsgroups. Unmoderated newsgroups therefore have a much higher percentage of spam than moderated newsgroups (which have none if correctly moderated), but messages appear on them much more quickly than they do on moderated newsgroups because the latter require verification of each message.

When you post a message to an unmoderated newsgroup or once a message has been approved for posting on a moderated newsgroup, the news server you are connected to adds a copy of that message to its local repository of messages. It also forwards your message to other news servers that the news server is configured to feed messages to. Long ago, traffic between Unix news servers was done over telephone lines using a protocol called Unix-to-Unix Copy Protocol (UUCP), but the majority of this traffic is now done over the Internet using Network News Transport Protocol (NNTP), which was specified in Internet RFC 977.

Usenet newsgroups not only provide a valuable (though perhaps overwhelming) source of information on a variety of topics, but can also be just plain fun. Over 99,000 Usenet newsgroups are available, on topics ranging from GCC to highly polarized political and sexual topics. The number of Usenet newsgroups that you have access to depends on the number that are carried by the news server that you select.

NOTE For the most part, the Usenet newsgroups associated with GCC receive the same messages that are exchanged on the GCC mailing lists discussed in the next section. One significant advantage of asking for information on and using Usenet newsgroups is that they are always available from a large number of sites, are not subject to transient e-mail outages, and serve as a long-term repository for information that is almost always available somewhere on the Web.

Most Internet service providers (ISPs) make a news server available to their customers. If yours does not, for whatever reason, a number of open news servers are available on the Internet. These freely accessible news servers enable anyone to access the newsgroups that they host, but are often restricted to just letting people read Usenet news or post messages to Usenet newsgroups. You can obtain an up-to-date list of these by doing a Web search for “netnews Public server.” At the time of this writing, one of the best open news servers for reading Usenet news is news-cache0.freenet.de, which provides read-only access to over 23,000 newsgroups. Similarly, one of the best open news servers for posting messages to Usenet newsgroups is available at news.wplus.net. This open news server enables you to post messages to over 62,000 newsgroups.

Selecting Software for Reading Usenet News

Web browsers such as Mozilla, Netscape, and Internet Explorer all include software for reading Usenet news, as does e-mail software such as Pine. Also, a large number of applications are designed specifically for reading Usenet news, many of which are available as open source. Some of the most common open source newsreaders are the following:

- *Agent*: Agent is a popular commercial newsreader from Forte, Inc. that runs on Windows systems. A scaled-down but free version of Agent, known as Free Agent, is also available from Forte. For more information or to download or purchase a copy, see Forte's Web site at <http://www.forteinc.com/agent/>.
 - *Knews*: Knews is an open source graphical newsreader that uses the X Window system. Knews has an easy-to-use, intuitive interface and supports displaying message threads (related sets of messages on a common topic). You can obtain source code or binary versions of Knews from its home page at <http://www.matematik.su.se/~kjj/>.
 - *NewsXpress*: NewsXpress is a popular newsreader that is available free from malch.com, and is designed for Windows 95 and Windows NT-based systems. It is available from their Web site at <http://www.malch.com/nxfaq.html>.
 - *slrn*: slrn (which stands for s-lang read news) is a terminal-oriented open source newsreader that runs on Mac OS X, Windows, and Unix/Linux systems. You can obtain source code or binary versions of slrn from its home page at <http://slrn.sourceforge.net/>.
 - *tin*: tin (which stands for Tass + Iaian's Newsreader) is an open source terminal-oriented newsreader that is quite popular and has an incredible number of options available for fine-tuning its behavior and performance. You can obtain the source code for the latest versions of tin from its home page at <http://www.tin.org/>.
-

The following are the primary GCC-related newsgroups that are available via Usenet. The entry for each newsgroup explains the purpose of the group, indicates which (if any) GNU mailing list it is associated with, and provides an estimate of the amount of traffic you can expect to see. For more information about GCC-related mailing lists, see the next section, “Mailing Lists for GCC.”

- *gnu.gcc*: Due to the hierarchical nature of Usenet newsgroups, this newsgroup is primarily intended as a container for the other GCC newsgroups. You should not post to this list—though you will occasionally see posts appear there, they should actually go into one of the other newsgroups discussed in this section.
- *gnu.gcc.announce*: This newsgroup is intended for distributing announcements and progress reports on GCC. This is a moderated list that is not intended for general discussion, but is restricted to posts made by GCC maintainers. As such, this is a low-traffic list. This newsgroup contains the same messages that are distributed through the info-gcc mailing list.
- *gnu.gcc.bug*: This newsgroup is intended for posting and viewing bug reports for GCC, fixes for reported problems, and suggestions for future improvements to GCC. This is a moderated list. If you are working on GCC, you can post fixes in the form of patches applied to some version of GCC. When posting patches, you should also include sample code that illustrates the problem that your patch resolves. GCC is actively under development all over the world, and it may be that your fix is subsumed in someone else’s—in which case your test code will help demonstrate the correctness of any other patches. When preparing a test case, the most generally accepted form is output from the C preprocessor (cpp) that can be passed directly to the first phase of GCC (cc1). It is also best to provide standard C code (rather than C++ or Objective C), since this reduces the chance that the problem that you are fixing actually occurs in a preprocessing phase of the compiler. This newsgroup contains the same messages that are distributed through the bug-gcc mailing list. This is a low-traffic newsgroup.
- *gnu.gcc.help*: This newsgroup is intended as a forum where people using and installing GCC can ask general GCC questions or for help with specific issues. It is not a forum for reporting problems—those should be posted to the gnu.gcc.bug newsgroup or the bug-gcc mailing list. The difference between a help request and a problem report can be subtle: basically, if GCC does not build on your system, if it does not execute correctly once installed, or if a command-line option does not do what it is supposed to, those sorts of problems should be reported as bugs. (Make sure that you have read this book and the documentation before reporting something as a bug!) This newsgroup contains the same messages that are distributed through the help-gcc mailing list. This is a high-traffic newsgroup.

- *pilot.programmer.gcc*: This newsgroup is intended for use by people who are developing applications for Palm personal digital assistants (PDAs) using GCC. The original Palms were known as *Palm Pilots*, hence the name. Developing applications on platforms with limited memory is a challenge in the first place, and the Palm OS provides some interesting challenges in terms of segmenting applications so that they load and execute correctly. This is a low-traffic newsgroup.

Though there are other groups whose names contain the string “gcc” (such as linux.act.gcc, linux.dev.gcc, list.linux-activists.gcc, and pocunix.mail.linux.gcc, among others), these are generally ghost lists that were created at one time but no longer receive any significant traffic. However, aside from the GCC-specific lists mentioned in the previous section, there are other Usenet newsgroups that contain relevant information that is useful to GCC users. Some of the most useful related lists are the following:

- *gnu.g++.announce*: Analogous to gnu.gcc.announce, this is a moderated newsgroup that is intended for distributing announcements and progress reports on g++, the C++ compiler provided as part of the GNU Compiler Collection. This is a low-traffic list where any news is good news. This newsgroup contains the same messages that are distributed through the info-g++ mailing list.
- *gnu.g++.bug*: Analogous to gnu.gcc.bug, this is a moderated newsgroup where you can report problems with g++ and the g++ debugger gdb+, submit fixes, and propose suggestions for future versions of the g++ compiler. This newsgroup contains the same messages that are distributed through the bug-g++ mailing list. This is a low-traffic list.
- *gnu.g++.help*: Analogous to gnu.gcc.help, this is an unmoderated list where you can ask general questions about g++ or request help with specific problems. This newsgroup contains the same messages that are distributed through the help-g++ mailing list. This is a high-traffic list.
- *gnu.g++.lib.bug*: Analogous to gnu.glibc.bug, this is a moderated list where you can report problems with the g++ library (/usr/lib/libstdc++.so.<version>), submit fixes, and propose suggestions for future extensions to the library. This newsgroup contains the same messages that are distributed through the bug-lib-g++ mailing list. This is a low-traffic list.
- *gnu.gdb.bug*: Analogous to gnu.gcc.bug, gnu.g++.bug, and gnu.g++.lib.bug, this is a moderated list where you can report problems with the GNU debugger, gdb, submit fixes, and propose suggestions for future enhancements to gdb. This newsgroup contains the same messages that are distributed through the bug-gdb mailing list. This is a low-traffic list.

- *gnu.glibc.bug*: Analogous to gnu.g++.lib.bug, this is a moderated list where you can report problems with the GNU C library (libc-<version>.so), submit fixes, and propose suggestions for future extensions to the library. This newsgroup contains the same messages that are distributed through the bug-glibc mailing list. This is a low-traffic list.

Usenet newsgroups provide a convenient, classic way of viewing posts, questions, and responses made by others, and for posting your own questions to unmoderated lists. Accessing Usenet newsgroups to retrieve this information is especially convenient in locations where ISPs charge for e-mail. There are a variety of repositories on the Internet (most notably foo.google.com) where old newsgroups are archived, providing long-term access to information that would otherwise be somewhat transient. Due to the number of posts made to Usenet newsgroups, most news servers periodically delete posts that are older than a specified period that is defined in each news server's configuration files.

If you do not have access to a Usenet news server, have no limitations on the amount of e-mail that you can receive, and want the immediate participation provided by e-mail, you may want to join one or more of the GCC-related mailing lists. The various GCC-related mailing lists are discussed in the next section.

Mailing Lists for GCC

Mailing lists are analogous to the Usenet newsgroups discussed in the previous section, except that posts made to a list are sent to a central address and are then directly forwarded to all of the subscribers to the list. Posts made to most mailing lists can be received one-by-one, as fast as they can be forwarded to subscribers by the list server, or in digest form, where all of the posts made each day are collected and sent as a single daily e-mail message.

Aside from the existence of most of the utilities that all Linux and most *BSD distributions depend on, one indication of the tremendous amount of software that the Free Software Foundation has created and enhanced over the years is the huge number of mailing lists that the Free Software Foundation hosts. Almost every package and GNU utility has its own mailing list, though the amount of traffic on each list varies with the popularity of the utility or package, the amount of change and number of open defects associated with the software, and whether the list is moderated or not. Open, unmoderated mailing lists, such as those where users can ask for help or information on using a specific utility, tend to get a substantial amount of traffic.

You can subscribe to the basic GNU mailing lists through the Web site at <http://mail.gnu.org/mailman/listinfo/>. However, due to their sheer number, the GCC lists are hosted through a different mechanism. To subscribe to the GCC lists, visit the URL <http://gcc.gnu.org/> and scroll down the page until you find the subscription form. This form provides a drop-down list of available GCC-related lists and enables you to specify whether you want to receive posts one-by-one or

in digest format. After clicking the Process That! button, the list will send confirmation e-mail to the e-mail subscription address that you specified. You can generally simply reply to this message, and your subscription to the list will be accepted.

Posting to any of the GCC-related lists is as simple as sending e-mail to *listname@gcc.gnu.org*, where *listname* is the name of the list that you want to post your message to. The names of the lists hosted by *gcc.gnu.org* that you can post to are explained in the next section, “GCC Mailing Lists at *gcc.gnu.org*.” For information about how to be a good party member and not irritate any of the people on the lists (or frustrate yourself), see the section later in this chapter entitled “Netiquette for the GCC Mailing Lists.”

After subscribing to any GCC mailing list, you can unsubscribe by sending mail to the unsubscribe address listed in the confirmation e-mail that you received. Information about unsubscribing is also provided on the Web page at <http://gcc.gnu.org/>.

GCC Mailing Lists at gcc.gnu.org

This section lists the various GCC-related mailing lists that are hosted by *gcc.gnu.org* for discussion and related activities regarding GCC, its components, and associated software packages. Each entry explains the type of information that a list is intended to provide, and highlights any relationship between these mailing lists and the Usenet newsgroups discussed in the previous section.

The *gcc.gnu.org* site hosts two general classes of mailing lists: those intended for public consumption and those that are primarily targeted for internal use by GCC developers, people porting GCC to other platforms, and GCC maintainers. Although all of the mailing lists hosted at *gcc.gnu.org* can be read by anyone, only the lists intended for public consumption can also be posted to by members of the general public. Because the internal lists are targeted towards a relatively small group of focused developers, members of the general public cannot post to these lists. This makes them easier for their intended audience to use, and also eliminates the chance that they are bombarded by spam.

The next two sections summarize the available mailing lists in each of these categories.

Read/Write Mailing Lists

This section lists the GCC mailing lists that are hosted at *gcc.gnu.org* and can be both read and posted to by members of the general public. At the time of this writing, the open mailing lists hosted at *gcc.gnu.org* are the following:

- *gcc*: An open, high-volume list for discussing GCC development and testing issues that are not specifically relevant to any of the other GCC lists discussed in this section. This list is also used for announcing and discussing major changes to the GNU Compiler Collection itself, such as abandoning ports to specific systems or architectures, abandoning specific front ends to GCC, and so on. This newsgroup is essentially a GCC newspaper, keeping you abreast of important topics and trends.
- *gcc-announce*: A moderated, low-volume list where GCC maintainers post announcements about releases or other important events.
- *gcc-bugs*: An open, high-volume list where users can file problem reports, submit fixes, and generally discuss unresolved issues in GCC. The posts made to this mailing list are also forwarded to the `gnu.gcc.bugs` mailing list, but are moderated there.
- *gcc-help*: An open, high-volume list that provides a forum where people can ask for assistance in building and using GCC. All of the posts made to this mailing list are also forwarded to the `gnu.gcc.help` Usenet newsgroup.
- *gcc-patches*: An open, high-volume list to which GCC developers can post and discuss patches to GCC. These patches can apply to any aspect of GCC, from patches to front ends, patches to the GCC core, and even patches to and corrections for the GCC Web pages.
- *gcc-testresults*: An open, moderate-volume list where users of GCC can post test results for different versions of GCC on any platform.
- *java*: An open, high-traffic list for discussing the Java language front end for GCC (`gcj`), the runtime library associated with this front end (`libgcj`), and the development of both. Patches to these should not be posted to this list—instead, they should be posted to the `java-patches` list.
- *java-announce*: A moderated, low-volume list where the maintainers and developers of the Java language front end or runtime library for GCC post announcements about releases or other important events.
- *java-patches*: An open, medium-traffic list for submitting and discussing patches to the Java language front end to GCC and its associated runtime library. Patches to `gcj` and `libgcj` should be submitted to both this mailing list and to the standard `gcc-patches` mailing list.
- *libstdc++*: An open, high-traffic list for discussing the standard C++ library (`libstdc++-v3`) and for posting patches to this library. Patches to this library should be sent to both this list and the `gcc-patches` mailing list.

Read-Only Mailing Lists

As mentioned previously, some of the GCC mailing lists hosted at gcc.gnu.org are intended for a somewhat limited audience and/or are automatically posted to by automated systems such as CVS. As such, they can only be read by the general public and cannot be posted to by most people. This section lists the GCC mailing lists that can only be read by members of the general public.

At the time of writing, the read-only mailing lists hosted at gcc.gnu.org are the following:

- *gccadmin*: A medium-volume list where output from nightly cron jobs run by the gccadmin account on the system gcc.gnu.org is posted.
- *gcc-cvs*: A relatively high-volume list that tracks source code check-ins to the GCC CVS repository. CVS (Concurrent Versions System) is a source code maintenance and tracking system that is widely used on Unix, Linux, and *BSD systems. For more information about CVS, see its home page at <http://www.cvshome.org/>.
- *gcc-cvs-wwwdocs*: A relatively low-volume list that tracks check-ins to the GCC Web pages portion of the GCC CVS repository.
- *gcc-prs*: A relatively high-volume list that tracks problem reports as they are entered into the GNATS database that is used to track these issues. GNU GNATS is an excellent open source bug tracking and reporting system. (GNATS stands for GNats: A Tracking System.) GNATS facilitates problem report management and supports communication with users in a variety of different ways. Each GNATS instance stores problem reports in its own databases and provides tools for querying, editing, and maintaining these databases. For more information about GNATS, see the GNATS home page at <http://www.gnu.org/software/gnats/>. An excellent open source Web interface to GNATS, Gnatsweb, is available at the URL <http://ftp.gnu.org/pub-gnu/gnatsweb/>.
- *gcc-regression*: A medium-volume list where the results from running regression tests on the GCC compilers are posted.
- *java-cvs*: A relatively high-volume list that tracks check-ins to the Java language compiler and runtime portions of the GCC CVS repository. Like the java-prs mailing list, a separate mailing list is provided for these messages because some GCC developers and maintainers are primarily interested in Java, and this makes it easier for them to find relevant posts. Because the Java front end to GCC is just one of a variety of front ends, messages posted to this list are also sent to the standard *gcc-cvs* mailing list.

- *java-prs*: A relatively high-volume list that tracks Java-related problem reports as they are entered into the GCC GNATS database. A separate mailing list is provided for these problem reports because some GCC developers and maintainers are primarily interested in Java, and this makes it easier for them to find relevant posts. Because the Java front end to GCC is just one of a variety of front ends, messages posted to this list are also sent to the standard *gcc-prs* mailing list.
- *libstdc++-cvs*: A relatively low-volume list that tracks source code check-ins to the libstdc++-v3 portion of the GCC CVS repository. Because this is a component of the entire GCC CVS source tree, the messages posted to this list are a proper subset of those that are posted to the *gcc-cvs* mailing list.

As you would expect from a dynamic, open source project that is as widely used as GCC, other mailing lists were available in the past whose contents have subsequently been rolled into one of the mailing lists discussed in this and the previous section. Of these, the one that was perhaps best known was the *libstdc++-prs* mailing list dedicated to problem reports regarding the C++ library used with the GCC front end for C++ and *g++*. This list is no longer active. Any outstanding problem reports for *libstdc++* have been rolled into the standard GCC GNATS database. Old postings to this list are still available at gcc.gnu.org as an archive—the list itself is no longer active or supported.

For additional information about using any of the GCC mailing lists, send a blank e-mail message to listname-help@gcc.gnu.org.

Netiquette for the GCC Mailing Lists

The GCC mailing lists are provided as a community service by the folks at the Free Software Foundation. As such, you should follow a few simple rules to ensure that your messages get there and are well received.

The first rule of posting to one of the GCC lists is “Do not post off-topic messages.” When posting to one of these lists, make sure that your message is relevant to the intent of the list. Similarly, you should refrain from sending the same message to multiple lists—if you choose the right list in the first place, it is the right list; broadcasting your post across multiple lists will probably bring you more derision than it will answers.

A second, equally important, rule is that posts to the GCC lists must adhere to the spirit of open source software. Nothing will fill your mailbox with e-mail flames faster than posting a commercial message on one of the GCC lists. For example, we would be committing cultural and e-mail suicide if we posted a recommendation for this book there. Free Software is free and the source code for it is freely available and must be redistributed. This book, though about GCC, is a commercial item—although it is completely relevant and valuable to any GCC user, posting a note about it on any of the GCC lists would just be wrong.

A third basic rule is “Only post messages in text form.” Many of the people who subscribe to these lists do not use mail clients that support HTML, RTF, or any other bell-and-whistle message format. Text messages are the lowest common denominator and are therefore the preferred format—they can be displayed and read in any e-mail client. Text messages are the preferred format for pure text content, such as code examples or patches.

When posting to the GCC-related mailing lists, you should try to keep your messages under 25,000 characters so that they pass successfully through all Internet mailers. That is a lot of typing, but a relatively small amount of error output if you are experiencing a problem. A better approach is to post a message containing an extract of a problem report, and then sending the complete output to anyone who is willing or able to help you. The GCC lists themselves have a hard limit of 100K per post, which means that you can post messages larger than 25K, but you still have no guarantee that such posts will successfully make it through the chain of mail servers necessary to reach gcc.gnu.org.

NOTE *The gcc-prs list, where summaries of problem reports are posted, accepts messages up to 2MB in size, but it is not an open list.*

Other GCC-Related Mailing Lists

Though the majority of the GCC-centric mailing lists are hosted at gnu.org, other GCC-related mailing lists are hosted at other sites. These generally do not discuss specific GNU GCC packages, but instead focus on specific ports or applications of GCC that are not directly sponsored by the Free Software Foundation. The most interesting of these are the following:

- **COBOL for GCC mailing lists:** Though most people think of GCC primarily in terms of its C, C++, and Java compilers, the GCC core can be used as a compiler for many other programming languages. One of the more interesting of these is COBOL, not only because COBOL is a language that many people only associate with older computer systems, but also because COBOL is still one of the most widely used languages in the computer industry. Three different COBOL for GCC mailing lists are hosted at SourceForge: `cobolforgcc-announce`, a list for announcements about COBOL for GCC; `cobolforgcc-devel`, an unmoderated list for discussing the development of COBOL for GCC, and `cobolforgcc-users`, an unmoderated list for discussing COBOL for GCC. To post to any of these lists, send e-mail to `listname@lists.sourceforge.net`, replacing `listname` with the name of the list that you want to post to. You can subscribe to, unsubscribe from, or view the archives of any of these lists through the URL http://sourceforge.net/mail/?group_id=5709.

- *GCC for Palm OS mailing list:* Conceptually related to the pilot.programmer.gcc newsgroup discussed in the previous section, the GCC for Palm OS mailing list is intended for use by Palm programmers who are using the GCC-base prc-tools (Palm Resource tools) to cross-compile applications for Palm PDAs. Three different GCC for Palm OS lists are hosted at SourceForge: prc-tools-announce, an unmoderated list for release information and announcements related to the PRC tools; prc-tools-cvs, a read-only list that gives changes to the CVS source code archive for the PRC-tools; and prc-tools-devel, a high-traffic list for discussing enhancements and posting patches to the PRC tools. To post to any of these lists, send e-mail to *listname@lists.sourceforge.net*, replacing *listname* with the name of the list that you want to post to. You can subscribe to, unsubscribe from, or view the archives of any of these lists through the URL http://sourceforge.net/mail/?group_id=4429.

World Wide Web Resources for GCC

As you would expect, the best source for an entire spectrum of GCC information is the primary GCC Web site at <http://gcc.gnu.org/>. The primary GCC page contains introductory information and a summary of recent GCC news and announcements. The remainder of the site is organized into sections providing general information about GCC, links to a variety of documentation, links to download locations, general information about GCC development, and links for reporting or perusing defect reports (e.g., bugs). The GCC site is nicely organized and is compatible with both graphical Web browsers and text-oriented browsers such as elinks, links, and lynx. If you are using a VT100 for your Web browsing, we recommend using the links or elinks browsers, as they do a superior job of rendering tables. The links browser's home page is at <http://artax.karlin.mff.cuni.cz/~mikulas/links/>. The elinks browser's home page is at <http://elinks.or.cz/>.

Given the popularity of GCC, there are a number of other sites that provide relevant and useful information about GCC and compiler technology in general. The Web is a transient environment for information, but the resources listed in this section have been around for a while, and will (hopefully) still be available if you ever need to consult them. Some suggested sites are the following:

- *The Compiler Resources Page (<http://www.bloodshed.net/compilers/>):* This page is a general resource for a variety of compilers, including GCC. It lists all known free compilers and provides a mechanism where you can add others to the list if you are aware of one that is missing. It also provides links to a number of compiler construction toolkits, articles and tutorials on compiler technology, and links to other compiler-related sites. Though this Web site is an excellent general resource for compilers and compiler technology, the only GCC-specific information available on this page is a link to the GCC home page and links to several Windows-based ports of GCC.

- *Compiler Resources* (<http://www.eng.auburn.edu/users/langfml/compiler.html>): This page is another excellent general resource for compilers and compiler technology, though it provides little GCC-specific information. This site provides an excellent starting point for investigating compiler and compiler technology-related research projects that are taking place at a variety of universities.

Given that the Web is virtually infinite in size and scope, hundreds of other sites are available that provide information about building, using, and solving problems with GCC. A few seconds with your favorite search engine should result in enough links for a few days of browsing.

Publications About GCC and Related Topics

Of course, we recommend this book as a publication that tells you all you need to know about installing and using GCC, but a number of other publications on GCC are also available. The majority of these are published by the GNU Press, the publishing arm of the Free Software Foundation. The primary Web site for GNU Press is at <http://www.gnu.org/doc/gnupresspub.html>. You can contact them via e-mail at press@gnu.org, by phone at 617-542-2652 (ask for GNU Press), and also via snail mail at the following address:

GNU Press
c/o Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111-1307
USA

The following list describes the other publications that are available on GCC and related topics, summarizes their contents, and provides information about obtaining them:

- *Debugging with GDB: The GNU Source-Level Debugger*, Richard M. Stallman et al. (GNU Press, 2002. ISBN: 1-882114-88-4.): After a compiler, a debugger is a developer's best friend, and the GNU debugger, `gdb`, has more debugging bells and whistles than you can shake a symbol at. This is yet another book that is not specifically about GCC, though it discusses the compilation requirements necessary to compile applications that can be successfully debugged using `gdb`. The GNU debugger enables you to analyze application crashes, monitor application execution, and step through application execution, and provides both local and remote debugging capabilities. GDB was written to work closely with GCC, and supports debugging applications written in the C, C++, Java, Fortran, and assembly languages.

- *GCC—The Complete Reference*, Arthur Griffith (Osborne/McGraw-Hill, 2002. ISBN: 0-07-222405-3.): This book is a good reference for building, installing, and using GCC with a variety of different languages. It also provides summary information about related topics such as GDB, Make, Autoconf, and so on. We, of course, prefer the book you are reading now, but it is always handy to have multiple books on the same subject. This book lags the current release of GCC by a few revisions, but that is to be expected in the exciting, fast-paced world of GCC. For more information about this book, see its entry at the Osbourne/McGraw-Hill Web site at the URL <http://shop.osborne.com/cgi-bin/osborne/0072224053.html>.
- *GNU C Library Reference Manual* (two vols.), Sandra Loosemore et al. (GNU Press, 2001. ISBN: 1-882114-55-8.): This not a GCC-specific book, but discusses the GNU implementation of the standard C libraries (Glibc), which are typically used with GCC. Available from GNU Press, this book discusses the entire spectrum of interfaces available in Glibc, and provides complete reference information, including code examples. Like GCC itself, Glibc has been undergoing a fair amount of evolution and refinement over the last few years, and therefore the actual Glibc version discussed in this book (2.2.x) lags the current release by a few versions. Regardless, the information that the book provides is extremely useful and available nowhere else (unless you read the Glibc source code). This book is available online at the URL http://www.gnu.org/manual/glibc-2.2.3/html_chapter/libc_toc.html.
- *GNU C Programming Tutorial*, Mark Burgess and Ron Hale-Evans (GNU Press, 2003. ISBN: 1-882114-61-2.): While not a GCC-specific book, this tutorial on learning the C programming language uses GCC as its compilation environment. This text is intended for beginning programmers, and may therefore be too primal for existing developers looking for deep insights into GCC and C programming.
- *GNU Make: A Program for Directing Recompilation*, Richard M. Stallman and Roland McGrath (GNU Press, 2002. ISBN: 1-882114-82-5.): Though not specific to GCC, this is an excellent book on the GNU make program that is used to incrementally compile and maintain almost every open source application (and many commercial ones). This book is an excellent resource and is well worth having, especially when using GNU make to manage the compilation of complex software projects. This book is available online at the URL http://www.gnu.org/manual/make-3.79.1/html_chapter/make_toc.html.

- *Using and Porting GNU CC*, Richard M. Stallman (GNU Press, 1999. ISBN: 1-882114-38-8.): No one knows more about GCC than its original author and the head of the Free Software Foundation, Richard Stallman. This book is an excellent book, though not necessarily light reading. The book explains how to install, run, debug, configure, and port the GNU Compiler Collection, and discusses using the C, C++, Objective C, and Fortran front ends for GCC. Given the frequency with which new versions of GCC have been released over the last year or two, the actual GCC version discussed in this book (2.95.3) lags the current release by a few versions, but the information that the book provides is still almost entirely germane. This book is available online at the URL <http://gcc.gnu.org/onlinedocs/> `gcc-2.95.3/gcc_toc.html`.

APPENDIX A

Building and Installing Glibc

THE GNU C LIBRARY, popularly known as Glibc, is the unseen force that makes GCC, most C language applications compiled with GCC on Linux systems, and all GNU/Linux systems themselves work. Any code library provides a set of functions that simplify writing certain types of applications. In the case of the GNU C library, the applications facilitated by Glibc are any C programming language applications. The functions provided by Glibc range from functions as fundamental to C as `printf()` all the way to Portable Operating System Interface for Computer Environments (POSIX) functions for opening low-level network connections (more about the latter later).

Rebuilding and changing the library on which almost every GNU/Linux application depends can be an intimidating thought. Luckily, it is not as problematic or complex as you might think. This chapter explains how to obtain, build, and install newer versions of the GNU C library, discusses problems that you might encounter, and also explains how to work around those problems in order to get your system(s) up and running with newer or alternate versions of the GNU C library that may be present by default on the system(s) that you are using.

NOTE *The most important requirement for upgrading Glibc on your system is having sufficient disk space to download and build the source code, back up files that you want to preserve in case you encounter upgrade problems, and install the new version of Glibc. You should make sure that your system has approximately 200MB of free space in order to build and install Glibc.*

What Is in Glibc?

Because most of the fundamental Un*x, Linux, Hurd, and *BSD applications are written in the C programming language, every Unix-like operating system needs a C library to provide the basic capabilities required in C applications and to provide the system calls that enable C applications to interface with the operating system. The GNU C library, Glibc, is the one true C library in the GNU system and most newer systems with the Linux kernel.

The contents of interfaces provided as part of Glibc have evolved over time and reflect the history of Unix and relevant standards. Glibc provides support for the following standards and major Un*x variants:

- *Berkeley (BSD)*: No one with any history on Un*x systems could be unaware of the enhancements to Un*x that were provided by the Berkeley Standard Distribution. NetBSD, FreeBSD, and even Apple's Mac OS X still carry the flag of many of the networking, I/O, and usability improvements made to Un*x by the University of California at Berkeley and other academic institutions such as Carnegie Mellon University, and long-lived BSD-based Un*x implementations such as Sun Microsystems' SunOS. Glibc supports many of the capabilities found in the 4.2 BSD, 4.3 BSD, and 4.4 BSD Unix systems, and in SunOS. This heightens code compatibility with 4.4 BSD and later SunOS 4.X distributions, which themselves support almost all of the capabilities of the ISO C and POSIX standards.
- *ISO C*: This is the C programming language standard adopted by the American National Standards Institute (ANSI) in "American National Standard X3.159-1989—ANSI C," and later by the International Standardization Organization (ISO) in their "ISO/IEC 9899:1990, Programming languages—C." This is colloquially referred to as the *ISO C standard* throughout the Glibc documentation.

NOTE *The header files and functions provided by Glibc are a superset of those specified in the ISO C standard. If you need to write applications that strictly follow the ISO C standard, you must use the -ansi option when compiling programs with GCC. This will identify any non-ANSI constructs that you might have used accidentally.*

- *POSIX*: This is the Portable Operating System Interface for Computer Environments (ISO/IEC 9945-1:1996), later adopted by ANSI and IEEE as ANSI/IEEE Std 1003. The POSIX standard has its roots in Unix systems, and was designed as a standard that would facilitate developing applications that could be compiled across all compliant Un*x systems. The POSIX standard is a superset of the ISO C standard, adding new functions and extending existing functions in the ISO C standard. If you need to sling applicable acronyms, the Glibc manual states that Glibc is compliant with POSIX, POSIX.1, IEEE Std 1003.1 (and IEEE Std 1003.2, Draft 11), ISO/IEC 9945-1, POSIX.2, and IEEE Std 1003.2. Glibc also implements some of the functionality required in the "POSIX Shell and Utilities" standard (a.k.a. POSIX.2 or ISO/IEC 9945-2:1993).

- **SYSV Unix:** Un*x history began at AT&T Bell Labs. Glibc supports the majority of the capabilities specified in the AT&T “System V Interface Description” (SVID) document, which is a superset of the POSIX standard mentioned earlier.
- **XPG:** The “X/Open Portability Guide,” published by the X/Open Company, Ltd., is a general Un*x standard. This document specifies the requirements for systems that are intended to be conformant Unix systems. Glibc complies with the “X/Open Portability Guide, Issue 4.2,” and supports all of the X/Open System Interface (XSI) and X/Open Unix extensions. This should not be a big surprise, since the majority of these are derived from enhancements to Unix made on System V or BSD Unix, and Glibc is compliant with those.

Today’s Glibc has come a long way from the statically linked version 1.x Glibc of the 1980s. Today, Glibc is a powerful set of shared libraries that is used on hundreds of thousands of computer systems all over the world. Like GCC, Glibc is a living testimonial to the power of open source software and the insight and philanthropy of its designers and contributors.

Alternatives to Glibc

The completeness and power of Glibc has an unfortunate, but not unsurprising, side effect—Glibc is big. Glibc is also a shared library, which has distinct administrative and performance improvements for desktop and server systems, but may not be desirable in all circumstances. The most notable environment in which you may not want all of the overhead of Glibc is in embedded systems running Linux. Linux is becoming incredibly popular for use in embedded systems because of its power, flexibility, and huge code base—and also because using generic Linux in an embedded system eliminates the need for operating system royalty payments.

There are two popular alternatives to Glibc, both targeted for use in embedded systems. The next two sections provide an overview of each of these and information about how to obtain them. These sections do not discuss how to install and use them, as this is somewhat outside the scope of a chapter dedicated to Glibc.

NOTE *Depending upon your application and the system that you are developing for, you may also want to statically link applications in order to conserve memory and eliminate library access time on embedded systems.*

uClibc: The Micro C Library

The micro C library, uClibc, is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but provides almost all of the functionality of Glibc without approaching its size. In most cases, porting applications from Glibc to uClibc simply requires recompiling the source code, linking against uClibc rather than Glibc.

NOTE *The letter u is short for the Greek letter mu, which resembles a lowercase u and is commonly used as the abbreviation for the word micro. The capital C is an abbreviation for the word controller. uClibc is generally pronounced as "yew-see-lib-see," or as "mew-see-lib-see" by the more technically obsessed.*

uClibc is maintained by Erik Andersen (also the author and maintainer of BusyBox, another favorite of embedded Linux developers). uClibc is licensed under the GNU Library General Public License. This license enables you to build and distribute commercial applications using uClibc without requiring that you provide the source code for them—you can still own your own intellectual property if you so desire.

For more information about uClibc, see its home page at <http://www.uclibc.org/>. You can obtain the source code for the latest version of uClibc there or at the uClibc project page at Freshmeat.net (<http://freshmeat.net/projects/uclibc/>). The latest version of uClibc was 0.9.19 at the time this book was written.

Newlib

Newlib is an alternative to Glibc (and uClibc) that was originally developed for use in embedded systems by Cygnus Software (later acquired by Red Hat), and is still maintained by people at Red Hat (Jeff Johnston and Tom Fitzsimmons). With the purchase of Cygnus Software, Red Hat made a substantial move into the embedded development market from which they have subsequently retreated with somewhat oxymoronic statements along the lines of “You can just take your desktop or enterprise Red Hat distribution and scale it down for use on an embedded system.” Though this is much like saying “You can just use your Cadillac as a sports car if you shave down the body and put on smaller tires,” that is their position. This does not detract from the usability and value of Newlib in embedded systems, however.

For more information about Newlib, see its home page at <http://sources.redhat.com/newlib/>. You can also download the source code for

the latest version of Newlib there. Newlib is only distributed as source code—you always have to build it yourself. This is certainly reasonable given its target audience, which uses a variety of processors and architectures on embedded systems, any one of which would be hard to anticipate.

Newlib has some interesting customizations for embedded systems, whose processors often support multiple modes of operation. By default, Newlib builds *multilibs*, which means that it builds multiple versions of Newlib, each of which supports a different set of operational permutations. The most common example of this is endianness, in which processors can use either the high byte or low byte as the significant byte that contains the lowest address. When building Newlib for such processors, the compiler will create libraries in each of these formats. Similarly, building Newlib for processors with additional, optional, features will create libraries that cover each permutation of these options. You can use GCC's `print-multi-lib` option to display a list of all of the library permutations that will be built for Newlib on a specific processor or architecture.

Why Build Glibc from Source?

Like any piece of open source software, particularly one with hundreds of thousands of users, inadvertent quality assurance personnel, and potential contributors, Glibc is continually being enhanced and improved. Even aside from bug fixes (this is software, after all) and performance improvements, general enhancement and improved organization of Glibc is a continuous process.

The latest version of Glibc available at the time of writing is Glibc 2.3.2. As you would expect with shared libraries, systems with Glibc 2.3.x installed should be able to successfully execute programs compiled under any 2.x releases of Glibc. Remember that compatibility only works in one direction—programs compiled on a system running Glibc 2.3.x will not run correctly (or at all) on systems with earlier versions of Glibc.

Improvements to Glibc 2.3.2 from previous versions include the following:

- Increased and improved support for internationalization through locales, many new supported character sets, and Unicode 3.2
- General performance improvements
- New and faster implementations of fundamental features such as `malloc` and regular expressions

As stated in its release announcement, Glibc 2.3.2 is actively supported on the following platforms and system types:

- *i[3456]86-*-gnu*: GNU Hurd on Intel
- *i[3456]86-*-linux-gnu*: Linux-2.x on Intel
- *alpha*-linux-gnu*: Linux-2.x on DEC Alpha
- *powerpc*-linux-gnu*: Linux and MkLinux on PowerPC systems
- *powerpc64-*-linux-gnu*: Linux-2.4.19+ on 64-bit PowerPC systems
- *sparc-*-linux-gnu*: Linux-2.x on SPARC
- *sparc64-*-linux-gnu*: Linux-2.x on UltraSPARC 64-bit
- *ia64-*-linux-gnu*: Linux-2.x on ia64
- *s390-*-linux-gnu*: Linux-2.x on IBM S/390
- *s390x-*-linux-gnu*: Linux-2.4+ on IBM S/390 64-bit
- *sh-*-linux-gnu*: Linux-2.x on Super Hitachi
- *x86-64-*-linux-gnu*: Linux-2.4+ on x86-64

Platform names in this list use standard regular expression syntax—for example, * means any matching string in the current field, and [abc] means “any of a, b, or c.”

The following platforms are identified as being “close to usable,” meaning that testing is not complete and you may therefore encounter bugs (which you should, of course, report to the bug-glibc@gnu.org mailing list discussed later in this appendix):

- **-*-gnu*: GNU Hurd on platforms other than Intel
- *arm-*-linux-gnu*: Linux-2.x on ARM
- *cris-*-linux-gnu*: Linux-2.4+ on CRIS
- *hppa*-linux-gnu*: Linux-2.x on HP/PA
- *m68k-*-linux-gnu*: Linux-2.x on Motorola 680x0
- *mips*-linux-gnu*: Linux-2.x on MIPS

Platforms for which previous versions of Glibc were supported, but whose status is now unknown and untested, are the following:

- *arm-*-none*: ARM standalone systems
- *arm-*-linuxaout*: Linux-2.x on ARM using a.out

For software developers, just staying ahead of (or on top of) the curve is a good argument for upgrading your system to newer versions of Glibc. While the basic set of functions used in C programming is not really going to change radically, implementations of those functions can. Making sure that your software will continue to compile and work correctly on new or upcoming releases of Glibc is “a good thing.”

Much of the work being done in Glibc nowadays consists of internal enhancements that reflect the increasing maturity of the Linux code base and the software development community in general. The most notable example of this is increased support for and emphasis on internationalization. A few years ago, i18n (the commonly used abbreviation for the word *internationalization*) was something that people would get to one of these days. Nowadays, the computing (and Linux) communities are truly international, and well-crafted applications provide intrinsic support for different languages and character sets in messages and graphical displays.

Potential Problems in Upgrading Glibc

Like any shared library, Glibc follows the naming and compatibility conventions discussed in the section “Shared Libraries” in Chapter 8. As a quick recap, the major, minor, and release version numbers of a shared library are updated based on the type of changes made between different versions of the shared library. When a new version of a shared library is released in order to fix bugs in a previous version, only the release number is incremented. When new functions are added to a shared library but existing functions in the library still provide the same interfaces, the minor version number is incremented and the release number is reset. When interfaces in functions in a shared library change in such a way that existing applications cannot transparently call those functions, the major number is updated and the minor and release numbers are reset. Applications that depend on specific interfaces to functions can then still load the shared library with the correct major and minor number at runtime, while applications compiled against a new version of the same shared library (featuring different parameters or a different parameter sequence) to existing functions can link against the version of the shared library with the new major number.

In a nutshell (sorry O'Reilly guys!), this means that it should be trivial to upgrade a system to a new version of Glibc with a higher release or minor number. You can expect to encounter problems when upgrading across major release numbers, because all of your existing applications will have to be relinked with the new Glibc: the basic symbolic links to the “well-known” version of Glibc (/lib/libc.so.6), the shared Linux loader library (/lib/ld-linux.so.2), and the POSIX threads library for your architecture (/lib/arch/libpthread.so.0, such as /lib/i686/libpthread.so.0 on i686 systems) will all have changed with the major version number.

You need not be paranoid when upgrading Glibc on your system, but you should be careful and aware of potential problems. The section “Troubleshooting Glibc Installation Problems” later in this chapter explains some problems that you may encounter and provides solutions and workarounds. Nowadays, upgrading Glibc is nowhere nearly as problematic as it was in earlier, more primeval Linux days, such as when Linux systems converted from using libc version 5 to version 6, or when the executable binary format of all Linux applications changed from a.out to ELF. Those were the bad old days—life is simpler now.

Identifying Which Glibc a System Is Using

If you are considering upgrading to a newer version of Glibc, it is important to be able to determine what version of Glibc your system is actually using. The easiest way to do this is simply to list the main Glibc library on your system. The name of the primary C library on your system is a symbolic link to the version of the C library that is actually being used on your system. A symbolic link is used rather than an explicit filename because, although every C language program that uses shared libraries has to know the name of the primary C library, using a single, well-known name ensures portability of executables that use shared libraries across multiple systems.

The primary Glibc shared library used by C programs on modern systems that use GCC is /lib/libc.so.6. Listing this in long format results in something like the following:

```
$ ls -al /lib/libc.so.6
lrwxrwxrwx 1 root root 14 Jan 14 17:08 /lib/libc.so.6 -> libc-2.2.93.so
```

The system on which this command was executed is running Glibc version 2.2.93. You can verify this by listing the other critical shared object filename on your system, which is the name of the Linux load library, as in the following example:

```
$ ls -al /lib/ld-linux.so.2
lrwxrwxrwx 1 root root 12 Jan 14 17:08 /lib/ld-linux.so.2 -> ld-2.2.93.so
```

As an alternative to listing symlinks, you can also write a small program that executes the Glibc function that is provided to display Glibc version information. Though this does not provide any information beyond what you can infer from the names of the symbolic links, it is more empirical because it queries the library itself rather than installation details. The code to display Glibc version information is the following:

```
#include <stdio.h>
#include <gnu/libc-version.h>
int main (void) {
    puts (gnu_get_libc_version ());
    return 0;
}
```

Compiling and running this on a sample system produces output like the following:

```
$ gcc glibc_version.c -o glibc_version
$ ./glibc_version
2.2.93
```

Simply listing symbolic links or querying the library itself provides simple, high-level information about the version of Glibc that your system is running, but does not actually tell you what is in it. As discussed in the section “Glibc Add-Ons,” Glibc is traditionally compiled with a number of external capabilities, known as *extensions* or *add-ons*, that you retrieve separately and integrate into your Glibc source directory before you begin building Glibc. The next section explains how to get more detailed information about the capabilities provided by the version of Glibc that is installed on your system.

Getting More Details About Glibc Versions

Glibc provides a tremendous amount of functionality in a single library. The sets of functions that it provides by default can be augmented by incorporating add-ons, which are library source code modules that you must obtain separately and integrate into your Glibc source directory before you begin building Glibc. If you are planning to upgrade Glibc on your system(s), knowing the capabilities provided by the version of Glibc that your system is currently running is fairly important. This information enables you to ensure that the updated version of Glibc that you build and install on your system provides the capabilities that the applications that are currently on your system require. As we will discuss in the section “Downloading and Installing Source Code,” the Glibc configuration process actively warns you if some basic add-ons are not present on the version of Glibc that you are configuring.

The easiest way to get detailed information about the version of Glibc that you are running on a particular system is to simply execute the main Glibc shared library (or its symbolic link) from the command line. This provides a wealth of information about the version of Glibc that you are running, and also explains the add-ons/extensions that are integrated into that library. The following is some sample output from a generic Red Hat 8.0 system:

```
$ /lib/libc.so.6
GNU C Library development release version 2.2.93, by Roland McGrath et al.
Copyright (C) 1992-2001, 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 3.2 20020903 (Red Hat Linux 8.0 3.2-7).
Compiled on a Linux 2.4.9-9 system on 2002-09-05.
Available extensions:
    GNU libio by Per Bothner
    crypt add-on version 2.1 by Michael Glad and others
    The C stubs add-on version 2.1.2.
    linuxthreads-0.10 by Xavier Leroy
    BIND-8.2.3-T5B
    NIS(YP)/NSS modules 0.19 by Thorsten Kukuk
    Glibc-2.0 compatibility add-on by Cristian Gafton
    libthread_db work sponsored by Alpha Processor Inc
Report bugs using the 'glibcbug' script to <bugs@gnu.org>.
```

This output shows that the system is indeed running Glibc version 2.2.93, and that it was compiled with a variety of extensions. How standard are these extensions? The following is sample output from a version of Glibc 2.3.2 that we built and installed for one of our systems:

```
$ /lib/libc.so.6
GNU C Library stable release version 2.3.2, by Roland McGrath et al.
Copyright (C) 2003 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 3.2 20020903 (Red Hat Linux 8.0 3.2-7).
Compiled on a Linux 2.4.9-9 system on 2003-03-08.
Available extensions:
    GNU libio by Per Bothner
    crypt add-on version 2.1 by Michael Glad and others
    linuxthreads-0.10 by Xavier Leroy
    BIND-8.2.3-T5B
```

libthread_db work sponsored by Alpha Processor Inc
 NIS(YP)/NIS+ NSS modules 0.19 by Thorsten Kukuk
 Report bugs using the 'glibcbug' script to <bugs@gnu.org>.

Aside from the sequence in which the add-ons are listed, the only difference between these two is the presence of the line Glibc-2.0 compatibility add-on by Cristian Gafton, which is discussed in more detail in the next section.

Glibc Add-Ons

The previous sections introduced Glibc add-ons, which are separate, external functionality that you can integrate into a Glibc build by uncompressed them in your Glibc source directory and then configuring Glibc using the --enable-add-ons command-line option. This section discusses current and former add-ons.

The three primary add-on packages that are suggested for use with current versions of Glibc (2.3.x) are the following:

- A POSIX-compliant thread library, glibc-linuxthreads, used on almost all supported GNU/Linux systems. The libthreads module is so standard that its numbering conventions follow those of Glibc itself. When downloading Glibc in order to compile it for a GNU/Linux system, you should also obtain the version of the archive for the glibc-linuxthreads library that corresponds to the Glibc version that you will be building.
- The Berkeley DB NSS (Name Service Switch) module, previously integrated into earlier versions of the GNU C Library, was broken out into a separate nss-db add-on. This add-on enables you to store host information in Berkeley DB format, and thus to use db as a meaningful entry in the name service configuration file /etc/nsswitch.conf. The nss-db add-on provides support for Berkeley DB releases 2.x and 3.x. If you are still dependent on Berkeley DB 1.x, you can obtain the source code for a compatible version of the library from Sleepycat Software (<http://www.sleepycat.com>), compile it separately as a shared library, and install it in the system library directory (/lib).
- The nss-1wres module supports the lightweight resolver in Berkeley Internet Name Daemon (BIND) 9 on systems that use Glibc. The lightweight resolver is a combination of library functions and a daemon process that reduce name service lookup overhead when compared to full-blown DNS. For more information about the lightweight resolver in BIND 9, see Section 5 of the BIND 9 documentation. A sample link to this documentation is <http://www.csd.uwo.ca/staff/magi/doc/bind9/Bv9ARM.ch05.html>.

You can obtain all of these from the primary Glibc download site, <http://ftp.gnu.org/gnu/glibc>, or any of its mirrors.

If you have built Glibc before, you may notice that some of the add-ons you built with older versions of Glibc are no longer used or required. The most common of these, and the reason(s) why they are no longer used, are the following:

- **glibc-compat:** Glibc 2.1 introduced some fundamental (and incompatible) changes in the Glibc IO library, `libio`. The `glibc-compat` add-on provides a `libcompat.a` library, older `nss` modules, and a few other files that make it possible to do development with old static libraries on a system running Glibc 2.1. Unless you are maintaining an older system or need to be able to continue to run Glibc 2.0 binaries on a modern Linux system, you do not need this add-on. If you still need it, it can be challenging to find, but is certainly still available on the Internet somewhere. Is not everything?
- **glibc-crypt:** Glibc 2.2 and later versions no longer require a separate cryptography add-on. In a rare burst of cluefulness, U.S. Export Regulations were relaxed so that the DES cryptographic functions could be incorporated into the base GNU C library distribution without sending anyone to jail. If anyone knows who was responsible for this breakthrough, perhaps you could have them take a look at similar foolishness like the DCMA and the one-click patent!
- **localedata:** Glibc 2.1 and later versions no longer require a separate module for easily producing internationalized code. Maybe this is one world, after all.

If you are upgrading an existing version of Glibc to a version that still requires the `glibc-crypt` or `localedata` add-ons, you can still obtain them from the primary Glibc download site, <http://ftp.gnu.org/gnu/glibc/>, or any of its mirrors. The latest versions of these add-ons available at the time of writing are the following:

- `glibc-crypt-2.1.tar.gz` (40K)
- `glibc-localedata-2.0.6.tar.gz` (511K)

Now that you know what is in your existing Glibc, the following section provides an overview of the process required to download, configure, build, and install a different Glibc than the one that is already running on your system.

Previewing the Build Process

As explained throughout the remainder of this chapter, the process of building and installing a new version of Glibc on your system is actually fairly simple,

thanks to fine tools from the Free Software Foundation such as Autoconf and Automake, which produce a configure script for Glibc that provides a flexible, automatic configuration interface for Glibc. The general process of building Glibc on your system is as follows.

NOTE *Before beginning the process of replacing or updating Glibc on your system, make sure that you create a rescue disk so that you can still boot your system if something goes wrong during the Glibc installation/update process. For information about building a rescue disk, see the sidebar later in this chapter entitled "Using a Rescue Disk" or online sources of information such as <http://www.linuxplanet.com/linuxplanet/reports/4294> and <http://www.hrlug.org/rescuedisk.html>.*

1. Download and unpack the version of Glibc that you want to upgrade your system to. The version used in the examples in this book is Glibc 2.3.2.
2. Configure the Glibc source code directory for your system by running the configure script with the options that are appropriate for your system.
3. Compile the Glibc source code using the `env LANGUAGE=C LC_ALL=C make` command. This compiles Glibc with standard default locale settings (in the LANGUAGE and LC_ALL environment variables). The LANGUAGE environment variable specifies the language to use in error messages. The LC_ALL environment variable specifies the default font set for displaying those messages.
4. Assuming that Glibc compiled correctly, verify the compiled Glibc using the command `env LANGUAGE=C LC_ALL=C make check`.
5. If you have configured Glibc as a replacement for your system's existing C library and want to upgrade your system, you should then perform the remainder of the steps in this procedure. If you are installing it as an alternate to your system's C library, you are done!
6. Assuming that all of the Glibc tests pass successfully, shut your system down to single user mode (just to be on the safe side—this is mandatory if other users are using the system), and install the new version of Glibc using the `env LANGUAGE=C LC_ALL=C make install` command.

NOTE *On most Linux systems, you can shut your system down to single user mode by executing the command `telinit 1`.*

7. Execute the `ldconfig` command to refresh your system's library cache.

NOTE *Do not panic if you experience problems at this point! See the section "Troubleshooting Glibc Installation Problems" for information about resolving Glibc upgrade problems. A truly nice feature of installing newer versions of Glibc is that the new libraries are installed but the existing libraries are left in place.*

8. Execute a few of your favorite commands to ensure that everything is still working correctly with the new C library.
9. Compile a sample application to ensure that GCC still works correctly.
10. Reboot.

That is all there is to it! Depending on your system's configuration, the process might require relinking or rebuilding your system's library cache, but upgrading Glibc is nowhere nearly as complex as during some of the major Linux library upheavals, such as when Linux systems converted from using libc version 5 to version 6, or when the executable binary format of all Linux applications changed from a.out to ELF. You may also find that you need to relink or update some of the applications that you may have compiled for your system if they have dependencies on misfeatures of previous functions in the Glibc library or related libraries.

Subsequent sections discuss these steps in more detail, walking you through all of the nuances of each stage. Sections where you might encounter a problem contain sidebars discussing these potential problems and provide workarounds.

Recommended Tools for Building Glibc

Each Glibc source code distribution provides a file called `INSTALL`, which contains up-to-date information about building and installing the version of Glibc that it accompanies. This section summarizes the information about suggested, and in some cases required, versions of development utilities that you should have on your system in order to correctly build and install the version of Glibc that was current when this book was written, Glibc 2.3.2. Rather than trying to verify the version of each of these utilities that is installed on your system, we suggest that you consult this section only if you encounter a problem compiling Glibc.

Building and installing Glibc requires that up-to-date versions of the following utilities are installed on your system. The following list discusses suggested minimum versions of various utilities used when building and installing Glibc:

Utility	Description
GNU awk 3.0 (or other POSIX awk)	The awk utility, a feature of Un*x systems since it was written by Aho, Weinberger, and Kernighan at Bell Labs in the 1970s (its name is their initials), is a pattern-matching and data-reformatting language that is used when building Glibc (and most other GNU utilities). GNU awk is compliant with the awk specification in the “POSIX Command Language and Utilities” specification, but you can actually use any POSIX-compliant awk implementation to build Glibc. In most cases, we suggest that you install GNU awk just to be safe—after all, you cannot beat the price! The source for the latest version of GNU awk is its home page at http://www.gnu.org/software/gawk/gawk.html . You can determine the version of awk that is installed on your system by executing the command <code>awk -version</code> .
GNU binutils 2.13 or later	The GNU ‘binutils’ package includes utilities such as <code>as</code> (the GNU assembler) and <code>ld</code> (the GNU linker/loader). You must use these utilities with Glibc—only the GNU assembler and linker/loader have the features required to successfully compile and link applications that use Glibc.
GNU make 3.79 or newer	Like most GNU utilities and libraries, Glibc makes heavy use of the features provided by GNU make in order to simplify configuring, building, managing, and installing Glibc. GNU make is the standard make nowadays anyway, but if you experience problems or error messages after executing the <code>make</code> command to build Glibc, execute the <code>make -v</code> command to determine if you are running GNU make and what version it is. GNU make is truly “a good thing.” You can get the source code for the latest version from its home page at http://www.gnu.org/software/make/make.html . Prepackaged binary versions for most platforms are also available. To find prepackaged Linux binaries, try http://www.rpmfind.net . Solaris versions are available at the impressively useful http://www.sunfreeware.com Web site.
GCC 3.2 or newer	If you are building the version of Glibc used as an example throughout this chapter (2.3.2) or a newer version, you will want to use the latest and greatest version of GCC. Version 2.3 and greater of Glibc require GCC 3.2 or better. We will not insult you by describing GCC in more detail here—you own this book, right? See Chapter 1 for more information about downloading and building the latest version of GCC from source.

Utility	Description
GNU 'sed' 3.02 or newer	Like awk, sed (stream editor) is an ancient and ubiquitous Un*x pattern-matching and transformation utility of which an enhanced GNU version is freely available. We strongly suggest that you get the latest version of GNU sed if you are building Glibc. GNU sed is available from its home page at the URL http://www.gnu.org/software/sed/sed.html . Though most versions of sed should be sufficient to build Glibc, some of the tests for validating Glibc will only work correctly if you use GNU sed. You can determine the version of awk that is installed on your system by executing the command <code>sed -version</code> .
GNU 'texinfo' 3.12f	The GNU texinfo utility, discussed in Chapter 10, is the basis of all of the online help for Glibc and most GNU utilities. The classic online reference format used by all other Un*x/Linux utilities is not the primary help mechanism for GNU utilities—up-to-date help for Glibc (as for GCC) is compiled and installed in “info” format instead. The authors of this book are divided as to whether this is a feature (Wall) or brain damage (von Hagen). Regardless of how you feel, you will need an appropriate version of texinfo to build and install the online documentation for Glibc. See Chapter 10 for more information about texinfo. The source for the latest version of texinfo is its home page at http://www.gnu.org/software/texinfo/texinfo.html .
Perl 5 or better	Perl, the magnificent programming language, can opener, and floor wax written by Larry Wall (and others), is not required in order to build Glibc, but is used when testing Glibc. It is hard to conceive of a modern Un*x system that does not have Perl installed. You can obtain the sources for the latest version of Perl and binary distributions of Perl from the Comprehensive Perl Archive Network site at http://www.cpan.org/ . If you are building Glibc for a relatively recent Linux system, it is highly likely that most of these utilities support the <code>--version</code> command-line option to display version information. If you need to upgrade any of them, the process for downloading, building, and installing them is the same as that for GCC or Glibc.

NOTE *You can compile applications that use Glibc with any modern compiler that you like, but if you are using GCC, we strongly suggest using the latest version of GCC whenever possible. Older versions of GCC have various bugs (specifically in floating-point calculations) that may be triggered by the math library in newer versions of Glibc.*

Updating GNU Utilities

If you are building and installing your own versions of these utilities, you should consider using an appropriate version of the `--prefix` command-line option when configuring them, so that the versions you are building will overwrite any older versions that are available on your system. To determine the correct value for `--prefix`, you can first use the `which` command to determine where the existing utility is installed on your system, and then specify an appropriate argument to `--prefix`. For example, determining where `awk` is installed on your system would look something like the following:

```
$ which awk  
/usr/bin/awk
```

In this case, you would then specify `--prefix=/usr` as an argument to the GNU `awk` `configure` script in order to configure your `awk` source code directory such that executing the `make install` command would overwrite the installed version of `awk`. You could always simply build and install `awk` with its default values, which would install it as `/usr/local/bin/awk`, but you would then have to remember to set your path correctly (and library path correctly, for some of the other utilities) when building Glibc to ensure that the build process uses the “right” version of `awk`.

Downloading and Installing Source Code

This section explains how to download the basic archive files required for building and installing Glibc, and how to extract the contents of those archives.

Downloading the Source Code

The source code for all 2.x versions of Glibc and related add-ons is available at the primary Glibc download site (`ftp://ftp.gnu.org/pub/gnu/glibc`) or one of its mirrors.

You can download the sources and related add-ons using a browser or your favorite FTP client, as explained in the next two sections. Glibc source code archives are available there in gzip and bzip2 formats. Add-ons there are archived in gzip format. Extracting the contents of archive files in both of the gzip and bzip2 formats is explained later in the section “[Installing Source Code Archives](#).”

Downloading the Source Code Using a Web Browser

To download the sources using a browser, open the URL `ftp://ftp.gnu.org/pub/gnu/glibc/` and, holding down the Shift key, left-click the name of the archive file

that you want to download. A dialog box displays, prompting you for the destination of the file—save the file to disk in a directory that you have write access to. We tend to create and use the directory /usr/local/src/glibc; though we build and install Glibc in /usr as the primary C library, putting the source in a local system directory such as /usr/local/src lets other users of the system see what source code has been compiled and installed on the particular system that you are using.

NOTE *Using the Shift+left-click command to explicitly save the file to disk is not required in all browsers, but does not hurt. Some browsers do not know how to handle files with the .bz2 file extension and may try to download the file as text—an ugly (and not very useful) process.*

Unless you have a specific reason to do so, you should always download and build the latest version of Glibc and related add-ons that are available there. At the time that this book was written, the latest version of Glibc was 2.3.2, the source for which is contained in either of the following two archive files:

- glibc-2.3.2.tar.bz2 (13064K)
- glibc-2.3.2.tar.gz (17691K)

The .bz2 extension indicates that the first file is compressed using the bzip2 utility, which offers greater compression than the standard gzip compression utility, and thus is gaining in popularity. Extracting the contents of archives in both of these formats is explained in the next section.

As explained in the section “Glibc Add-Ons,” you will also want to download any Glibc add-ons that you wish to install with Glibc. For current versions of Glibc, the only truly critical add-on for Glibc is the linuxthreads package. You should always download, integrate, and install the linuxthreads package with Glibc unless you have a specific reason not to. As explained later in this section, the Glibc configure script enforces the presence of this add-on—if you do not want to build Glibc with this add-on, you have to explicitly say so by using the configure scripts *amusing* and the aptly named --enable-sanity-checks command-line option.

Because it is so closely tied to Glibc, version numbering of the linuxthreads add-on follows the version numbering scheme used by Glibc itself. Therefore, the latest version of the linuxthreads add-on that was available at the time this book was written was contained in the following archive files:

- glibc-linuxthreads-2.3.2.tar.bz2 (212K)
- glibc-linuxthreads-2.3.2.tar.gz (279K)

Like the source for Glibc itself, archives of the linuxthreads source code are provided that have been compressed using the bzip2 and gzip archive utilities. In most graphical browsers, you can download the source archive that you want by Shift+left-clicking the name of the archive file in the browser window and specifying where you want the archive to be saved.

Downloading the Source Code Using an FTP Client

If you are more comfortable using a file transfer protocol (FTP) client than a browser to download files, you can download source archives from the Free Software Foundation's archive site using anonymous FTP. Any FTP client will work fine—ftp, gftp, lftp, ncftp, sftp, tnftp, or any other that you are comfortable with.

To download archive files using anonymous FTP, use your FTP client to connect to the site `ftp.gnu.org`. When prompted for a login name, enter **anonymous**. When prompted for the password, enter your e-mail address. You can then use the `cd /pub/gnu/glibc` command to change your working directory to the location where Glibc source is archived. We suggest using the `bin` command to explicitly put your FTP client in binary download mode, and to use the `hash` command to displays hash marks for each 1K that you download—that way, you will at least know that something is happening.

Once connected, in the right directory, and configured for binary downloads, you can use the `get` command to retrieve specific files. To display a listing of all available files, use the `ls` command.

You should then retrieve the Glibc source archive and appropriate add-on(s) such as the linuxthreads library, as discussed in the previous section. The following is a sample transcript of an FTP session where only the file `glibc-2.3.2.tar.bz2` was retrieved:

```
$ ftp ftp.gnu.org
Connected to gnudist.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:wvh): anonymous
331 Please specify the password.
Password:
[output deleted]
ftp> cd pub/gnu/glibc
250 Directory successfully changed.
ftp> hash
Hash mark printing on (1024 bytes/hash mark).
ftp> bin
200 Binary it is, then.
ftp> get glibc-2.3.2.tar.bz2
local: glibc-2.3.2.tar.bz2 remote: glibc-2.3.2.tar.bz2
```

```
227 Entering Passive Mode (199,232,41,7,108,162)
150 Opening BINARY mode data connection for glibc-2.3.2.tar.bz2 (13377170 bytes).
[output deleted]
226 File send OK.
13377170 bytes received in 01:02 (207.72 KB/s)
ftp> quit
```

NOTE You may not be prompted for a password in order to transfer files using anonymous FTP—this depends on how the FTP server is currently configured. The hash option is not mandatory, but is a convenience that displays the progress of your download, letting you know that something is actually happening.

Installing Source Code Archives

Once you have downloaded the source code for Glibc and any add-ons that you want to install, unpacking the archives is easy. All Glibc archives are compressed archives that are written in tar (tape archiver) format. The tar utility is a classic Un*x application that is still found on all Un*x systems, but has been reimplemented and extended (as always) by our friends at the Free Software Foundation.

NOTE You must use the appropriate commands explained in this section to extract the contents of both the Glibc source code archive and the archives of any add-ons that you downloaded before you proceed to subsequent sections of this chapter.

The arguments to the commands used to unpack archive files differ slightly depending on whether you are using GNU tar and whether you are extracting the contents of an archive that was compressed with gzip (and therefore have a .gz extension) or bzip2 (which therefore have a .bz2 extension). The gzip compression/decompression scheme is described in RFC 1952, and is a Lempel-Ziv coding (LZ77) with a 32-bit CRC. The bzip and bzip2 compression/decompression scheme is based on the Burrows-Wheeler block-sorting-based lossless compression algorithm, which offers superior compression to gzip but is not as widely used as gzip.

If you are using a Linux system or are working under Cygwin, you are using GNU tar. If you are using a Solaris or other Unix system, you are already making a big commitment to GNU software by installing GCC and Glibc. Why not install GNU tar as well? You can obtain GNU tar from its home page at the URL <http://www.gnu.org/software/tar/tar.html>.

If you are using GNU tar, you can unpack a gzipped archive into the current directory, execute a command like the following:

```
$ tar xzvf <archive-file>
```

The x option tells GNU tar to extract the contents of the specified archive file, the z option tells GNU tar to process the archive on the fly using the gzip compression scheme (in this case decompressing the archive file), the v option specifies verbose output, which lists files and directories as they are extracted from the archive, and the f option identifies the following argument as the name of an archive file.

If you are not using GNU tar, you can extract the contents of any gzipped archive using the gzip program in tandem with the tar program. (If gzip is not installed on your system, get it now. You can download the source code for gzip or precompiled binaries from its home page at <http://www.gzip.org>.)

To extract the contents of a gzipped archive if you are not using GNU tar, execute a command like the following:

```
$ gzip -cd <archive-file> | tar xvf -
```

The c argument to gzip directs output to the standard output stream, and the d option specifies decompression. The arguments to tar have the same meanings as the same arguments to GNU tar. The - tells the tar command to read from standard input rather than from an archive file.

To extract the contents of an archive file that is compressed using bzip2, execute a command like the following:

```
$ tar xjvf <archive-file>
```

The x option tells GNU tar to extract the contents of the specified archive file, the j option tells GNU tar to process the archive on the fly using the bzip2 compression scheme (in this case decompressing the archive file), the v option specifies verbose output, which lists files and directories as they are extracted from the archive, and the f option identifies the following argument as the name of an archive file.

If you are not using GNU tar, you can extract the contents of any bzipped archive using the bzip2 program in tandem with the tar program. To extract the contents of an archive that is compressed with bzip2 if you are not using GNU tar, execute a command like the following:

```
$ bzip2 -cd <archive-file> | tar xvf -
```

The c argument to bzip2 directs output to the standard output stream, and the d option specifies decompression. The arguments to tar have the same meanings as the same arguments to GNU tar. The - tells the tar command to read from standard input rather than from an archive file.

The Linux Standard Base

The Linux Standard Base is an up-and-coming standard for the applications that one should be able to find on Linux systems, where they are located, and the command-line arguments that they should provide. The goal of the Linux Standard Base is to provide a single standard that all Linux distributions can move toward.

One of the items discussed in the Linux Standard Base is the tar program and its options. To avoid an increasing number of options, one for each new compression scheme, the Linux Standard Base specifies that you should define the compression/decompression method as a separate argument, preceded with two dashes, rather than using options such as j (bzip2) or z (gzip). Equivalent commands to the previous few on systems that are compiled with the Linux Standard Base are the following:

```
$ tar xvf <archive-file> --gzip  
$ tar xvf <archive-file> --bzip2
```

This is the command-line structure of the future for tar, separating the actions of external utilities from those specific to tar itself, and simplifying the integration of new, even faster compression mechanisms as they are developed.

Once dearchived, the source code for Glibc requires 106MB. The source code for the directories extracted from the glibc-linuxthreads archive (the directories linuxthreads and linuxthreads_db) requires another 2.25MB.

Integrating Add-Ons into the Glibc Source Code Directory

Before configuring the Glibc source code and actually building Glibc, the last phase in installing the Glibc source code so that Glibc can be built successfully is to integrate the source code for any add-ons that you want to use with Glibc. For more information on Glibc add-ons, see the section “Glibc Add-Ons.”

To integrate the source code for the add-ons that you want to use with Glibc, you simply move the source code directories for these add-ons into the Glibc source directory, as in the following example:

```
$ pwd  
/usr/local/src  
$ ls  
glibc-2.3.2  linuxthreads  linuxthreads_db  
$ mv linuxthreads linuxthreads_db glibc-2.3.2  
$ ls  
glibc-2.3.2
```

Once you have integrated the add-ons for Glibc into the Glibc source code directory, you are ready to configure and then build Glibc!

Configuring the Source Code

As discussed in the introduction to Chapter 7, long ago, on Un*x systems far, far away, building applications that were distributed as source code usually meant manually customizing the applications' Makefiles. Things are much better now. Thanks to the magic of GNU build tools such as Autoconf and Automake, configuring, compiling, and testing Glibc on a variety of systems is both automatic and easy. You do not have to have these tools installed on your system in order to take advantage of their power—the configure script that is used to automatically configure Glibc for your hardware, operating system, and installation preferences was produced using these utilities, and is included with each Glibc source code distribution.

The best way to build Glibc is to create a separate directory (referred to in the Glibc documentation as *objdir*) to hold the final and intermediate results of compiling Glibc. This makes it easy for you to compile and install Glibc without ever changing any of the files in the actual source code directory (just using them, of course). This chapter describes configuring Glibc through this mechanism, using the directory `glibc-build` as the directory where Glibc will actually be compiled.

Where Should I Put Glibc?

By default, Glibc (like most applications and libraries whose configuration scripts are generated using Autoconf and Automake) will install into the appropriate subdirectories of `/usr/local` unless you specify another location for installation.

Installing Glibc into subdirectories of `/usr/local` is the wrong thing to do unless you only want to use it for testing and your system's load library path does not include the `/usr/local/lib` directory. The load library path contains the list (and sequence) of directories that your system's shared library loader will search for

shared libraries. Your load library path may be specified in the LD_LIBRARY environment variable or in a configuration file such as the /etc/ld.so.conf file used on Red Hat Linux systems. If the directory /usr/local/lib is already in your load library path and you install a newer version of Glibc there, your system may get confused about which library to use, though /lib and /usr/lib are supposed to be searched first, regardless of whether they are present in the loader configuration file or the LD_LIBRARY_PATH environment variable.

In order to install Glibc in a location other than /usr/local/lib, you must use the configure script's --prefix command-line option. The syntax of this option is --prefix=<directory>, where <directory> is the directory under which the lib and include directories associated with your new version of Glibc will be created (if necessary), and where the associated components of Glibc will be installed. If you plan to install the version of Glibc that you are building as the primary C library on your system, replacing whatever version of Glibc your system is currently running, you should specify --prefix=/usr.

The Glibc configure script has some built-in safeguards to ensure that you actually think about where you are installing Glibc. If you actually want to install the version of Glibc that you are building in the appropriate subdirectories of the directory /usr/local, you will have to specify the --disable-sanity-checks command-line option. If you attempt to configure Glibc for installation into its default location and do not specify this command-line option to the configure script, you will see a message like the following:

```
*** On GNU/Linux systems the GNU C Library should not be installed into  
*** /usr/local since this might make your system totally unusable.  
*** We strongly advise to use a different prefix. For details read the FAQ.  
*** If you really mean to do this, run configure again using the extra  
*** parameter '--disable-sanity-checks'.
```

Once this message displays, the configure script terminates. You must specify the --disable-sanity-checks option if you are not providing a specific prefix value. Note that specifying this option disables some other sanity checks for Glibc, such as whether add-ons are present, so you should use this option with care. It is a loaded gun.

To configure Glibc for compilation using a separate object directory, do the following:

1. Create the directory in which you want to build Glibc, and where the intermediate object files and final libraries will be created. This directory should typically be at the same level as the glibc-2.3.2 source directory that was extracted from the downloaded archive files, as explained in the previous section of this appendix. An example of this is the following:

```
$ pwd
/usr/local/src
$ ls
glibc-2.3.2
$ mkdir glibc-build
$ ls
glibc-2.3.2  glibc-build
```

2. Change your working directory to the directory you created in the previous step.

```
$ cd glibc-build
```

3. From this directory, execute the configure script found in the Glibc source directory, specifying the appropriate command-line options. The most common options that you will want to use are the `--enable-add-ons` option, to configure Glibc to be built with any add-ons that it detects in the source code directory, and the `--prefix` option. (See the sidebar “Where Should I Put Glibc?” for information about using the `--prefix` option to specify an installation location.)

An example of configuring Glibc to use any add-ons that it finds in its source directory and to eventually replace any existing version of Glibc on your system is the following:

```
$ pwd
glibc-build
$ ../glibc-2.3.2/configure --enable-add-ons --prefix=/usr
[much output deleted]
```

NOTE For a complete list of the options provided by the Glibc configure script, execute the configure script with the `--help` option. For more information about all of these options, see the file named `INSTALL`, located in your Glibc source directory.

One of the most common problems you may encounter when configuring Glibc for compilation is the following message:

```
*** On GNU/Linux systems it is normal to compile GNU libc with the
*** 'linuxthreads' add-on. Without that, the library will be
*** incompatible with normal GNU/Linux systems.
*** If you really mean to not use this add-on, run configure again
*** using the extra parameter '--disable-sanity-checks'.
```

As the message explains, this error message means that you did not download any of the expected Glibc add-ons, specifically the `glibc-linuxthreads` packages, or that you did not correctly move the directories extracted from the downloaded archives into your Glibc source code directory. Once this message displays, the `configure` script terminates. If you actually want to build Glibc without any of its standard add-ons (specifically, without threading support), you must specify the `--disable-sanity-checks` option on the `configure` script's command line. Note that specifying this option disables some other sanity checks for Glibc, such as the “correctness” of the installation locations, so you should use this option with care. Again, this option is a loaded gun.

Once the `configure` script completes successfully (which can take quite a while), you are ready to proceed to the next section and actually begin building Glibc.

Kernel Optimization Support During Glibc Configuration

In Glibc 2.3.2, the `configure` script for Linux systems introduced a new command-line option, `--enable-kernel=<version>`, where `<version>` is a specific kernel version or a supported keyword. Configuring Glibc using this option strips out compatibility code that is still present in Glibc to support older versions of the Linux kernel. The majority of this compatibility code is present in a number of frequently used functions. While compiling Glibc with this option may not substantially reduce the size of Glibc, it can provide performance improvements and a reduction in code path length. To use this option, specify `--enable-kernel=current` as shorthand for the current kernel version on the `configure` script's command line.

Be careful! You should only configure Glibc with this option if you will never run a kernel on your system whose version is older than the one that is running when you compile Glibc. If you compiled Glibc with this option and attempt to start up with an older kernel, your system may not boot.

Compiling Glibc

Once you have successfully configured Glibc, building Glibc is easy. If you are following the suggested procedure for using a separate build directory as described in this appendix, you need only make sure that the build directory is your working directory, and execute a command like the following:

```
$ env LANGUAGE=C LC_ALL=C make
```

The `env` command sets the `LANGUAGE` and `LC_ALL` environment variables to appropriate values for the default locale and character set, and then uses these values when executing the `make` command. The `LANGUAGE` environment variable specifies the language to use in error messages. The `LC_ALL` environment variable specifies the default font set for displaying those messages. The values shown in this sample command are appropriate for our location, which is in the continental United States. If this is not appropriate for you, you may want to change the value of these variables to reflect your default locale and character set.

TIP *If you are building Glibc on a multiprocessor system with a reasonable amount of memory, you may want to take advantage of GNU make's ability to execute multiple compilations in parallel. You can do this by using the `make` command's '`-j`' option, following it by the number of build commands to execute in parallel.*

NOTE *If you are unfamiliar with internationalization issues, a bewildering number of potentially relevant standards and documents are available on the Web. Given that internationalization is a community, by definition, a good place to start for information on open standards for internationalization under Linux is the OpenI18N.org site at <http://www.li18nux.org/>. A good collection of general links on internationalization topics is available at the i18nGurus Web site's documentation section at <http://www.i18ngurus.com/docs>.*

Once you have started building Glibc, the next step is to be patient. Depending upon the speed of your system, disk drives, and the system load, building Glibc can take quite a while. On an unloaded 500 MHz Celeron system with 256MB of memory and running Red Hat Linux 8.0, building Glibc 2.3.2 took 4 hours and 45 minutes. On an unloaded 1.7 GHz Athlon system with 1GB of memory and the same version of Linux, building Glibc was a tad faster, taking approximately 30 minutes.

NOTE *If the Glibc build exits with an error when creating the `ld.map` file, this indicates that you are running a version of gawk that has problems in some of the `system()` calls that it uses to create and rename files. To resolve this problem, download, build, and install the latest version of gawk from <http://www.gnu.org/software/gawk/gawk.html>, and restart your build of Glibc.*

Testing the Build

All Glibc source code distributions include a complete set of tests that you can use to automatically verify that your new version of Glibc has been built correctly. Testing Glibc before you install it is just plain smart, because the libraries that are built as part of Glibc, the standard C library, the loader library on Linux systems, and the Linux threads library, are the fundamental shared libraries used by all applications compiled with Glibc. This is especially critical on Linux systems, where all system applications outside the kernel depend on the standard C library and also require the loader to load shared libraries in the first place.

Once Glibc has built successfully on your system, you can automatically run all of the included Glibc tests by executing the `make check` command. The Glibc Makefile's `check` target compiles a number of sample applications, links them using your new version of Glibc, and then executes them to verify the correctness of the new Glibc.

Running `make check` may take a while. Once it completes successfully, you are ready to install your new version of Glibc. If it does not complete successfully, and you are sure that your system is configured correctly, you may want to consider asking a question on one of the Glibc newsgroups, or even filing a bug against Glibc as described in the last section of this appendix.

Installing Glibc

The instructions described in this and subsequent sections have always worked for us when upgrading Glibc—your mileage may, unfortunately, vary. Before upgrading your system, back up your critical data (never a bad idea). A Glibc upgrade does not modify your file systems other than to install new libraries, but can still render your system unbootable or unusable because critical applications may turn out to be dependent on the vagaries of your old Glibc. Worst case, it is almost always possible to quickly revert to your previous Glibc, but the operative syllable in the word *software* is still *soft*. It is impossible to predict every scenario. Like a boy scout, “Be Prepared”!

Assuming that your new version of Glibc compiled correctly and passed all of its verification tests, you are now ready to install it on your system. This only takes a few minutes.

NOTE Before attempting to install your new version of Glibc, especially if you have configured and compiled it to replace your system's current version of Glibc, you should first make sure that you have the tools necessary to recover from any problems encountered during the installation process. Most Linux applications use shared libraries, including basic utilities such as ls and ln that you will need if you experience problems during the upgrade.

The key to correcting most Glibc upgrade problems is to make sure that you have access to statically linked versions of the ls, rm, and ln utilities or some other statically linked program that provides the same functionality. Most Linux distributions do not provide static versions of these utilities, instead providing a utility such as BusyBox, which is (usually) a statically linked program that provides the same functionality as utilities such as ls, rm, ln, and a host of others. Originally developed for use on embedded Linux systems, where disk space and memory is at a premium and having a single executable that can do the job of many separate utilities is "a good thing," BusyBox is also a supremely useful tool when recovering from a host of system problems.

If you cannot find a version of BusyBox on your system (in other words, if /sbin/busybox does not exist and the shell's which busybox command does not return anything), see the sidebar entitled "Using a Rescue Disk" later in this chapter. This sidebar provides information about downloading and creating a bootable disk that you can use to resolve upgrade problems. You should, of course, do this before upgrading Glibc, unless you have another Linux system handy (or a Windows system, if you are really desperate).

The remainder of this section explains how to install the version of Glibc that you have just built and tested as your primary C library, how to install it as an alternate C library for use in development or testing, and how to correct and/or recover from problems that you may experience during the installation.

Installing Glibc As the Primary C Library

This section explains how to install a new version of Glibc as a replacement for the existing version of Glibc that is installed on your system. If you have configured and built Glibc to be an alternate C library (in other words, if you configured and compiled it with no prefix or a prefix other than /usr), see the section, "Installing an Alternate Glibc."

Assuming that Glibc has been built correctly on your system and has passed all of the tests described in the previous section, you are ready to install it on your system. Before doing so, you should take a few precautions—replacing the primary shared libraries that are used by almost all applications on your system is fairly major, as upgrades go.

Before replacing your system's existing Glibc with a new version, do the following:

- Verify that the BusyBox utility is installed on your system. If it is not, either install it or prepare a rescue disk that provides it as described in the sidebar "Using a Rescue Disk" later in this appendix.
- Copy or otherwise back up your existing /usr/include directory. If the installation fails and you want to revert to your previous Glibc, you will want to be able to return to the set of include files associated with that version of the C library. You can back up your /usr/include directory using a command like the following:

```
$ cd /usr ; tar czvf include.tgz include
```

- Print a copy of your system's /etc/fstab or at least write down the disk partition corresponding to your root file system, as shown by the `df` command.

Once you have completed these precautions, upgrading your Glibc is simply a matter of executing the `make install` command and sitting back as the `make` command does the dirty upgrade work for you.

Once `make install` completes, it is quite easy to verify whether the upgrade was successful—just type `ls` or almost any other Linux command. If the command executes normally, congratulations—you have upgraded successfully!

If, instead, you receive a message like the following:

```
relocation error: /lib/i686/libc.so.6: symbol __libc_stack_end,
version Glibc_2.1 not defined in file ld-linux.so.2 with link time reference
```

do not panic! This sort of error means that the upgrade did not complete successfully. If you followed the instructions at the beginning of this section about installing BusyBox or creating a rescue disk, it should only take a few minutes to correct the problem and have your system up (and upgraded) and running.

Using a Rescue Disk

If your system does not provide a version of BusyBox and cannot run any program that uses shared libraries after upgrading to a new version of Glibc, the easiest way to repair your system is to reboot your system from a disk known as a *rescue disk*, located on removable media such as a floppy disk or CD. Such rescue disks are freely available on the Internet, and are designed to help you boot failed systems, resolve or work around common problems, and quickly restore your system to self-sufficiency.

My favorite rescue disks are the RamFloppy rescue disk (a bootable floppy-based rescue disk) and the RIP rescue disk (a bootable CD, though a floppy-based version is also available). These are available on the Internet from Kent Robotti's Web page at [http://www.tux.org/pub/people/kent-robotti/looplinux/rip/index.html](http://www.tux.org/pub/people/kent-robotti/loopllinux/rip/index.html). The RamFloppy and RIP floppy rescue systems support ext2, ext3, is9660, ntfs, umsdos, ReiserFS, and VFAT file systems, making it easy for you to mount and access these types of file systems on the computer that you are having problems with. They provide BusyBox, as well as a variety of file system management tools if you are experiencing disk corruption on your computer system. The RIP bootable rescue CD provides all of the tools on the floppy systems, but adds support for the JFS and XFS journaling file systems, as well as the fsck.jfs and xfs_repair utilities that can be used to repair corrupted file systems of those types.

To build a floppy or CD containing these rescue systems, check the README files on the Web page for instructions on downloading and installing the rescue system you are interested in. To download RamFloppy and create a bootable rescue disk on a Linux system, download the file ramf-118.exe and uncompress its contents using the `unzip -L ramf-118.exe` command. You should then use `cd` to go to the ramflop directory, insert a blank floppy disk in your system's floppy drive, and execute the `mkrescue.sh` shell script to create a bootable rescue floppy. Your system must have support for the msdos file system—if the `mkrescue.sh` command complains about an unsupported file system type, try executing the `insmod msdos` command as the superuser to load the correct kernel module, and then rerunning the `mkrescue.sh` shell script. Explaining how to recompile your kernel to add support for msdos file systems is outside the scope of this sidebar.

If you are building a rescue disk on a Windows system, open a command window and execute the ramf-118.exe file to extract its contents. Change the directory to the directory created by the unzip process, and execute the `mkrescue.bat` file to create a bootable rescue floppy for your Linux system.

Once you have created a rescue floppy or CD, you can feel quite virtuous, as though you have just purchased computer system insurance. Put the rescue disk in a safe (but memorable) place, and continue with the upgrade. See the section "Resolving Upgrade Problems Using a Rescue Disk" for information about actually using a rescue disk to repair your system, should that be necessary.

NOTE *Most of the rescue disks available for Linux systems are for x86-based Linux systems. If you are using a PPC-based Linux distribution such as Yellow Dog, you can boot from the distribution disk and select Rescue Mode to perform the recovery procedures described later in this appendix.*

Troubleshooting Glibc Installation Problems

This section focuses on resolving problems encountered after a failed upgrade to a newer version of Glibc on Linux systems, where Glibc is the primary C library.

If your system cannot run any program that uses shared libraries after upgrading Glibc, there is no need to panic—problems are generally easy to resolve once your blood pressure has returned to normal.

As explained in the note at the beginning of the section “Installing Glibc,” the key to correcting most Glibc upgrade problems is to make sure that you have access to statically linked versions of the ls and ln utilities or some other statically linked program such as BusyBox that provides the same functionality. If you cannot find a version of BusyBox on your system (in other words, /sbin/busybox does not exist and the shell’s which busybox command does not return anything), see the sidebar entitled “Using a Rescue Disk” in the previous section for information about downloading a bootable floppy disk image and creating a boot floppy that you can use to boot your system, mount hard drive partitions corresponding to your system’s / partition, and correct several symbolic links.

The next two sections explain how to correct most common Glibc upgrade problems using BusyBox or a rescue disk, respectively. The third section explains how to back out of your Glibc upgrade, if you discover that you have critical applications that no longer execute or now execute incorrectly because they are dependent on specific features of your previous version of Glibc.

Resolving Upgrade Problems Using BusyBox

If the system that you are upgrading has a copy of BusyBox installed, you are in luck—you can resolve the types of problems described in the previous section using BusyBox, usually without rebooting your computer system.

On x86 systems, installing a new version of Glibc on your system involves updating four primary libraries, all of which are actually symbolic links to a specific version of the associated Glibc library. These library links are the following:

```
/lib/libc.so.6
/lib/ld-linux.so.2
/lib/i686/libc.so.6
/lib/i686/libm.so.6
```

These are usually symbolic links to appropriate libraries associated with the version of Glibc that a system is running. For example, on a stock Red Hat 8.0 system, these entries would point to the following libraries:

```
/lib/libc.so.6      -> /lib/libc-2.2.93.so
/lib/ld-linux.so.2  -> /lib/ld-2.2.93.so
/lib/i686/libc.so.6 -> /lib/i686/libc-2.2.93.so
/lib/i686/libm.so.6 -> /lib/i686/libm-2.2.93.so
```

Similarly, on a system that has been upgraded to Glibc 2.3.2, these links would look like the following:

```
/lib/libc.so.6      -> /lib/libc-2.3.2.so
/lib/ld-linux.so.2 -> /lib/ld-2.3.2.so
/lib/i686/libc.so.6 -> /lib/i686/libc-2.3.2.so
/lib/i686/libm.so.6 -> /lib/i686/libm-2.3.2.so
```

Errors of the following form generally mean that one or more of the symbolic links for the critical Linux libraries has not been correctly updated:

```
relocation error: /lib/i686/libc.so.6: symbol __libc_stack_end,
version Glibc_2.1 not defined in file ld-linux.so.2 with link time reference
```

Note that this error message indicates a discrepancy between /lib/i686/libc.so.6 and /lib/ld-linux.so.2.

In most cases, to correct a failed upgrade, you simply have to set the symbolic links correctly, which can be tricky if you do not have access to utilities such as ls (to see what state everything is in), rm (to remove incorrect symbolic links), and ln (to create the correct symbolic links). Enter BusyBox, the favorite utility of everyone in this circumstance!

TIP When you cannot execute any standard utility, you can still use some of the bash shell's built-in commands, most notably the cd built-in, to change directories, and the echo built-in as a surrogate version of the ls command, to help you find your copy of BusyBox. For example:

```
# cd /sbin
# echo busy*
busybox busybox.anaconda
```

Looking at the system that generated the sample error message shown previously (using BusyBox, since ls was broken at this point), you can examine the symbolic links:

```
# busybox ls -l /lib/libc.so.6 /lib/ld-linux.so.2 /lib/i686
lrwxrwxrwx    1 root      root   14 Jan 14 17:08 /lib/libc.so.6 -> libc-2.3.2.so
lrwxrwxrwx    1 root      root   12 Jan 14 17:08 /lib/ld-linux.so.2 -> ld-2.2.93.so

/lib/i686:
total 21944
drwxr-xr-x      2 root      root          4096 Mar  9 09:34 .
drwxr-xr-x      9 root      root          8192 Mar 22 09:45 ..
```

```
-rwxr-xr-x      1 root      root      1395734 Sep  5  2002 libc-2.2.93.so
-rwxr-xr-x      1 root      root      18701992 Mar  9  02:49 libc-2.3.2.so
lrwxrwxrwx  1 root      root      13 Mar  9  02:50 libc.so.6 -> libc-2.3.2.so
-rwxr-xr-x  1 root      root      170910 Sep  5  2002 libm-2.2.93.so
-rwxr-xr-x  1 root      root      1102227 Mar  9  02:49 libm-2.3.2.so
lrwxrwxrwx  1 root      root      13 Mar  9  02:51 libm.so.6 -> libm-2.3.2.so
-rwxr-xr-x  1 root      root      1037065 Mar  9  07:35 libpthread-0.10.so
lrwxrwxrwx  1 root      root      18 Mar  9  09:34 libpthread.so.0 -> libpthread-0.10.so
```

In this case, the symbolic link /lib/ld-linux.so.2 still points to the wrong version of the Linux loader. You can correct this using the following commands:

```
# cd /lib
# busybox rm ld-linux.so.2
# busybox ln -s ld-2.3.2.so ld-linux.so.2
```

At this point, you should be able to execute the standard ls command again.

NOTE If one of the symbolic links created during installation is incorrect, you should check all of them to ensure that they all point to shared libraries from the same version of Glibc. In other words, if correcting the symbolic links in /lib corrects your problem, you should still check the symbolic links in the /lib/i686 directory to ensure that they point to the right files. If one of them is incorrect but others are correct, you will experience apparently spontaneous failures in some commands, while others will work fine.

Resolving Upgrade Problems Using a Rescue Disk

If you ignored our previous pleas to create a rescue disk and keep it handy during the Glibc upgrade process, we are somewhat disappointed but not surprised. Both of the authors learned how useful rescue disks can be through bitter and time-consuming experience. Rescue disks exist because things occasionally go wrong and people occasionally need to be rescued.

The sidebar “Using a Rescue Disk” explained the idea of rescue disks, where to get our personal favorites, and how to use the instructions and command scripts that come with them to produce a bootable floppy or CD rescue disk.

Since you are reading this section, you must find yourself in some circumstance where you need to use your rescue disk. To boot your system from the rescue disk, reboot your system with the rescue floppy in the floppy drive (or the rescue CD in your CD drive). If your system does not boot from the floppy or CD, make sure that your system’s BIOS is configured to attempt to boot from the floppy drive (usually identified as the A drive) or the CD drive (usually identified as CD) before attempting to boot from your hard drive.

Once you have booted from the rescue floppy or CD, you can log in as root (no password is necessary) and mount the partition corresponding to the root file system on your hard drive using a command like the following:

```
# mount -t ext2 /dev/hda2 /mnt
```

You should consult the copy of your `/etc/fstab` file that we suggested you print off earlier to identify the disk partition corresponding to your system's root partition `/`.

Once you've mounted the partition of your hard drive that corresponds to your system's root file system (`/`), you can then access the files and directories on that hard drive partition and correct the symbolic links associated with Glibc, as described in the section "Resolving Upgrade Problems Using BusyBox."

After correcting these links, change directory to `/` (the floppy or CD) and unmount the hard disk partition using a command like the following:

```
# umount /mnt
```

You should then be able to reboot your system, which should now reboot normally, using your new version of Glibc.

If you are still experiencing problems executing commands, you are probably sick of all this upgrade stuff, and wish that you could just put your system back the way it was and worry about upgrading Glibc in your next lifetime. For information about backing out of an attempt at upgrading Glibc, see the next section.

Backing Out of an Upgrade

Backing out of a Glibc upgrade is very similar to correcting problems with an upgrade, except that you are changing the symbolic links associated with Glibc to point to the libraries associated with your old version of Glibc rather than ensuring that they all point to the libraries associated with your new version of Glibc. As a refresher, the basic libraries associated with Glibc are the following:

- `/lib/libc.so.6`
- `/lib/ld-linux.so.2`
- `/lib/i686/libc.so.6`
- `/lib/i686/ibm.so.6`

If you are not sure what version of Glibc you were using before attempting the upgrade, you can determine which ones are available by using BusyBox or

the utilities on your rescue disk to list the contents of one of the relevant directories, as in the following example:

```
# busybox ls -l /lib/i686
total 21932
-rwxr-xr-x 1 root root 1395734 Sep 5 2002 libc-2.2.93.so
-rwxr-xr-x 1 root root 18701992 Mar 9 02:49 libc-2.3.2.so
lrwxrwxrwx 1 root root 13 Mar 9 02:50 libc.so.6 -> libc-2.3.2.so
-rwxr-xr-x 1 root root 170910 Sep 5 2002 libm-2.2.93.so
-rwxr-xr-x 1 root root 1102227 Mar 9 02:49 libm-2.3.2.so
lrwxrwxrwx 1 root root 13 Mar 9 02:51 libm.so.6 -> libm-2.3.2.so
-rwxr-xr-x 1 root root 1037065 Mar 9 07:35 libpthread-0.10.so
lrwxrwxrwx 1 root root 18 Mar 9 09:34 libpthread.so.0 -> libpthread-0.10.so
```

In this case, you can see that Glibc 2.3.2 is the current version, but that the appropriate files for Glibc 2.2.93 are also available, indicating that this was the version of Glibc that was previously used on your system.

In this case, to return your system to using your previous version of Glibc, you would use commands like the following:

```
# cd /lib
# busybox rm ld-linux.so.2
# busybox ln -s ld-2.3.2.so ld-linux.so.2
# busybox rm libc.so.6
# busybox ln -s libc-2.2.93.so libc.so.6
# cd /lib/i686
# busybox rm libc.so.6
# busybox ln -s libc-2.2.93.so libc.so.6
# busybox rm libm.so.6
# busybox ln -s libm-2.2.93.so libm.so.6
```

At this point, you should be able to execute any of the commands that you could execute before you began your upgrade.

When reverting to a previous version of Glibc, you should also restore your previous version of the /usr/include directory, so that the include files therein match the version of Glibc that you are using. We asked you to back this up at the beginning of the section “Installing Glibc As the Primary C Library.” To restore this, you could use commands like the following:

```
# cd /usr
# mv include include.bad
$ tar xzvf include.tgz
```

The actual command that you have to execute depends on the name of the archive file that you created. Once the restore completes successfully, you can delete the directory associated with the failed upgrade (include.bad in the previous example). You may want to keep your backup archive around just in case—such as if you decide to try upgrading Glibc after whatever utility dependencies or bugs you encountered have been resolved.

Installing an Alternate Glibc

This section highlights issues that you may encounter if you have configured, built, and installed Glibc as an alternate version of your system's C library—in other words, if you configured Glibc using its default installation prefix of /usr/local or any prefix other than /bin.

The most common case in which you may want to install an alternate version of Glibc is when you are doing development and/or testing, and wish to ensure that your application works correctly with other versions of Glibc than the primary one that is installed on your system.

There are a few common problems with installing an alternate version of Glibc. These are listed in order by the number of times that we have shot ourselves in the foot “using” these mechanisms. Each potential problem is followed by an explanation of how to avoid that problem:

- Accidentally compiling critical applications using that version of Glibc and then needing to execute them before the partition where your alternate version of Glibc lives is mounted

FIX: Do not compile applications used during the system boot process with any version of Glibc other than your primary one. If you must do this for some reason, ensure that these applications are only executed after all of your partitions are mounted, or ensure that your alternate version of Glibc is installed somewhere on your system's root (/) partition, which is the first partition mounted during the boot process (aside from an initial RAM disk, if you are using one).

- Accidentally compiling critical applications using an alternate version of Glibc and then removing it

FIX: Always keep backup copies of such applications, so that you can boot from a rescue disk and restore them if necessary.

- Modifying the library specification files used by GCC to automatically use an alternate version of Glibc and then removing the alternate version of Glibc

FIX: Do not do this! If you want to be able to automatically compile and link against an alternate Glibc, building a separate version of GCC that knows about the new version of Glibc is a much better idea than hacking the spec files of an existing GCC to use your new Glibc. Hacking the specification files used by an existing version of GCC is simply asking for trouble unless you are a true wizard, in which case you should not be making any other errors.

Getting More Information About Glibc

As you would expect nowadays, a wealth of additional information about Glibc is available on the Web and Internet in general. This section provides an overview of additional sources of information about Glibc, including mailing lists where you can submit problem reports or simply ask for help if the information in this chapter does not suffice.

Glibc Documentation

Not surprisingly, the one true source of information about Glibc is its primary Web site (<http://www.gnu.org/software/libc/>). This site provides the following types of information:

- Glibc Frequently Asked Questions (FAQ) file at the URL <http://www.gnu.org/software/libc/FAQ.html>.
- General status information and links to release announcements on the Glibc home page (<http://www.gnu.org/software/libc/>).
- Various versions of the Glibc manual. For example, the Glibc 2.2.5 manual is available online at the URL <http://www.gnu.org/manual/glibc-2.2.5/libc.html>.

NOTE *To generate the manual that corresponds to the version of Glibc that you are building and installing, execute the `make dvi` command from a configured Glibc installation directory. If you are using the `objdir` build model (with a separate directory for your object code and configuration files), the documentation output files will still end up in the `manual` subdirectory of the directory where you installed the Glibc source code.*

- Information about porting Glibc to other platforms is available at the URL <http://www.gnu.org/software/libc/porting.html>.

Other Glibc Web Sites

A simple Web search using your favorite search engine will show you thousands of messages about Glibc and Web sites that discuss every possible Glibc problem, suggestion, bug, and workaround. Besides the standard Glibc Web site, one Web site that always provides a good deal of information about Glibc is Red Hat's Glibc site, which is available at the URL <http://sources.redhat.com/glibc>.

Glibc Mailing Lists

A number of mailing lists about Glibc are hosted at Red Hat and the primary GNU Web site. You can subscribe to these lists using the forms available at <http://sources.redhat.com/glibc/> or at <http://www.gnu.org/software/libc/>.

Mailing lists related to Glibc are the following:

- *bug-glibc*: A relatively high-traffic list to which you should report problems with Glibc. You can view archives of this list at the URL <http://sources.redhat.com/ml/bug-glibc/>.
- *glibc-cvs*: A relatively high-traffic list showing modifications to the Glibc source code archive stored in the CVS source code control system. You can view archives of postings to this list at the URL <http://sources.redhat.com/ml/glibc-cvs/>.
- *libc-alpha*: Discusses issues in porting and supporting Glibc on a variety of platforms. You can view archives of posts to this list at the URL <http://sources.redhat.com/ml/libc-alpha/>.
- *libc-announce*: A low-traffic list to which announcements of new releases of Glibc are posted. You can view archives of posts to this mailing list at the URL <http://sources.redhat.com/ml/libc-announce/>.
- *libc-hacker*: A closed list in which details of Glibc development and porting are discussed by the Glibc maintainers. Mere mortals cannot post to this list, but can view archives of postings to this list at the URL <http://sources.redhat.com/ml/libc-hacker>.

Reporting Problems with Glibc

As mentioned in the previous section, you should report Glibc problems to the bug-glibc mailing list. You can submit problem reports via e-mail to bug-glibc@gnu.org. You can also view archives of postings made to this list at the URL <http://sources.redhat.com/glibc>.

The actual Glibc bug database is available online at the URL <http://bugs.gnu.org/cgi-bin/gnatsweb.pl>. This is the ultimate and easiest-to-use source of information about known problems in different versions of Glibc. You should check this database before submitting a problem report, not only to avoid duplication, but also to see if the problem that you are experiencing has already been fixed.

APPENDIX B

Machine and Processor-Specific Options for GCC

As discussed in Chapter 11, GCC provides hundreds of machine-specific options that you will rarely need to use unless you are using a specific platform and need to take advantage of some of its unique characteristics. This appendix provides a summary and discussion of machine-specific options for GCC, organized by the platform to which they are relevant.

Machine and architecture-specific configuration information for GCC is stored in the `gcc/config` subdirectory of a GCC source code installation. Each supported system has its own directory that contains general configuration information as well as specific information for supported variants of that processor or architecture.

The architecture and system-specific configuration files provided with GCC 3.3.2 are the following:

- *alpha*: alpha.md, ev4.md, ev5.md, ev6.md
- *arc*: arc.md
- *arm*: arm.md
- *avr*: avr.md
- *c4x*: c4x.md
- *cris*: cris.md
- *d30v*: d30v.md
- *dsp16xx*: dsp16xx.md
- *fr30*: fr30.md

- *frv*: frv.md
- *h8300*: h8300.md
- *i370*: i370.md
- *i386*: athlon.md, i386.md, k6.md, pentium.md, ppro.md
- *i960*: i960.md
- *ia64*: ia64.md
- *ip2k*: ip2k.md
- *m32r*: m32r.md
- *m68hc11*: m68hc11.md
- *m68k*: m68k.md
- *m88k*: m88k.md
- *mcore*: mcore.md
- *mips*: 5400.md, 5500.md, mips.md, sr71k.md
- *mmix*: mmix.md
- *mn10200*: mn10200.md
- *mn10300*: mn10300.md
- *ns32k*: ns32k.md
- *pa*: pa.md
- *pdp11*: pdp11.md
- *romp*: romp.md
- *rs6000*: altivec.md, rs6000.md, spe.md
- *s390*: s390.md

- *sh*: sh.md
- *sparc*: cypress.md, hypersparc.md, sparc.md, sparclet.md, supersparc.md, ultra1_2.md, ultra3.md
- *stormy16*: stormy16.md
- *v850*: v850.md
- *vax*: vax.md
- *xtensa*: xtensa.md

The majority of the machine- and CPU-specific options available in GCC are specific values for GCC's `-m` command-line option, which enables you to identify characteristics of the machine for which GCC is generating code.

NOTE *The options discussed in this section are only relevant if you are using a version of GCC that was either compiled to run directly on the specified platform (not always possible) or if you are using a version of GCC that has been built as a cross-compiler, generating binaries for the specified platform even though it is actually executing on another platform.*

Alpha Options

The 64-bit Alpha processor family was originally developed by Digital Equipment Corporation (DEC) and inherited by its purchasers, Compaq Computer and (later) Hewlett-Packard. The Alpha was an extremely fast processor for its time whose widespread adoption was hampered by VMS and DEC's series of one-night stands with a variety of flavors of Unix.

GCC options available when compiling code for Unix-like operating systems running on the DEC Alpha family of processors are the following:

-malpha-as: Specifying this option tells GCC to generate code that is to be assembled by the DEC assembler.

-mbuild-constants: Specifying this option causes GCC to construct all integer constants using code that checks to see if the program can construct the constant from smaller constants in two or three instructions. If it cannot, GCC outputs the constant as a literal and generates code to load it from the data segment at runtime. Normally, GCC only performs this check on 32- or 64-bit integer constants. The goal of this option is to keep constants out of the data segment and in the code segment whenever possible. This option is typically used when building a dynamic loader for shared libraries, because the loader must be able to relocate itself before it can locate its data segment.

-mbwx: Specifying this option tells GCC to generate code to use the optional BWX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mcix: Specifying this option tells GCC to generate code to use the optional CIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mcpu=CPU-type: Specifying this option tells GCC to use the instruction set and instruction scheduling parameters that are associated with the machine type *CPU-type*. You can specify either the chip name (EV-style name) or the corresponding chip number. If you do not specify a processor type, GCC will default to the processor on which the GCC was built.

Supported values for *CPU-type* are the following:

`ev4 | ev45 | 21064`: Schedules as an EV4 without instruction set extensions

`ev5 | 21164`: Schedules as an EV5 without instruction set extensions

`ev56 | 2164a`: Schedules as an EV5 and supports the BWX instruction set extension

`pca56 | 21164pc | 21164PC`: Schedules as an EV5 and supports the BWX and MAX instruction set extensions

`ev6 | 21264`: Schedules as an EV6 and supports the BWX, FIX, and MAX instruction set extensions

`ev67 | 21264a`: Schedules as an EV6 and supports the BWX, CIX, FIX, and MAX instruction set extensions

-mexplicit-relocs: Specifying this option tells GCC to generate code that explicitly marks which type of symbol relocation should apply to which instructions. See the discussion of the `-msmall-data` and `-mlarge-data` options for additional information related to symbol relocation when the `-explicit-relocs` option is specified. This option is essentially a workaround for older assemblers that could only do relocation by using macros.

-mfix: Specifying this option tells GCC to generate code to use the optional FIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mfloat-ieee: Specifying this option tells GCC to generate code that uses IEEE single and double precision instead of VAX F and G floating-point arithmetic.

-mfloat-vax: Specifying this option tells GCC to generate code that uses VAX F and G floating-point arithmetic instead of IEEE single and double precision.

-mfp-reg: Specifying this option causes GCC to generate code that uses the floating-point register set. This is the default.

-mfp-rounding-mode=*rounding-mode*: Using this option enables you to specify the IEEE rounding mode used to round off floating-point numbers. The *rounding-mode* must be one of the following:

c: Chopped rounding mode. Floating-point numbers are rounded toward zero.

d: Dynamic rounding mode. A field in the floating-point control register (FPCR, see the reference manual for the Alpha architecture and processors) controls the rounding mode that is currently in effect. The C library initializes this register for rounding towards plus infinity. Unless your program modifies this register, a *rounding-mode* of **d** means that floating-point numbers are rounded towards plus infinity.

m: Round towards minus infinity.

n: Normal IEEE rounding mode. Floating-point numbers are rounded towards the nearest number that can be represented on the machine, or to the nearest even number that can be represented on the machine if there is no single nearest number.

-mfpu-trap-mode=trap-mode: Specifying this option controls what floating-point related traps are enabled. *trap-mode* can be set to any of the following values:

n: Normal. The only traps that are enabled are the ones that cannot be disabled in software, such as the trap for division by zero. This is the default setting.

su: Safe underflow. This option is similar to u *trap-mode*, but marks instructions as safe for software completion. (See the Alpha architecture manual for details.)

sui: Safe underflow inexact. This enables inexact traps as well as the traps enabled by su *trap-mode*.

u: Underflow. This enables underflow traps as well as the traps enabled by the normal *trap-mode*.

-mgas: Specifying this option tells GCC to generate code that is to be assembled by the GNU assembler.

-mieee: Specifying this option causes GCC to generate fully IEEE-compliant code except that the INEXACT-FLAG is not maintained. The Alpha architecture provides floating-point hardware that is optimized for maximum performance and is generally compliant with the IEEE floating-point standard. This GCC option makes generated code fully compliant.

When using this option, the preprocessor macro _IEEE_FP is defined during compilation. The resulting code is less efficient but is able to correctly support denormalized numbers and exceptional IEEE values such as not-a-number and plus/minus infinity.

-mieee-conformant: Specifying this option tells GCC to mark the generated code as being IEEE conformant. You must not use this option unless you also specify -mtrap-precision=i and either -mfpu-trap-mode=su or -mfpu-trap-mode=sui. Specifying this option inserts the line .eflag 48 in the function prologue of the generated assembly file. Under DEC Unix, this line tells GCC to link in the IEEE-conformant math library routines.

-mlarge-data: Specifying this option causes GCC to store all data objects in the program's standard data segment. This increases the possible size of the data area to just below 2GB, but may require additional instructions to access data objects. Programs that require more than 2GB of data must use malloc or mmap to allocate the data in the heap instead of in the program's data segment.

NOTE When generating code for shared libraries on an Alpha, specifying the `-fPIC` option implies the `-mlarge-data` option.

-mmax: Specifying this option tells GCC to generate code to use the optional MAX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mmemory-latency=*time*: Specifying this option tells GCC how to set the latency that the scheduler should assume for typical memory references as seen by the application. The specified value supplied for *time* depends on the memory access patterns used by the application and the size of the external cache on the machine. Supported values for *time* are the following:

number: A decimal *number* representing clock cycles.

`L1 | L2 | L3 | main`: Use internal estimates of the number of clock cycles for typical EV4 and EV5 hardware for the Level 1, 2, and 3 caches, as well as to main memory. (The level 1, 2, and 3 caches are also often referred to as Dcache, Scache, and Bcache, respectively.) Note that L3 is only valid for EV5.

-mno-bwx: Specifying this option tells GCC not to generate code to use the optional BWX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mno-cix: Specifying this option tells GCC not to generate code to use the optional CIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mno-explicit-relocs: Specifying this option tells GCC to generate code that follows the old Alpha model of generating symbol relocations through assembler macros. Use of these macros does not allow optimal instruction scheduling.

-mno-fix: Specifying this option tells GCC not to generate code to use the optional FIX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mno-fp-reg: Specifying this option causes GCC to generate code that does not use the floating-point register set. When the floating-point register set is not used, floating-point operands are passed in integer registers as if they were integers and floating-point results are passed in '\$0' instead of '\$f0'. This is a nonstandard calling sequence, so any function with a floating-point argument or return value called by code compiled with `-mno-fp-reg` must also be compiled with that option. Specifying this option implies the `-msoft-float` option.

-mno-max: Specifying this option tells GCC not to generate code to use the optional MAX instruction set. The default is to use the instruction sets supported by a CPU type specified by using the `-mcpu=CPU-type` option, or the instruction sets supported on the CPU on which GCC was built if the `-mcpu=CPU-type` option was not specified.

-mno-soft-float: Specifying this option tells GCC to use hardware floating-point instructions for floating-point operations. This is the default.

-msmall-data: Specifying this option causes GCC to store objects that are 8 bytes long or smaller in a small data area (the `sdata` and `sbss` sections). These section are accessed through 16-bit relocations based on the `$gp` register. This limits the size of the small data area to 64K, but allows variables to be directly accessed using a single instruction. This option can only be used when the `-explicit-relocs` option has also been specified.

NOTE *When generating code for shared libraries on an Alpha, specifying the `-fpic` option implies the `-msmall-data` option.*

-msoft-float: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

NOTE Ironically, Alpha implementations without floating-point operations are still required to have floating-point registers.

-mtrap-precision=trap-precision: Floating-point traps are imprecise in the Alpha architecture; without software assistance, it is impossible to recover from a floating trap, and programs that trap must usually be terminated. Specifying this option causes GCC to generate code that can help operating system trap handlers determine the exact location that caused a floating-point trap. The following levels of precision can be specified as the value of *trap-precision*:

- f: Function precision. The trap handler can determine the function that caused a floating-point exception.
- i: Instruction precision. The trap handler can determine the exact instruction that caused a floating-point exception.
- p: Program precision. The trap handler can only identify which program caused a floating-point exception. This is the default.

-mtune=CPU-type: Specifying this option tells GCC to set only the instruction scheduling parameters based on the specified *CPU-type*. The instruction set is not changed. Possible values for *CPU-type* are the same as those that can be specified using the `-mcpu=CPU-type` option.

Alpha/VMS Options

Many workstations using Alpha processors from DEC (and later Compaq Computer or Hewlett-Packard) shipped with a quaint operating system called VMS. For years, VMS was the only operating system for DEC computers for which support was officially available from DEC.

The sole GCC option available when compiling code for Alpha systems running VMS is the following:

-mvms-return-codes: Specifying this option causes GCC to return VMS error codes from `main`, rather than the default POSIX-style error codes used by the majority of the known universe.

AMD x86-64 Options

The options in this section can only be used when compiling code targeted for 64-bit AMD processors (the GCC x86-64 build target). For the discussion of options that can only be used on i386 and AMD x86-64 systems, see the section of this appendix entitled “i386 and AMD x86-64 Options.” For the discussion of options that can only be used on IA-64 (64-bit Intel) systems, see the section of this appendix entitled “IA-64 Options.”

GCC options available when compiling code for 64-bit AMD systems are the following:

-m32: Specifying this option tells GCC to generate code for a 32-bit environment. The 32-bit environment sets int, long, and pointer to 32 bits and generates code that runs on any i386 system.

-m64: Specifying this option tells GCC to generate code for a 64-bit environment. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits and generates code for AMD’s x86-64 architecture.

-mcmodel=kernel: Specifying this option tells GCC to generate code for the kernel code model. The kernel runs in the negative 2GB of the address space. This model must be used for Linux kernel code.

-mcmodel=large: Specifying this option tells GCC to generate code for the large code model. This model makes no assumptions about addresses and sizes of sections. This option is reserved for future expansion—GCC does not currently implement this model.

-mcmodel=medium: Specifying this option tells GCC to generate code for the medium code model. This means that the program is linked in the lower 2GB of the address space, but symbols can be located anywhere in the address space. Programs can be statically or dynamically linked, but building shared libraries is not supported by this memory model.

-mcmodel=small: Specifying this option tells GCC to generate code for the small code model. This means that the program and its symbols must be linked in the lower 2GB of the address space, pointers are 64 bits, and programs can be statically or dynamically linked. This is the default code model.

-mno-red-zone: Specifying this option tells GCC not to use a so-called *red zone* for x86-64 code. The red zone is mandated by the x86-64 ABI, and is a 128-byte area beyond the location of the stack pointer that will not be modified by signal or interrupt handlers and can therefore be used for temporary data without adjusting the stack pointer. Using the red zone is enabled by default.

AMD29K Options

The AMD 29000 processor is a RISC microprocessor descended from the Berkley RISC design, and includes a MMU as well as support for the AMD 29027 floating-point unit (FPU). Like conceptually similar processors such as the SPARC, the 29000 has a large register set split into local and global sets, and provides sophisticated mechanisms for protecting, manipulating, and managing registers and their contents. The AMD 29000 family of processors includes the 29000, 29005, 29030, 29035, 29040, and 29050 microprocessors.

GCC options available when compiling code for the AMD AM29000 family of processors are the following:

-m29000: Specifying this option causes GCC to generate code that only uses instructions available in the basic AMD 29000 instruction set. This is the default.

-m29050: Specifying this option causes GCC to generate code that takes advantage of specific instructions available on the AMD 29050 processor.

-mbw: Specifying this option causes GCC to generate code that assumes the system supports byte and half-word write operations. This is the default.

-mdw: Specifying this option causes GCC to generate code that assumes the processor's DW bit is set, which indicates that byte and half-word operations are directly supported by the hardware. This is the default.

-impure-text: Specifying this option in conjunction with the **-shared** option tells GCC not to pass the **-assert pure-text** option to the linker when linking a shared object.

-mkernel-registers: Specifying this option causes GCC to generate references to registers gr64 through gr95 instead of to registers gr96 through gr127 (the latter is the default). This option is often used when compiling kernel code that wants to reserve and use a set of global registers that are disjoint from the set used by user-mode code. Any register names passed as compilation options using GCC's **-f** option must therefore use the standard user-mode register names.

-mlarge: Specifying this option causes GCC to always use **calli** instructions, regardless of the size of the output file. You should use this option if you expect any single file to compile into more than 256K of code.

-mnbw: Specifying this option causes GCC to generate code that assumes the processor does not support byte and half-word operations. Specifying this option automatically sets the related **-mdw** option.

-mndw: Specifying this option causes GCC to generate code that assumes the processor's DW bit is not set, indicating that byte and half-word operations are not directly supported by the hardware. This option is automatically set if you specify the `-mnbw` option.

-mno-impure-text: Specifying this option tells GCC to pass the `-assert pure-text` option to the linker when linking a shared object.

-mno-multm: Specifying this option causes GCC not to generate `multm` or `multmu` instructions. This option is used when compiling for 29000-based embedded systems that do not have trap handlers for these instructions.

-mno-reuse-arg-reg: Specifying this option tells GCC to not to reuse incoming argument registers for copying out arguments.

-mno-soft-float: Specifying this option tells GCC to use hardware floating-point instructions for floating-point operations. This is the default.

-mno-stack-check: Specifying this option causes GCC not to insert a call to `_msp_check` after each stack adjustment.

-mno-storem-bug: Specifying this option causes GCC not to generate code that keeps `mtsrim`, `insn`, and `storem` instructions together. This option should be used when compiling for the 29050 processor, which can handle the separation of these instructions.

-mnnormal: Specifying this option causes GCC to use the normal memory model that generates call instructions only when calling functions in the same file, and generates `calli` instructions otherwise. This memory model works correctly if each file occupies less than 256K but allows the entire executable to be larger than 256K. This is the default.

-mreuse-arg-reg: Specifying this option tells GCC to use only incoming argument registers for copying out arguments. This helps detect functions that are called with fewer arguments than they were declared with.

-msmall: Specifying this option causes GCC to use a small memory model that assumes that all function addresses are either within a single 256K segment or at an absolute address of less than 256K. This allows code to use the `call` instruction instead of a `const`, `consth`, and `calli` sequence.

-msoft-float: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

-mstack-check: Specifying this option causes GCC to insert a call to `_msp_check()` after each stack adjustment. This option is often used when compiling kernel or general operating system code.

-mstorem-bug: Specifying this option causes GCC to generate code for AMD 29000 processors that cannot handle the separation of `mtsrm`, `insn`, and `storem` instructions. This option should be used on most 29000 processors, with the exception of the 29050.

-muser-registers: Specifying this option causes GCC to use the standard set of global registers (gr96 through gr127). This is the default.

ARC Options

The ARC processor is a highly customizable 32-bit RISC processor core that is often used in embedded systems.

GCC options available when compiling code for ARC processors are the following:

-EB: Specifying this option causes GCC to generate code that is compiled for big endian mode (where the most significant byte of a word has the lowest address—the word is stored big end first).

-EL: Specifying this option causes GCC to generate code that is compiled for little endian mode (where the most significant byte of a word has the highest significance—the word is stored little end first). This is the default.

-mcpu=CPU: Specifying this option causes GCC to generate code compiled for the specific ARC variant named CPU. The variants supported by GCC depend on the GCC configuration. Possible values are `arc`, `arc5`, `arc6`, `arc7`, `arc8`, and `base`. All ARC variants support `-mcpu=base`, which is the default.

-mdata=*data-section*: Specifying this option causes GCC to store data in the section whose name is specified as *data-section*. This command-line option can be overridden by using the `_attribute_` keyword to set the `section` attribute for a specific function, as explained in Chapter 4.

-mmangle-cpu: Specifying this option causes GCC to add the name of the CPU at the beginning of all public symbol names. Many multiple-processor systems use ARC CPU variants with different instruction and register set characteristics. Using this option prevents code compiled for one CPU from being linked with code compiled for another.

-mrodata=READONLY-data-section: Specifying this option causes GCC to store read-only data in the section whose name is specified as READONLY-*data-section*. This command-line option can be overridden by using the `_attribute_` keyword to set the section attribute for a specific function, as explained in Chapter 4.

-mtext=text-section: Specifying this option causes GCC to put functions in the section whose name is specified as *text-section*, rather than in the default text section. This command-line option can be overridden by using the `_attribute_` keyword to set the section attribute for a specific function, as explained in Chapter 4.

ARM Options

The Advanced RISC Machines (ARM) processor is a 32-bit processor family that is extremely popular in embedded, low-power systems. The ARM acronym originally stood for Acorn RISC Machine because the processor was originally designed by Acorn Computer Systems. Advanced RISC Machines, Ltd. was formed to market and develop the ARM processor family, related chips, and associated software, at which point the more modern acronym expansion was adopted.

NOTE *The ARM instruction set is a complete set of 32-bit instructions for the ARM architecture. The Thumb instruction set is an extension to the 32-bit ARM architecture that provides very high code density through a subset of the most commonly used 32-bit ARM instructions that have been compressed into 16-bit-wide operation codes. On execution, these 16-bit instructions are decoded to enable the same functions as their full 32-bit ARM instruction equivalents.*

The ARM Procedure Call Standard (APCS) is frequently referenced in ARM-oriented command-line options for GCC. APCS is a set of standards that define how registers are used on ARM processors, conventions for using the stack, mechanisms for passing and returning arguments across function calls, and the format and use of the stack.

GCC options available when compiling code for ARM processors are the following:

-mabort-on-noreturn: Specifying this option causes GCC to generate a call to the abort function at the end of each noreturn function. The call to abort is executed if the function tries to return.

-malignment-traps: Specifying this option causes GCC to generate code that will not trap if the MMU has alignment traps enabled; by replacing unaligned accesses with a sequence of byte accesses. This option is only relevant for ARM processors prior to the ARM 4, and is ignored on later processors because these have instructions to directly access half-word objects in memory.

ARM architectures prior to ARM 4 had no instructions to access half-word objects stored in memory. However, a feature of the ARM architecture allows a word load to be used when reading from memory even if the address is unaligned, because the processor core rotates the data as it is being loaded. Specifying this option tells GCC that such misaligned accesses will cause a MMU trap and that it should replace the misaligned access with a series of byte accesses. The compiler can still use word accesses to load half-word data if it knows that the address is aligned to a word boundary.

-mapcs: This option is a synonym for the **-mapcs-frame** option.

-mapcs-26: Specifying this option causes GCC to generate code for an ARM processor running with a 26-bit program counter. The generated code conforms to calling standards for the APCS 26-bit option. The **-mapcd-26** option replaces the **-m2** and **-m3** options provided in previous releases of GCC.

-mapcs-32: Specifying this option causes GCC to generate code for an ARM processor running with a 32-bit program counter. The generated code conforms to calling standards for the APCS 32-bit option. The **-mapcd-26** option replaces the **-m6** option provided in previous releases of GCC.

-mapcs-frame: Specifying this option causes GCC to generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Using this option in conjunction with the **-fomit-frame-pointer** option causes GCC not to generate stack frames for leaf functions (functions that do not call other functions). The default is not to generate stack frames.

-march=name: Using this option enables you to identify the ARM architecture used on the system for which you are compiling. Like the **-mcpu** option, GCC uses this name to determine the instructions that it can use when generating assembly code. This option can be used in conjunction with or instead of the **-mcpu=** option. Possible values for *name* are armv2, armv2a, armv3, armv3m, armv4, armv4t, armv5, armv5t, and armv5te. See the **-mcpu=name** and **-mtune=name** options for related information.

-mbig-endian: Specifying this option causes GCC to generate code for an ARM processor running in big endian mode.

-mbsd: Specifying this option causes GCC to emulate the native BSD-mode compiler. This is the default if **-ansi** is not specified. This option is only relevant when compiling code for RISC iX, which is Acorn's version of Unix that was supplied with some ARM-based systems such as the Acorn Archimedes R260.

-mcallee-super-interworking: Specifying this option causes GCC to insert an ARM instruction set header before executing any externally visible function. *Interworking mode* is a mode of operation in which ARM and Thumb instructions can interact. This header causes the processor to switch to Thumb mode before executing the rest of the function. This allows these functions to be called from noninterworking code.

-mcaller-super-interworking: Specifying this option enables calls via function pointers (including virtual functions) to execute correctly regardless of whether the target code has been compiled for interworking or not. This option causes a slight increase in the cost of executing a function pointer, but facilitates interworking in all circumstances.

-mcpu=name: Using this option enables you to specify the particular type of ARM processor used on the system that you are compiling for. GCC uses this name to determine the instructions that it can use when generating assembly code. Possible values for *name* are arm2, arm250, arm3, arm6, arm60, arm600, arm610, arm620, arm7, arm7m, arm7d, arm7dm, arm7di, arm7dmi, arm70, arm700, arm700i, arm710, arm710c, arm7100, arm7500, arm7500fe, arm7tdmi, arm8, strongarm, strongarm110, strongarm1100, arm8, arm810, arm9, arm9e, arm920, arm920t, arm940t, arm9tdmi, arm10tdmi, arm1020t, and xscale. See the **-march=name** and **-mtune=name** options for related information.

-mfpu=number | -mfpe=number: Using this option enables you to specify the version of floating-point emulation that is available on the system for which you are compiling. Possible values are 2 and 3, which internally identify different implementations. This option is provided for compatibility with earlier versions of GCC.

-mhard-float: Specifying this option causes GCC to generate output containing floating-point instructions. This is the default.

-mlittle-endian: Specifying this option causes GCC to generate code for an ARM processor running in little endian mode. This is the default.

-mlong-calls: Specifying this option causes GCC to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This option is necessary if the target function will lie outside of the 64MB addressing range of the offset-based version of the subroutine call instruction. Specifying this option does not affect calls to static functions, functions that have the short-call attribute, functions that are inside the scope of a `#pragma no_long_calls` directive, and functions whose definitions have already been compiled within the current compilation unit. Specifying this option does affect weak function definitions, functions with the long-call attribute or the section attribute, and functions that are within the scope of a `#pragma long_calls` directive.

-mno-alignment-traps: Specifying this option causes GCC to generate code that assumes that the MMU will not trap unaligned accesses. This produces better code for ARM processors prior to the ARM 4, where the target instruction set does not have half-word memory access operations. Unfortunately, you cannot use this option to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory. This is the default setting for ARM 4 and later processors.

-mno-apcs-frame: Specifying this option causes GCC not to generate stack frames that are compliant with the ARM Procedure Call Standard for all functions unless they are necessary for correct execution of the code. This option is the default.

-mno-long-calls: Specifying this option causes GCC to perform function calls in the standard fashion, rather than by the mechanism described in the `-mlong-calls` option. This is the default.

-mno-sched-prolog: Specifying this option causes GCC to suppress possible optimizations, preventing reordering instructions in the function prolog and preventing merging those instructions with the instructions in the function's body. Specifying this option means that all functions will start with a recognizable set of instructions from one of the set of default function prologues, which can then be used to identify the beginning of functions within an executable piece of code. Using this option typically results in larger executables than the default option, `-msched-prolog`.

- mno-short-load-bytes: This is a deprecated alias for the -mno-alignment-traps option.
- mno-short-load-words: This is a deprecated alias for the -malignment-traps option.
- mno-soft-float: Specifying this option tells GCC to use hardware floating-point instructions for floating-point operations. This is the default.
- mno-symrename: Specifying this option causes GCC not to run the assembler post-processor, symrename, after assembling code on a RISC iX system. This post-processor is normally run in order to modify standard symbols in assembly output so that resulting binaries can be successfully linked with the RISC iX C library. This option is only relevant for versions of GCC that are running directly on a RISC iX system: no such post-processor is provided by GCC when GCC is used as a cross-compiler.
- mno-thumb-interwork: Specifying this option causes GCC to generate code that does not support calls between the ARM and Thumb instruction sets. This option is the default. See the -mthumb-interwork option for related information.
- mno-tpcs-frame: Specifying this option causes GCC not to generate stack frames that are compliant with the Thumb Procedure Call Standard. This option is the default. See the -mtpcs-frame option for related information.
- mno-tpcs-leaf-frame: Specifying this option causes GCC not to generate stack frames that are compliant with the Thumb Procedure Call Standard. This option is the default. See the -mtpcs-leaf-frame option for related information.
- mnop-fun-dllimport: Specifying this option causes GCC to disable support for the `dllimport` attribute.
- mpic-register=*reg*: Using this option enables you to specify the register to be used for PIC addressing. The default register used for PIC addressing is register R10 unless stack checking is enabled, in which case register R9 is used.
- mpoke-function-name: Specifying this option causes GCC to write the name of each function into the text section immediately preceding each function prologue. Specifying this option means that all functions will be preceded by a recognizable label that can then be used to identify the beginning of functions within an executable piece of code.

-msched-prolog: Specifying this option causes GCC to optimize function prolog and body code sequences, merging operations whenever possible. This option is the default.

-mshort-load-bytes: This is a deprecated alias for the **-malignment-traps** option.

-mshort-load-words: This is a deprecated alias for the **-mno-alignment-traps** option.

-msingle-pic-base: Specifying this option causes GCC to treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The runtime system is responsible for initializing this register with an appropriate value before execution begins.

-msoft-float: Specifying this option causes GCC to generate output containing library calls for floating point. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

TIP *Specifying this option also changes the calling convention used in the output file. You must therefore compile all of the modules of your program with this option, including any libraries that you reference. You must also compile libgcc.a, the library that comes with GCC, with this option in order to be able to use this option.*

-mstructure-size-boundary=*n*: Using this option causes GCC to round the size of all structures and unions up to a multiple of the number of bits (*n*) specified with this option. Valid values are 8 and 32, and vary for different output formats: the default value is 8 for COFF output toolchains. Code compiled with one value will not necessarily work with code or libraries compiled with the other value. Specifying the larger number might produce faster, more efficient code, but may also increase the size of the program.

-mthumb: Specifying this option causes GCC to generate code for the 16-bit Thumb instruction set. The default is to use the 32-bit ARM instruction set.

-mthumb-interwork: Specifying this option causes GCC to generate code that supports calls between the ARM and Thumb instruction sets. If this option is not specified, the two instruction sets cannot be reliably used inside one program. Specifying this option causes GCC to generate slightly larger executables than the ARM-specific default option, `-mno-thumb-interwork`.

-mtpcs-frame: Specifying this option causes GCC to generate stack frames that are compliant with the Thumb Procedure Call Standard. Specifying this option only affects nonleaf functions (i.e., functions that call other functions). The default is `-mno-tpcs-frame`—in other words, not to generate compliant stack frames.

-mtpcs-leaf-frame: Specifying this option causes GCC to generate stack frames that are compliant with the Thumb Procedure Call Standard. Specifying this option only affects leaf functions (i.e., functions that do not call other functions). The default is `-mno-tpcs-leaf-frame`—in other words, not to generate compliant stack frames.

-mtune=*name*: Specifying this option causes GCC to tune the generated code as though the target ARM processor were of type *name*, but to still generate code that conforms to the instructions available for an ARM processor specified using the `-mcpu` option. Using these two options together can provide better performance on some ARM-based systems. See the `-march=name` and `-mcpu=name` options for related information.

-mwords-little-endian: Specifying this option causes GCC to generate code for a little endian word order but a big endian byte order (i.e., of the form “32107654”). This option only applies when generating code for big endian processors, and should only be used for compatibility with code for big endian ARM processors that was generated by versions of GCC prior to 2.8.

-mxopen: Specifying this option causes GCC to emulate the native X/Open-mode compiler. This is the default if `-ansi` is not specified. This option is only relevant when compiling code for RISC iX, which is Acorn’s version of Unix that was supplied with some ARM-based systems such as the Acorn Archimedes R260.

AVR Options

Atmel’s AVR processors are microcontrollers with a RISC core running single-cycle instructions. They provide a well-defined I/O structure that limits the need for external components, and are therefore frequently used in embedded systems.

GCC options available when compiling code for AVR processors are the following:

-mcall-prologues: Specifying this option causes GCC to expand function prologues and epilogues as calls to the appropriate subroutines, reducing code size.

-minit-stack=n: Specifying this option enables you to define the initial stack address, which may be a symbol or numeric value. The value `_stack` is the default.

-mmcu=MCU: Specifying this option enables you to specify the AVR instruction set or *MCU* type for which code should be generated. Possible values for the instruction set are the following:

avr1: The minimal AVR core. This value is not supported by the C compiler but only by the assembler. Associated *MCU* types are at90s1200, attiny10, attiny11, attiny12, attiny15, and attiny28.

avr2: The classic AVR core with up to 8K program memory space. This is the default. Associated *MCU* types are at90s2313, at90s2323, attiny22, at90s2333, at90s2343, at90s4414, at90s4433, at90s4434, at90s8515, at90c8534, and at90s8535.

avr3: The classic AVR core with up to 128K program memory space. Associated *MCU* types are atmega103, atmega603, at43usb320, and at76c711.

avr4: The enhanced AVR core with up to 8K program memory space. Associated *MCU* types are atmega8, atmega83, and atmega85.

avr5: The enhanced AVR core with up to 128K program memory space. Associated *MCU* types are atmega16, atmega161, atmega163, atmega32, atmega323, atmega64, atmega128, at43usb355, and at94k.

-mno-interrupts: Specifying this option causes GCC to generate code that is not compatible with hardware interrupts, reducing code size.

-mno-tablejump: Specifying this option causes GCC not to generate table-jump instruction, which may increase code size.

-msize: Specifying this option causes GCC to output instruction sizes to the assembler output file.

-mtiny-stack: Specifying this option tells GCC to generate code that only uses the low 8 bits of the stack pointer.

Clipper Options

The Clipper is a family of RISC processors that were primarily used in older Intergraph Unix workstations.

GCC options available when compiling code for Clipper processors are the following:

-mc300: Specifying this option causes GCC to generate code for a C300 Clipper processor. This is the default.

-mc400: Specifying this option causes GCC to generate code for a C400 Clipper processor. The generated code uses floating-point registers f8 through f15.

Convex Options

Convex Computer systems were minisupercomputers that were targeted for use by small to medium-sized businesses. Systems such as the C1, C2, and C3 were high-performance vector-processing systems that were substantially less expensive than competing systems from Cray Research. The later Exemplar systems were based on the Hewlett-Packard PA-RISC CPU series. Convex was acquired by HP in 1995.

GCC options available when compiling code for Convex systems are the following:

-margcnt: Specifying this option causes GCC to generate code that puts an argument count in the word preceding each argument list. This argument count word is compatible with regular CC provided by ConvexOS, and may be needed by some programs.

-mc1: Specifying this option causes GCC to generate code targeted for Convex C1 systems, but which will run on any Convex machine. This option defines the preprocessor symbol `_convex_c1_`.

-mc2: Specifying this option causes GCC to generate code targeted for Convex C2 systems, but which should also run on Convex C3 machines, though scheduling and other optimizations are chosen for maximum performance on C2 systems. This option defines the preprocessor symbol `_convex_c2_`.

-mc32: Specifying this option causes GCC to generate code targeted for Convex C32xx systems, using scheduling and other optimizations chosen for maximum performance on C32xx systems. This option defines the preprocessor symbol `_convex_c32_`.

-mc34: Specifying this option causes GCC to generate code targeted for Convex C34xx systems, using scheduling and other optimizations chosen for maximum performance on C34xx systems. This option defines the preprocessor symbol `_convex_c34_`.

-mc38: Specifying this option causes GCC to generate code targeted for Convex C38xx systems, using scheduling and other optimizations that are chosen for maximum performance on C38xx systems. This option defines the preprocessor symbol `_convex_c38_`.

-mlong32: Specifying this option causes GCC to define type long as 32 bits, the same as type int. This is the default.

-mlong64: Specifying this option causes GCC to define type long as 64 bits, the same as type long long. This option is essentially useless because there is no support for this convention in the GCC libraries.

-mnoargcount: Specifying this option causes GCC to omit the argument count word. This is the default. See the `-margcount` option for related information.

-mvolatile-cache: Specifying this option causes GCC to generate code in which volatile references are cached. This is the default.

-mvolatile-nocache: Specifying this option causes GCC to generate code in which volatile references bypass the data cache, going directly to memory. This option is only needed for multiprocessor code that does not use standard synchronization instructions. Making nonvolatile references to volatile locations will not necessarily work.

CRIS Options

Axis Solutions' Code Reduced Instruction Set (CRIS) processors are frequently used in network-oriented embedded applications.

GCC options available when compiling code for CRIS systems are the following:

-m16-bit: Specifying this option tells GCC to align the stack frame, writable data, and constants to all be 16-bit aligned. The default is 32-bit alignment.

-m32-bit: Specifying this option tells GCC to align the stack frame, writable data, and constants to all be 32-bit aligned. This is the default.

-m8-bit: Specifying this option tells GCC to align the stack frame, writable data, and constants to all be 8-bit aligned. The default is 32-bit alignment.

-maout: This deprecated option is a NOOP that is only recognized with the cris-axis-aout GCC build target.

-march=architecture-type | -mcpu=architecture-type: Specifying this option causes GCC to generate code for the specified architecture. Possible values for *architecture-type* are v3 (for ETRAX 4), v8 (for ETRAX 100), and v10 (for ETRAX 100 LX). The default is v0 except for the cris-axis-linux-gnu GCC build target, where the default is v10.

-mcc-init: Specifying this option tells GCC not to use condition-code results from previous instructions, but to always generate compare and test instructions before using condition codes.

-mconst-align: Specifying this option tells GCC to align constants for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

-mdata-align: Specifying this option tells GCC to align individual data for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

-melf: This deprecated option is a NOOP that is only recognized with the cris-axis-elf and cris-axis-linux-gnu GCC build targets.

-mlinux: Specifying this option tells GCC to select a GNU/Linux-like multilib, using include files and instruction set for **-march=v8**. This option is only recognized with the cris-axis-aout GCC build target.

-mlinux-stacksize=n: Specifying this option tells GCC to include instructions in the program so that the kernel loader sets the stack size of the program to *n* bytes. This option is only available on the cris-axis-aout GCC build target.

-metrax4: This option is a synonym for the **-march=v3** option.

-metrax100: This option is a synonym for the **-march=v8** option.

-mgotplt: When used with the **-fPIC** or **-fPIC** options, specifying this option tells GCC to generate instruction sequences that load addresses for functions from the PLT part of the GOT rather than by making calls to the PLT. This is the default.

-mlinux: This deprecated option is a NOOP that is only recognized with the cris-axis-linux-gnu GCC build target.

-max-stack-frame=n: Specifying this option causes GCC to display a warning when the stack frame of a function exceeds *n* bytes.

-mno-const-align: Specifying this option tells GCC not to align the constants for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

-mno-data-align: Specifying this option tells GCC not to align individual data for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

-mno-gotplt: When used with the `-fpic` or `-fPIC` options, specifying this option tells GCC to make calls to the PLT rather than loading addresses for functions from the PLT part of the GOT.

-mno-prologue-epilogue: Specifying this option tells GCC to omit the normal function prologue and epilogue that sets up the stack frame, and not to generate return instructions or return sequences in the code. This option is designed to simplify visual inspection of the assembler, as it may generate code that stops on certain registers. No warnings or errors are generated when call-saved registers must be saved, or when storage for local variable needs to be allocated.

-mno-side-effects: Specifying this option tells GCC not to emit instructions with side effects when using addressing modes other than post-increment.

-mno-stack-align: Specifying this option tells GCC not to align the stack frame for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

-mpdebug: Specifying this option tells GCC to enable verbose, CRIS-specific debugging information in the assembly code. This option also turns off the `#NO_APP` formatted-code indicator at the beginning of the assembly file.

-mprologue-epilogue: Specifying this option tells GCC to include the normal function prologue and epilogue that set up the stack frame. This is the default.

-mstack-align: Specifying this option tells GCC to align the stack frame for the maximum single data access size for the specified CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by this option.

-mtune= *architecture-type*: Specifying this option causes GCC to tune the generated code for the specified *architecture-type*, except for the ABI and the set of available instructions. The choices for *architecture-type* are the same as those for the **-march= *architecture-type*** option.

-sim: Specifying this option tells GCC to link with input-output functions from a simulator library. Code, initialized data, and zero-initialized data are allocated consecutively. This option is only recognized for the `cris-axis-aout` and `cris-axis-elf` GCC build targets.

-sim2: Specifying this option tells GCC to link with input-output functions from a simulator library, and to pass linker options to locate initialized data at `0x40000000` and zero-initialized data at `0x80000000`. Code, initialized data, and zero-initialized data are allocated consecutively. This option is only recognized for the `cris-axis-aout` and `cris-axis-elf` GCC build targets.

D30V Options

Mitsubishi's D30V processor is a RISC processor with integrated hardware support for a real-time MPEG-2 decoder. The D30V is frequently used in embedded multimedia applications.

GCC options available when compiling code for D30V-based systems are the following:

-masm-optimize: Specifying this option tells GCC to pass the `-O` option to the assembler during optimization. The assembler uses the `-O` option to automatically parallelize adjacent short instructions whenever possible.

-mbranch-cost=n: Specifying this option tells GCC to increase the internal cost of a branch to the value specified as *n*. Higher costs mean that the compiler will generate more instructions in order to avoid doing a branch. Avoiding branches is a common performance optimization for RISC processors. The default is 2.

-mcond-exec=n: Specifying this option enables you to specify the maximum number of conditionally executed instructions that replace a branch as the value *n*. The default is 4.

-mextmem | -mextmemory: Specifying this options tells GCC to link the text, data, bss, strings, rodata, rodata1, and data1 sections into external memory, which starts at location `0x80000000`.

-monchip: Specifying this option tells GCC to link the text section into on-chip text memory, which starts at location 0x0. This option also tells GCC to link data, bss, strings, rodata, rodata1, and data1 sections into on-chip data memory, which starts at location 0x20000000.

-mno-asmp-optimize: Specifying this option tells GCC to not pass the -O option to the assembler, thereby suppressing optimization.

H8/300 Options

H8 is a family of 8-bit microprocessors featuring an H8/300 CPU core and a variety of on-chip supporting modules that provide a variety of system functions. Often used as microcontrollers in embedded environments, H8/300-based microprocessors are available from Hitachi, SuperH, and now Renasas.

GCC options available when compiling code for H8/300-based systems are the following:

-malign-300: Specifying this option when compiling for the H8/300H and H8/S processors tells GCC to use the same alignment rules as for the H8/300. The default for the H8/300H and H8/S is to align longs and floats on 4-byte boundaries. Specifying the -malign-300 option causes longs and floats to be aligned on 2-byte boundaries. This option has no effect on the H8/300.

-mh: Specifying this option tells GCC to generate code for the H8/300H processor.

-mint32: Specifying this option tells GCC to make int data 32-bits long by default, rather than 16-bits long.

-mrelax: Specifying this option tells GCC to shorten some address references at link time, when possible, by passing the -relax option to ld, the GNU linker/loader.

-ms: Specifying this option tells GCC to generate code for the H8/S processor.

-ms2600: Specifying this option tells GCC to generate code for the H8/S2600 processor. The -s option must also be specified when using this option.

HP/PA (PA/RISC) Options

HP/PA stands for Hewlett Packard Precision Architecture, the original name for what are now commonly referred to as Precision Architecture, Reduced Instruction Set Computing (PA-RISC) systems. PA-RISC is a microprocessor architecture developed by Hewlett-Packard's Systems and VLSI Technology Operation, and owes some of its design to the RISC technologies introduced in Apollo's DN10000 RISC systems. Apollo was consumed by HP in the late 1980s, a sad time for us all. PA-RISC CPUs are used in many later HP workstations.

GCC options available when compiling code for PA-RISC systems are the following:

-march=architecture-type: Specifying this option tells GCC to generate code for the specified architecture. The choices for *architecture-type* are 1.0 for PA 1.0, 1.1 for PA 1.1, and 2.0 for PA 2.0 processors. The file /usr/lib/sched.models on an HP-UX system identifies the proper architecture option for specific machines. Code compiled for lower numbered architectures will run on higher numbered architectures, but not the other way around.

-mbig-switch: Specifying this option tells GCC to generate code suitable for big switch tables. You should only use this option if the assembler or linker complains about branches being out of range within a switch table.

-mdisable-fpregs: Specifying this option tells GCC to prevent floating-point registers from being used. This option is used when compiling kernels that perform lazy context switching of floating-point registers. GCC will abort compilation if you use this option and attempt to perform floating-point operations in the application that you are compiling.

-mdisable-indexing: Specifying this option tells GCC not to use indexing address modes. You should only use this option if you are running GCC on a PA-RISC system running Mach, and what are the chances of that?

-mfast-indirect-calls: Specifying this option tells GCC to generate code that assumes calls never cross space boundaries. This enables GCC to generate code that performs faster indirect calls. This option will not work in nested functions or when shared libraries are being used.

-mgas: Specifying this option enables GCC to use assembler directives that are only understood by the GNU assembler. This option should therefore not be used if you are using the HP assembler with GCC.

-mjump-in-delay: Specifying this option tells GCC to fill the delay slots of function calls with unconditional jump instructions by modifying the return pointer for the function call to be the target of the conditional jump.

-mlinkr-opt: Specifying this option tells GCC to enable the optimization pass in the HP-UX linker. Optimization makes symbolic debugging impossible, but will provide improved performance in most cases. If you are using GCC on HP-UX 8 or 9 systems, using the HP-UX linker may display erroneous error messages when linking some programs.

-mlong-load-store: Specifying this option tells GCC to generate the 3-instruction load and store sequences that are sometimes required by the HP-UX 10 linker. This option should be unnecessary if you are using the standard GNU linker/loader. This option is the same as the `+k` option provided by HP compilers.

-mno-space-reg: Specifying this option tells GCC to generate code that assumes the target has no space registers. This option enables GCC to generate faster indirect calls and use unscaled index address modes. Typically, this option should only be used on PA-RISC 1.0 systems or when compiling a kernel.

-mpa-risc-1-0: A deprecated synonym for the `-march=1.0` option.

-mpa-risc-1-1: A deprecated synonym for the `-march=1.1` option.

-mpa-risc-2-0: A deprecated synonym for the `-march=2.0` option.

-mportable-runtime: Specifying this option tells GCC to use the portable calling conventions proposed by HP for ELF systems.

-mschedule=*CPU-type*: Specifying this option tells GCC to schedule code according to the constraints for the machine type *CPU-type*. Possible choices for *CPU-type* are 700, 7100, 7100LC, 7200, and 8000. The file `/usr/lib/sched.models` on an HP-UX system shows the proper scheduling option for your machine.

-msoft-float: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

TIP Specifying the `-msoft-float` option changes the calling conventions in the output file. All of the modules of a program must be compiled using this option in order to be successfully linked. You will also need to compile `libgcc.a`, the library that comes with GCC, with `-msoft-float` in order to use this option.

i386 and AMD x86-64 Options

The Intel x86 family of 32-bit processors, commonly referred to as i386 processors, is the best known and most widely used type of processor found in modern desktop computers. It is easy to argue their merits against other popular processors such as Apple's PPC G4 and G5 processors, but it is impossible to argue with their ubiquity. The AMD64 processor is AMD's family of compatible 64-bit processors.

The options in this section can be used when compiling any i386 or x86-64 code. For the discussion of options that can only be used on AMD x86-64 systems, see the section of this appendix entitled "AMD x86-64 Options."

GCC options available when compiling code for all i386 and 64-bit AMD systems are the following:

-m128bit-long-double: Specifying this option tells GCC to use 128 bits to store long doubles, rather than using 12 bytes (96 bits) as specified in the i386 application binary interface. Pentium and newer processors provide higher performance when long doubles are aligned to 8- or 16-byte boundaries, though this is impossible to reach when using 12-byte long doubles for array access. If you specify this option, any structures and arrays containing long doubles will change size, and function calling conventions for functions taking long double arguments will also be modified.

-m386: This option is a deprecated synonym for the `-mcpu=i386` option.

-m3dnow: Specifying this option enables the use of built-in functions that allow direct access to the 3Dnow extensions of the instruction set.

-m486: This option is a deprecated synonym for the `-mcpu=i486` option.

-m96bit-long-double: Specifying this option tells GCC to set the size of long doubles to 96 bits as required by the i386 application binary interface. This is the default.

-maccumulate-outgoing-args: Specifying this option tells GCC to compute the maximum amount of space required for outgoing arguments in the function prologue. This is faster on most modern CPUs because of reduced dependencies, improved scheduling, and reduced stack usage when the preferred stack boundary is not equal to 2. The downside is a notable increase in code size. Specifying this option implies the `-mno-push-args` option.

-malign-double: Specifying this option tells GCC to align double, long double, and long long variables on a two word boundary. This produces code that runs somewhat faster on Pentium or better systems, at the expense of using more memory. Specifying this option aligns structures in a way that does not conform to the published application binary interface specifications for the 386.

-march=CPU-type: Specifying this option tells GCC to generate instructions for the specified machine type *CPU-type*. Possible values for *CPU-type* are i386, i486, i586 (equivalent to pentium), i686 (equivalent to pentiumpro), pentium, pentium-mmx, pentiumpro, pentium2, pentium3, pentium4, k6, k6-2, k6-3, athlon, athlon-tbird, athlon-4, athlon-xp, and athlon-mp. Specifying the `-march=CPU-type` option implies the `-mcpu=CPU-type` option.

-masm=dialect: Specifying this option tells GCC to Output `asm` instructions using the selected *dialect*, which can be either intel or att. The att dialect is the default.

-mcpu=CPU-type: Specifying this option tells GCC to tune the code that it generates for the specified *CPU-type*. The choices for *CPU-type* are the same as for the `-march=CPU-type` option. Specifying a particular *CPU-type* causes GCC to schedule generated code appropriately for that particular chip.

-mfpmath=unit: Specifying this option tells GCC to generate floating-point arithmetic for the floating-point unit *unit*. Valid choices for *unit* are the following:

387: Use the standard 387 floating-point coprocessor present in most i386 chips and emulated otherwise. Code compiled with this option will run almost everywhere. The temporary results are computed in 80-bit precision instead of precision specified by the type, resulting in slightly different results compared to most of other chips. See the standard GCC option `-ffloat-store` for more information. This is the default if the `-mfpmath` option is not specified.

sse: Use scalar floating-point instructions present in the Streaming SIMD Extensions (SSE) instruction set. This instruction set is supported by Intel Pentium III and newer chips, and in the Athlon 4, Athlon XP, and Athlon MP chips from AMD. Earlier versions of the SSE instruction set only supported single precision arithmetic, which means that double and extended precision math is still done using the standard 387 instruction set. The versions of the SSE instruction set provided in Pentium 4 and AMD x86-64 chips provides direct support for double precision arithmetic. If you are compiling for chips other than these, you should use the `-march=CPU-type` and `-mfpmath=sse` options to enable SSE extensions and to use this option effectively. The `-mfpmath=sse` option is the default for GCC compiled for the x86-64 target.

sse,387: Attempt to utilize both instruction sets at once. This effectively doubles the number of available registers, as well as the amount of execution resources on chips with separate execution units for 387 and SSE. Using this option may cause problems because the GCC register allocator does not always model separate functional units well.

-mieee-fp: Specifying this option tells GCC to use IEEE floating-point comparisons, which correctly handle the case where the result of a comparison is unordered.

-minline-all-stringops: Specifying this option tells GCC to inline all string operations. By default, GCC only inlines string operations when the destination is known to be aligned to at least a 4-byte boundary. This enables more inlining, which increases code size, but may also improve the performance of code that depends on fast `memcpy()`, `strlen()`, and `memset()` for short lengths.

-mmmx: Specifying this option enables the use of built-in functions that allow direct access to the MMX extensions of the instruction set.

-mno-3dnow: Specifying this option disables the use of built-in functions that allow direct access to the 3Dnow extensions of the instruction set.

-mno-align-double: Specifying this option tells GCC not to align `double`, `long double`, and `long long` variables on a two-word boundary, using a one-word boundary instead. This reduces memory consumption but may result in slightly slower code. Specifying this option aligns structures containing these data types in a way that conforms to the published application binary interface specifications for the 386.

-mno-align-stringops: Specifying this option tells GCC not to align the destination of inlined string operations. This switch reduces code size and improves performance when the destination is already aligned.

-mno-fancy-math-387: Specifying this option tells GCC to avoid generating the sin, cos, and sqrt instructions for the 387 because these instructions are not provided by all 387 emulators. This option has no effect unless you also specify the -funsafe-math-optimizations option. This option is overridden when -march indicates that the target CPU will always have an FPU and so the instruction will not need emulation. This option is the default for GCC on FreeBSD, OpenBSD, and NetBSD systems.

-mno-fp-ret-in-387: Specifying this option tells GCC not to use the FPU registers for function return values, returning them in ordinary CPU registers instead. The usual calling convention on i386 systems returns float and double function values in an FPU register, even if there is no FPU. This assumes that the operating system provides emulation support for an FPU, which may not always be correct.

-mno-ieee-fp: Specifying this option tells GCC not to use IEEE floating-point comparisons.

-mno-mmx: Specifying this option disables the use of built-in functions that allow direct access to the MMX extensions of the instruction set.

-mno-push-args: Specifying this option tells GCC to use standard SUB/MOV operations to store outgoing parameters, rather than the potentially smaller PUSH operation. In some cases, using SUB/MOV rather than PUSH may improve performance because of improved scheduling and reduced dependencies.

-mno-sse: Specifying this option disables the use of built-in functions that allow direct access to the SSE extensions of the instruction set.

-mno-sse2: Specifying this option disables the use of built-in functions that allow direct access to the SSE2 extensions of the instruction set.

-mno-svr3-shlib: Specifying this option tells GCC to place uninitialized local variables in the DATA segment. This option is only meaningful on System V Release 3 (SVR3) systems.

-fomit-leaf-frame-pointer: Specifying this option tells GCC not to keep the frame pointer in a register for leaf functions. This avoids generating the instructions to save, set up, and restore frame pointers and makes an extra register available in leaf functions. The option -fomit-frame-pointer removes the frame pointer for all functions, which might make debugging harder.

-mpentium: This option is a deprecated synonym for the `-mcpu=i586` and `-mcpu=pentium` options.

-mpentiumpro: This option is a deprecated synonym for the `-mcpu=i686` and `-mcpu=pentiumpro` options.

-mpreferred-stack-boundary=*num*: Specifying this option tells GCC to attempt to keep the stack boundary aligned to a $2^{**\textit{num}}$ byte boundary. This extra alignment consumes extra stack space and generally increases code size. Code that is sensitive to stack space usage, such as embedded systems and operating system kernels, may want to reduce the preferred alignment to `-mpreferred-stack-boundary=2`. If the `-mpreferred-stack-boundary` option is not specified, the default is 4 (16 bytes or 128 bits), except when optimizing for code size (using the `-Os` option), in which case the default is the minimum correct alignment (4 bytes for x86, and 8 bytes for x86-64).

On Pentium and PentiumPro systems, `double` and `long double` values should be aligned to an 8-byte boundary (see the `-falign-double` option), or they will incur significant runtime performance penalties. On Pentium III systems, the SSE data type `_m128` incurs similar penalties if it is not 16-byte aligned.

NOTE *To ensure proper alignment of the values on the stack, the stack boundary must be as aligned as required by any value stored on the stack. Therefore, every function and library must be generated such that it keeps the stack aligned. Calling a function compiled with a higher preferred stack boundary from a function compiled with a lower preferred stack boundary will probably misalign the stack. The GNU folks recommend that libraries that use callbacks always use the default setting.*

-mpush-args: Specifying this option tells GCC to use the `PUSH` operation to store outgoing parameters. This method is shorter and usually at least as fast as using `SUB/MOV` operations. This is the default.

-mregparm=*num*: Specifying this option controls the number of registers used to pass integer arguments. By default, no registers are used to pass arguments, and at most three registers can be used. You can also control this behavior for a specific function by using the function attribute `regparm`.

NOTE If you use this switch to specify a nonzero number of registers, you must build all modules with the same value, including any libraries that the program uses. This includes system libraries and startup modules.

-mrtd: Specifying this option tells GCC to use a function-calling convention where functions that take a fixed number of arguments return with the `ret num` instruction, which pops their arguments during the return. This saves one instruction in the caller because there is no need to pop the arguments there. This calling convention is incompatible with the one normally used on Unix, and therefore cannot be used if you need to call libraries that have been compiled with generic Unix compilers. When using this option, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf()`); if you do not, incorrect code will be generated for calls to those functions. You must also be careful to never call functions with extra arguments, which would result in seriously incorrect code. This option takes its name from the 680x0 rtd instruction.

TIP To optimize heavily used functions, you can specify that an individual function is called with this calling sequence with the function attribute `stdcall`. You can also override the `-mrtd` option for specific functions by using the function attribute `cdecl`.

-msoft-float: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

NOTE On machines where a function returns floating-point results in the 80387 register stack, some floating-point opcodes may be emitted even if `-msoft-float` is used.

-msse: Specifying this option enables the use of built-in functions that allow direct access to the SSE extensions of the instruction set.

-msse2: Specifying this option enables the use of built-in functions that allow direct access to the SSE2 extensions of the instruction set.

-msvr3-shlib: Specifying this option tells GCC to place uninitialized local variables in the BSS segment. This option is only meaningful on System V Release 3 (SVR3) systems.

-mthreads: Specifying this option tells GCC to support thread-safe exception handling on Mingw32 platforms. Code that relies on thread-safe exception handling must compile and link all code with this option. When compiling, specifying the **-mthreads** option sets the **-D_MT** symbol. When linking, specifying this option links in a special thread helper library (equivalent to the **-lmingwthrd** option) that cleans up per-thread exception-handling data.

IA-64 Options

The options in this section can only be used when compiling code targeted for 64-bit Intel processors (the GCC ia-64 build target). For a discussion of options that can be used on i386 systems, see the section of this appendix entitled “i386 and AMD x86-64 Options.” For a discussion of options that can only be used on AMD-64 (64-bit AMD) systems, see the section of this appendix entitled “AMD x86-64 Options.”

GCC options available when compiling code for 64-bit Intel systems are the following:

-mauto-pic: Specifying this option tells GCC to generate position-independent code (i.e., code that is self-relocatable). Specifying this option implies the **-mconstant-gp** option. This option is useful when compiling firmware code.

-mb-step: Specifying this option tells GCC to generate code that works around Itanium B step errata.

-mbig-endian: Specifying this option tells GCC to generate code for a big endian target. This is the default for HP-UX systems using the IA-64 processor.

-mconstant-gp: Specifying this option tells GCC to generate code that uses a single constant global pointer value. This is useful when compiling kernel code because all pointers are offset from a single global.

-mdwarf2-asm: Specifying this option tells GCC to generate assembler code for DWARF2 line number debugging info. This information may be useful when not using the GNU assembler.

-mfixed-range=register-range: Specifying this option tells GCC to generate code that treats the specified range of registers as fixed registers. A register range is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma. A fixed register is one that the register allocator cannot use. This option is useful when compiling kernel code.

-mgnu-as: Specifying this option tells GCC to generate code for the GNU assembler. This is the default.

-mgnu-ld: Specifying this option tells GCC to generate code for the GNU linker. This is the default.

-minline-divide-max-throughput: Specifying this option tells GCC to generate code for inline divides using the maximum throughput algorithm.

-minline-divide-min-latency: Specifying this option tells GCC to generate code for inline divides using the minimum latency algorithm.

-mlittle-endian: Specifying this option tells GCC to generate code for a little endian target. This is the default for AIX5 and Linux systems using the IA-64 processor.

-mno-dwarf2-asm: Specifying this option tells GCC not to generate assembler code for DWARF2 line number debugging.

-mno-gnu-as: Specifying this option tells GCC not to generate code for the GNU assembler, but to assume that assembly will be done using a system assembler.

-mno-gnu-ld: Specifying this option tells GCC not to generate code for the GNU linker, but to assume linking/loading will be done using a system linker.

-mno-pic: Specifying this option tells GCC to generate code that does not use a global pointer register. This results in code that is not position independent and violates the IA-64 ABI.

-mno-register-names: Specifying this option tells GCC not to generate in, loc, and out register names for the stacked registers.

-mno-volatile-asm-stop: Specifying this option tells GCC not to generate a stop bit immediately before and after volatile `asm` statements.

-mregister-names: Specifying this option tells GCC to generate in, loc, and out register names for the stacked registers. This may make assembler output more readable.

`-mno-sdata`: Specifying this option tells GCC to disable optimizations that use the small data section. This option may be useful for working around optimizer bugs.

`-msdata`: Specifying this option tells GCC to enable optimizations that use the small data section. Though this option provides performance improvements, problems have been reported when using these optimizations.

`-mvolatile-asm-stop`: Specifying this option tells GCC to generate a stop bit immediately before and after volatile `asm` statements.

Intel 960 Options

Intel's i960 family consists of high-performance, 32-bit embedded RISC processors supported by an outstanding selection of development tools (such as the one that you're reading about). Intel's i960 processors are often used in high-performance, embedded networking and imaging scenarios.

GCC options available when compiling code for Intel 960 systems are the following:

`-masm-compat` | `-mintel-asm`: Specifying either of these options tells GCC to enable compatibility with the iC960 assembler.

`-mcode-align`: Specifying this option tells GCC to align code to 8-byte boundaries in order to support faster fetching. This option is the default for C-series processors (as specified using the `-mcpu=CPU-type` option).

`-mcomplex-addr`: Specifying this option tells GCC to use a complex addressing mode to provide performance improvements. Complex addressing modes may not be worthwhile on the K-series processors, but they definitely are on the C-series processors. This option is the default for all processors (as specified using the `-mcpu=CPU-type` option) except for the CB and CC processors.

`-mcpu=CPU-type`: Specifying this option tells GCC to use the defaults for the machine type `CPU-type`, affecting instruction scheduling, floating-point support, and addressing modes. Possible values for `CPU-type` are `ka`, `kb`, `mc`, `ca`, `cf`, `sa`, and `sb`, which are different generations and versions of the i960 processor. The default is `kb`.

`-mic-compat`: Specifying this option tells GCC to enable compatibility with both the iC960 version 2.0 or version 3.0 C compilers from Intel.

`-mic2.0-compat`: Specifying this option tells GCC to enable compatibility with the iC960 version 2.0 C compiler from Intel.

-mic3.0-compat: Specifying this option tells GCC to enable compatibility with the iC960 version 3.0 C compiler from Intel.

-mleaf-procedures: Specifying this option tells GCC to attempt to alter leaf procedures to be callable with the `bal` instruction as well as `call`. This will result in more efficient code for explicit calls when the `bal` instruction can be substituted by the assembler or linker, but less efficient code in other cases, such as calls via function pointers or when using a linker that does not support this optimization.

-mlong-double-64: Specifying this option tells GCC to implement the `long double` type as 64-bit floating-point numbers. If you do not specify this option, `long doubles` are implemented by 80-bit floating-point numbers. This option is present because there is currently no 128-bit `long double` support. This option should only be used when using the `-msoft-float` option, e.g., for soft-float targets.

-mno-code-align: Specifying this option tells GCC not to align code to 8-byte boundaries for faster fetching (or do not bother). This option is the default for all non-C-series implementations (as specified using the `-mcpu-CPU-type` option)

-mno-complex-addr: Specifying this option tells GCC not to assume that the use of a complex addressing mode is a win on this implementation of the i960. Complex addressing modes may not be worthwhile on the K-series processors. This option is the default for the CB and CC processors (as specified using the `-mcpu-CPU-type` option).

-mno-leaf-procedures: Specifying this option tells GCC to always call leaf procedures with the `call` instruction.

-mno-strict-align: Specifying this option tells GCC to permit unaligned accesses.

-mno-tail-call: Specifying this option tells GCC not to make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. This is the default.

-mnumerics: Specifying this option tells GCC that the processor supports floating-point instructions.

-mold-align: Specifying this option tells GCC to enable structure-alignment compatibility with Intel's gcc release version 1.3 (based on gcc 1.37). It would be really sad if this option was still getting much use. This option implies the `-mstrict-align` option.

-msoft-float: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available.

When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

-mstrict-align: Specifying this option tells GCC not to permit unaligned accesses.

-mtail-call: Specifying this option tells GCC to make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete.

M32R Options

M32R is a family of 32-bit RISC microprocessors designed for embedded systems. M32Rs include an on-chip multiprocessor feature and can thus be used in both uniprocessor and symmetric multiprocessor (SMP) embedded designs.

GCC options available when compiling code for M32R systems are the following:

-G num: Specifying this option tells GCC to put global and static objects less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss sections. The default value of *num* is 8. The **-msdata** option must be set to either *sdata* or *use* for this option to have any effect.

NOTE All modules should be compiled with the same **-G num** value. Compiling with different values of *num* may or may not work. If it does not, the linker will display an error message and exit.

-m32r: Specifying this option tells GCC to generate code for the generic M32R. This is the default.

-m32rx: Specifying this option tells GCC to generate code for the M32R/X processor.

TIP *The addressability of a particular object can be set with the model attribute.*

-mcode-model=large: Specifying this option tells GCC to assume that objects may be anywhere in the 32-bit address space (GCC generates `seth/add3` instructions to load their addresses), and to assume that subroutines may not be reachable with the `b1` instruction (GCC generates the much slower `seth/add3/j1` instruction sequence).

-mcode-model=medium: Specifying this option tells GCC to assume that objects may be anywhere in the 32-bit address space (GCC generates `seth/add3` instructions to load their addresses), and to assume that all subroutines are reachable with the `b1` instruction.

-mcode-model=small: Specifying this option tells GCC to assume that all objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and to assume that all subroutines are reachable with the `b1` instruction. This is the default.

-msdata=None: Specifying this option tells GCC to disable the use of the small data area. The small data area consists of the sections `sdata` and `sbss`. Variables will be put into one of the `data`, `bss`, or `rodata` sections (unless the `section` attribute has been specified). This is the default.

TIP *Objects may be explicitly put in the small data area with the section attribute using one of these sections.*

-msdata=sdata: Specifying this option tells GCC to put small global and static data in the small data area, but not generate special code to reference them.

-msdata=use: Specifying this option tells GCC to put small global and static data in the small data area, and to generate special instructions to reference them.

M680x0 Options

The options in this section are used when employing GCC to compile applications for use on systems that rely on the Motorola 68000 family of processors.

These are the 68000, 68010, 68020, 68030, 68040, and 68060 processors. These legendary processors were used in almost all early computer workstations before the advent of RISC processors, and are still frequently used as microcontrollers.

GCC options available when compiling code for M680x0 systems are the following:

-m5200: Specifying this option tells GCC to generate code for a 520X “Coldfire” family CPU. This is the default when the compiler is configured for 520X-based systems. You should use this option when compiling code for microcontrollers with a 5200 core, including the MCF5202, MCF5203, MCF5204, and MCF5202 processors. The **-m5200** option implies the **-mnobitfield** option.

-m68000 | -mc68000: Specifying this option tells GCC to generate code for a 68000 processor. This is the default when the compiler is configured for 68000-based systems. You should use this option when compiling for microcontrollers with a 68000 or EC000 core, including the 68008, 68302, 68306, 68307, 68322, 68328, and 68356 processors. The **-m68000** option implies the **-mnobitfield** option.

-m68020 | -mc68020: Specifying this option tells GCC to generate code for a 68020 processor. This is the default when the compiler is configured for 68020-based systems. Specifying this option also sets the **-mbitfield** option.

-m68020-40: Specifying this option tells GCC to generate code for a 68040 processor, without using any instructions introduced since the 68020. This results in code that can run relatively efficiently on 68020/68881, 68030, or 68040 systems. The generated code uses the 68881 instructions that are emulated on the 68040 processor.

-m68020-60: Specifying this option tells GCC to generate code for a 68060 processor, without using any instructions introduced since the 68020 processor. This results in code that can run relatively efficiently on 68020/68881, 68030, 68040, or 68060 systems. The generated code uses the 68881 instructions that are emulated on the 68060 processor.

-m68030: Specifying this option tells GCC to generate code for a 68030 processor. This is the default when the compiler is configured for 68030-based systems.

-m68040: Specifying this option tells GCC to generate code for a 68040 processor. This is the default when the compiler is configured for 68040-based systems. Specifying this option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040 processor. You should use this option if your 68040 system does not have code to emulate those instructions.

-m68060: Specifying this option tells GCC to generate code for a 68060 processor. This is the default when the compiler is configured for 68060-based systems. This option inhibits the use of 68020 and 68881/68882 instructions that have to be emulated by software on the 68060 processor. You should use this option if your 68060 system does not have code to emulate those instructions.

-m68881: Specifying this option tells GCC to generate code containing 68881 instructions for floating point. This is the default for most 68020 systems unless the `--nfp` option was specified when GCC was configured.

-malign-int: Specifying this option tells GCC to align `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 32-bit boundary. Aligning variables on 32-bit boundaries produces code that runs somewhat faster on processors with 32-bit buses at the expense of more memory. Specifying this option aligns structures containing these data types differently than most published application binary interface specifications for the m68k.

-mbitfield: Specifying this option tells GCC to use the bit-field instructions. The `-m68020` option implies this option. This is the default if you use a configuration designed for a 68020 processor.

-mcpu32: Specifying this option tells GCC to generate code for a CPU32 processor core. This is the default when the compiler is configured for CPU32-based systems. You should use this option when compiling code for microcontrollers with a CPU32 or CPU32+ core, including the 68330, 68331, 68332, 68333, 68334, 68336, 68340, 68341, 68349, and 68360 processors. The `-mcpu32` option implies the `-mnobitfield` option.

-mfpa: Specifying this option tells GCC to generate code that uses the Sun FPA instructions for floating point.

-mnno-align-int: Specifying this option tells GCC to align `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 16-bit boundary. This is the default.

-mnno-strict-align: Specifying this option tells GCC to assume that unaligned memory references will be handled by the system.

-mnobitfield: Specifying this option tells GCC not to use the bit-field instructions. The `-m68000`, `-mc当地32` and `-m5200` options imply the `-mnobitfield` option.

-mpcrel: Specifying this option tells GCC to use the PC-relative addressing mode of the 68000 directly, instead of using a global offset table. This option implies the standard GCC **-fpic** option, which therefore allows at most a 16-bit offset for PC-relative addressing. The **-fpic** option is not presently supported with the **-mpcrel** option.

-mrtd: Specifying this option tells GCC to use a function-calling convention where functions that take a fixed number of arguments return with the rtd instruction, which pops their arguments during the return. The rtd instruction is supported by the 68010, 68020, 68030, 68040, 68060, and CPU32 processors, but not by the 68000 or 5200 processors. Using the rtd instruction saves one instruction in the caller since there is no need to pop the arguments there. This calling convention is incompatible with the one normally used on Unix, and therefore cannot be used if you need to call libraries that have been compiled with generic Unix compilers. When using this option, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`): if you do not, incorrect code will be generated for calls to those functions. You must also be careful to never call functions with extra arguments, which would result in seriously incorrect code.

TIP *To optimize heavily used functions, you can specify that an individual function is called with this calling sequence with the function attribute stdcall. You can also override the **-mrtd** option for specific functions by using the function attribute cdecl.*

-mshort: Specifying this option tells GCC to consider type `int` to be 16 bits wide, like `short int`.

-msoft-float: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC (except for the embedded GCC build targets `m68k-*-aout` and `m68k-*-coff`), but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

-mstrict-align: Specifying this option tells GCC not to assume that unaligned memory references will be handled by the system, and to perform the alignment itself.

M68hc1x Options

The M68HC1x family of microprocessors is a 16-bit microprocessor that is typically used as a microcontroller in embedded applications.

GCC options available when compiling code for M68hc1x systems are the following:

- m6811 | -m68hc11: Specifying this option tells GCC to generate code for a 68HC11 microcontroller. This is the default when the compiler is configured for 68HC11-based systems.

- m6812 | -m68hc12: Specifying this option tells GCC to generate code for a 68HC12 microcontroller. This is the default when the compiler is configured for 68HC12-based systems.

- mauto-incdec: Specifying this option tells GCC to use 68HC12 pre- and post- auto-increment and auto-decrement addressing modes.

- mshort: Specifying this option tells GCC to consider the int data type to be 16-bits wide, like short int.

- msoft-reg-count=*count*: Specifying this option enables you to tell GCC the number of pseudo-soft registers that are used for the code generation. The maximum number is 32. Depending on the your program, using more pseudo-soft registers may or may not result in improved performance. The default is 4 for 68HC11 and 2 for 68HC12.

M88K Options

The Motorola 88000 family of RISC processors were primarily designed for use in workstations such as the Data General AViiON.

GCC options available when compiling code for M88K systems are the following:

- m88000: Specifying this option tells GCC to generate code that works well on both the m88100 and the m88110 processors.

- m88100: Specifying this option tells GCC to generate code that works best for the m88100 processor, but that also runs on the m88110 processor.

- m88110: Specifying this option tells GCC to generate code that works best for the m88110 processor, and may not run on the m88100 processor.

- mbig-pic: This deprecated option has the same effect as the -fPIC option on M88K systems.

-mcheck-zero-division: Specifying this option tells GCC to generate code that guarantees that integer division by zero will be detected. This is the default. Some MC88100 processors do not correctly detect integer division by zero, though all MC88110 processors do. Using this option as a default generates code that will execute correctly on systems using either type of processor. This option is ignored if the **-m88110** option is specified.

-mhandle-large-shift: Specifying this option tells GCC to generate code that detects bit shifts of more than 31 bits and emits code to handle them properly.

-midentify-revision: Specifying this option tells GCC to include an `ident` directive in the assembler output that identifies the name of the source file, identifies the name and version of GCC, provides a timestamp, and records the GCC options used.

-mno-check-zero-division: Specifying this option tells GCC not to generate code that guarantees that integer division by zero will be detected. The MC88100 processor does not always trap on integer division by zero, so GCC generates additional code to explicitly check for zero divisors and trap with exception 503 when this is detected. This option is useful to reduce code size and possibly increase performance when running on systems with an MC88110 processor, which correctly detects all instances of integer division by zero.

-mno-ocs-debug-info: Specifying this option tells GCC not to include additional debugging information (about registers used in each stack frame) as specified in the 88open OCS (Object Compatibility Standard). This is the default on M88K systems running operating systems other than DG/UX, SVr4, and Delta 88 SVr3.2.

-mno-ocs-frame-position: Specifying this option tells GCC use the offset from the frame pointer register (register 30) when emitting COFF debugging information for automatic variables and parameters stored on the stack. When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the **-g** switch. This is the default on M88K systems running operating systems other than DG/UX, SVr4, Delta88 SVr3.2, and BCS.

-mno-optimize-arg-area: Specifying this option tells GCC not to reorganize the stack frame to save space, which results in increased memory use by the application. This is the default. The generated code conforms to the 88Open Object Compatibility Standard specification.

-mno-serialize-volatile: Specifying this option tells GCC not to generate code to guarantee the sequential consistency of volatile memory references. This option is useful because, by default, GCC generates code to guarantee serial consistency, even on the M88100 processor, where serial consistency is guaranteed. This is done to enable the same code to run on M88110 systems, where the order of memory references does not always match the order of the instructions requesting those references. For example, on an M88110 processor, a load instruction may execute before a preceding store instruction. If you intend to run your code only on the MC88100 processor, using the `-mno-serialize-volatile` option will produce smaller, faster code.

-mno-underscores: Specifying this option tells GCC to emit symbol names in assembler output without adding an underscore character at the beginning of each name. This is used for integration with linkers that do not follow the standard name mangling conventions. The default is to use an underscore as a prefix on each name.

-mocs-debug-info: Specifying this option tells GCC to include additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard. This extra information allows debugging of code that has had the frame pointer eliminated. This is the default for M88K systems running DG/UX, SVr4, and Delta 88 SVr3.2.

-mocs-frame-position: Specifying this option tells GCC to use the offset from the canonical frame address when emitting COFF debugging information for automatic variables and parameters stored on the stack. The canonical frame address is the stack pointer (register 31) on entry to the function. This is the default on M88K systems running DG/UX, SVr4, Delta88 SVr3.2, and BCS.

-moptimize-arg-area: Specifying this option tells GCC to save space by reorganizing the stack frame, reducing memory consumption. This option generates code that does not agree with the 88Open Object Compatibility Standard specification.

-mserialize-volatile: Specifying this option tells GCC to generate code that guarantees the serial consistency of volatile memory references. This is the default.

-mshort-data-num: Specifying this option tells GCC to generate smaller data references by making them relative to r0, which allows loading a value using a single instruction (rather than the usual two). The value specified for *num* enables you to control which data references are affected by this option by identifying the maximum displacement of short references that will be handled in this fashion. For example, specifying the **-mshort-data-512** option limits affected data references to those involving displacements of less than 512 bytes. The maximum value for *num* is 64K.

-msvr3: Specifying this option tells GCC to turn off compiler extensions related to System V Release 4 (SVR4). This option is the default for all GCC M88K build configurations other than the **m88k-motorola-sysv4** and **m88k-dg-dgux m88k** configurations.

-msvr4: Specifying this option tells GCC to turn on compiler extensions related to SVR4. Using this option makes the C preprocessor recognize `#pragma weak` and causes GCC to issue additional declaration directives that are used in SVR4. This option is the default for the **m88k-motorola-sysv4** and **m88k-dg-dgux m88k** GCC build configurations.

-mtrap-large-shift: Specifying this option tells GCC to generate code that traps on bit shifts of more than 31 bits. This is the default.

-muse-div-instruction: Specifying this option tells GCC to use the `div` instruction for signed integer division on the MC88100 processor. By default, the `div` instruction is not used. On the MC88100 processor, the signed integer division instruction traps to the operating system on a negative operand. The operating system transparently completes the operation, but at a large cost in execution time. On the MC88110 processor, the `div` instruction (also known as the `divs` instruction) processes negative operands without trapping to the operating system. Using this option causes GCC to generate code that will run correctly on either type of processor. This option is ignored if the **-m88110** option is specified.

NOTE *The result of dividing INT_MIN by -1 is undefined. In particular, the behavior of such a division with and without using the -muse-div-instruction option may differ.*

-mversion-03.00: This option is obsolete, and is ignored.

-mwarn-passed-structs: Specifying this option tells GCC to display a warning when a function passes a struct as an argument or result. Structure-passing conventions have changed during the evolution of the C language, and are often the source of portability problems. By default, GCC does not issue a warning.

MCore Options

Motorola's MCore processor family is a family of general-purpose 32-bit microcontrollers. Also known as M-Core or M*Core processors, the MCore family of processors is often used in embedded devices such as those employed for industrial control and measurement, health care and scientific equipment, and security systems.

GCC options available when compiling code for MCore-based systems are the following:

- m210:** Specifying this option tells GCC to generate code for the 210 processor.
- m340:** Specifying this option tells GCC to generate code for the 340 processor.
- m4byte-functions:** Specifying this option tells GCC to force all functions to be aligned to a 4-byte boundary.
- mbig-endian:** Specifying this option tells GCC to generate code for a big endian target.
- mcallgraph-data:** Specifying this option tells GCC to emit callgraph information.
- mdiv:** Specifying this option tells GCC to use the divide instruction. This is the default.
- mhardlit:** Specifying this option tells GCC to inline constants in the code stream if it can be done in two instructions or less.
- mlittle-endian:** Specifying this option tells GCC to generate code for a little endian target.
- mno-4byte-functions:** Specifying this option tells GCC not to force all functions to be aligned to a 4-byte boundary.
- mno-callgraph-data:** Specifying this option tells GCC not to emit callgraph information.

- mno-div: Specifying this option tells GCC not to use the divide instruction, replacing it with subtraction/remainder operations.
- mno-hardlit: Specifying this option tells GCC not to inline constants in the code stream.
- mno-relax-immediate: Specifying this option tells GCC not to allow arbitrarily sized immediates in bit operations.
- mno-slow-bytes: Specifying this option tells GCC not to prefer word access when reading byte quantities.
- mno-wide-bitfields: Specifying this option tells GCC not to treat bit fields as int-sized.
- mrelax-immediate: Specifying this option tells GCC to allow arbitrarily sized immediates in bit operations.
- mslow-bytes: Specifying this option tells GCC to prefer word access when reading byte quantities.
- mwide-bitfields: Specifying this option tells GCC to treat bit fields as int-sized.

MIPS Options

MIPS, an acronym for Microprocessor without Interlocked Pipeline Stages, is a microprocessor architecture developed by MIPS Computer Systems, Inc. based on research and development done at Stanford University. The MIPS R2000 and R3000 processors were 32-bit processors, while later processors such as the R5000, R8000, R10000, R12000, and R16000 are all 64-bit processors. The R6000 processor was a third-party R3000 that quickly vanished, while the R7000 was targeted for embedded use and never saw wide deployment. Processors based on various MIPS cores are widely used in embedded systems.

GCC options available when compiling code for MIPS-based systems are the following:

- EB: Specifying this option tells GCC to compile code for the processor in big endian mode. The required libraries are assumed to exist.
- EL: Specifying this option tells GCC to generate code for the processor in little endian mode. The required libraries are assumed to exist.

-G *num*: Specifying this option tells GCC to put global and static objects less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss sections. Using this option enables the assembler to emit one-word memory reference instructions based on the global pointer (GP or \$28), instead of using the normal two words. By default, *num* is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The **-G *num*** switch is also passed to the assembler and linker.

NOTE All modules should be compiled with the same **-G *num*** value. Compiling with different values of *num* may or may not work. If it does not, the linker will display an error message and exit.

-m4650: Specifying this option is a convenient shortcut for specifying the **-msingle-float**, **-mmad**, and **-mcpu=r4650** options.

-mabi=32: Specifying this option tells GCC to generate code for the 32-bit application binary interface (ABI). The default instruction level is **-mips1** when using this option.

-mabi=64: Specifying this option tells GCC to generate code for the 64-bit application binary interface (ABI). The default instruction level is **-mips4** when using this option.

-mabi=eabi: Specifying this option tells GCC to generate code for the extended application binary interface (EABI). The default instruction level is **-mips4** when using this option.

-mabi=n32: Specifying this option tells GCC to generate code for the new 32-bit ABI. The default instruction level is **-mips3** when using this option.

-mabi=o64: Specifying this option tells GCC to generate code for the old 64-bit ABI. The default instruction level is **-mips4** when using this option.

-mabicalls: Specifying this option tells GCC to generate code containing the pseudo operations **.abicalls**, **.cupload**, and **.cprestore** that some System V.4 ports use for position-independent code.

-march=*CPU-type*: Specifying this option tells GCC to use the defaults for the specified machine type *CPU-type* when generating instructions.

Possible values for *CPU-type* are r2000, r3000, r3900, r4000, r4100, r4300, r4400, r4600, r4650, r5000, r6000, r8000, and orion. The r2000, r3000, r4000, r5000, and r6000 values can be abbreviated as r2k (or r2K), r3k, and so on.

-mcpu=CPU-type: Specifying this option is the same as providing both the **-march** and **-mtune** options with a certain *CPU-type*. See the definition of the **-march** option for a list of valid values for *CPU-type*.

-mdouble-float: Specifying this option tells GCC to assume that the floating-point coprocessor supports double precision operations. This is the default.

-membedded-data: Specifying this option tells GCC to allocate variables in the read-only data section first if possible, then in the small data section if possible, or finally in the data section. This results in slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

-membedded-pic: Specifying this option tells GCC to generate PIC code that is suitable for embedded systems. All calls are made using PC-relative addresses, and all data is addressed using the \$gp register. No more than 65,536 (64K) bytes of global data may be used. Using this option requires GNU as and GNU ld, which do most of the work. This option currently only works on targets that use the ECOFF binary output format. It does not work with the ELF binary output format.

-mentry: Specifying this option tells GCC to use the entry and exit pseudo ops. This option can only be used with the **-mips16** option.

-mfix7000: Specifying this option tells GCC to pass an option to the GNU assembler that will cause NOOPs to be inserted if a read of the destination register of an **mphi** or **mflo** instruction occurs in the following two instructions.

-mflush-func=func: Specifying this option tells GCC the function to call in order to flush the I and D caches. The function must take the same arguments as the common `_flush_func()` function, which are the address of the memory range for which the cache is being flushed, the size of the memory range, and the number 3 (to flush both caches). The default is usually `_flush_func` or `__cpu_flush`, and is defined by the macro `TARGET_SWITCHES` in the machine description file.

-mfp32: Specifying this option tells GCC to assume that 32 32-bit floating-point registers are available. This is the default.

-mfp64: Specifying this option tells GCC to assume that 32 64-bit floating-point registers are available. This is the default when the **-mips3** option is used.

- mfused-madd: Specifying this option tells GCC to generate code that uses the floating-point multiply and accumulate instructions when they are available. These instructions are generated by default if they are available.
- mgas: Specifying this option tells GCC to generate code for the GNU assembler. This is the default on the OSF/1 reference platform, which uses the OSF/rose object format. More significantly, this is the default if the GCC configuration option --with-gnu-as is used, which is the GCC default.
- mgp32: Specifying this option tells GCC to assume that 32 32-bit general-purpose registers are available. This is the default.
- mgp64: Specifying this option tells GCC to assume that 32 64-bit general-purpose registers are available. This is the default when the -mips3 option is used.
- mgpopt: Specifying this option tells GCC to write all of the data declarations before the instructions in the text section. This allows the MIPS assembler to generate one-word memory references instead of using two words for short global or static data items. This is the default if optimization is selected.
- mhalf-pic: Specifying this option tells GCC to put pointers to extern references into the data section, rather than putting them in the text section.
- mhard-float: Specifying this option tells GCC to generate code that contains floating-point instructions. This is the default.
- mint64: Specifying this option tells GCC to force int and long types to be 64-bits wide. See the discussion of the -mlong32 option for an explanation of the default and the width of pointers.
- mips1: Specifying this option tells GCC to generate instructions that are conformant to level 1 of the MIPS instruction set architecture (ISA). This is the default. r3000 is the default *CPU-type* at this ISA level. The default ABI is 32 (-mabi=32) when using this option.
- mips16: Specifying this option tells GCC to enable the use of 16-bit instructions.
- mips2: Specifying this option tells GCC to generate instructions that are conformant to level 2 of the MIPS ISA (branch likely, square root instructions). r6000 is the default *CPU-type* at this ISA level. The default ABI is 32 (-mabi=32) when using this option.

-mips3: Specifying this option tells GCC to generate instructions that are conformant to level 3 of the MIPS ISA (64-bit instructions). r4000 is the default *CPU-type* at this ISA level. The default ABI is 64 (-mabi=64) when using this option.

-mips4: Specifying this option tells GCC to generate instructions that are conformant to level 4 of the MIPS ISA (conditional move, prefetch, enhanced FPU instructions). r8000 is the default *CPU-type* at this ISA level. The default ABI is 64 (-mabi=64) when using this option.

-mlong32: Specifying this option tells GCC to force long, int, and pointer types to be 32-bits wide.

NOTE *If none of the options -mlong32, -mlong64, or -mint64 are set, the size of ints, longs, and pointers depends on the ABI and ISA chosen. For -mabi=32, and -mabi=n32, ints and longs are 32-bits wide. For -mabi=64, ints are 32-bits wide, and longs are 64-bits wide. For -mabi=eabi and either -mips1 or -mips2, ints and longs are 32-bits wide. For -mabi=eabi and higher ISAs, ints are 32 bits, and longs are 64-bits wide. The width of pointer types is the smaller of the width of longs or the width of general-purpose registers (which in turn depends on the ISA).*

-mlong64: Specifying this option tells GCC to force long types to be 64-bits wide. See the discussion of the -mlong32 option for an explanation of the default and the width of pointers.

-mlong-calls: Specifying this option tells GCC to make all function calls using the JALR instruction, which requires loading a function's address into a register before making the call. You must use this option if you call outside of the current 512-megabyte segment to functions that are not called through pointers.

-mmad: Specifying this option tells GCC to permit the use of the mad, madu, and mul instructions, as on the r4650 chip.

-memcpy: Specifying this option tells GCC to make all block moves call the appropriate string function (`memcpy()` or `bcopy()`) instead of possibly generating inline code.

-mmips-as: Specifying this option tells GCC to generate code for the MIPS assembler, and invoke `mips-tfile` to add normal debug information. If either of the `-gstabs` or `-gstabs+` switches are used, the `mips-tfile` program will encapsulate the stabs within the MIPS ECOFF binary output format. This option is the default for all platforms except for the OSF/1 reference platform, or unless GCC has been configured and built with the `--with-gnu-as` option. The latter fact probably means that this option is no longer meaningful for anyone but a computer collector.

-mmips-tfile: Specifying this option tells GCC to post-process the object file with the `mips-tfile` program, which adds debugging support after the MIPS assembler has generated it. This option is only relevant if you are using the MIPS assembler.

-mno-abicalls: Specifying this option tells GCC not to generate code containing the pseudo operations `.abicalls`, `.cupload`, and `.cprestore` that some System V.4 ports use for position-independent code.

-mno-flush-func: Specifying this option tells GCC not to call any function to flush the I and D caches.

-mno-fused-madd: Specifying this option tells GCC not to generate code that uses the floating-point multiply and accumulate instructions, even if they are available. These instructions may be undesirable if the extra precision causes problems or on certain chips in the modes where denormals are rounded to zero and where denormals generated by multiply and accumulate instructions cause exceptions anyway.

-mno-gopt: Specifying this option tells GCC not to write all of the data declarations before the instructions in the text section, preventing some optimizations but resulting in more readable assembly code.

-mno-long-calls: Specifying this option tells GCC not to use the `JALR` instruction when making function calls.

-mno-mad: Specifying this option tells GCC not to permit the use of the `mad`, `madu`, and `mul` instructions.

-mno-memcpy: Specifying this option tells GCC to possibly generate inline code for all block moves rather than calling the appropriate string function (`memcpy()` or `bcopy()`).

-mno-mips-tfile: Specifying this option tells GCC not to post-process the object file with the `mips-tfile` program, which adds debugging support after the MIPS assembler has generated it. If the `mips-tfile` program is not run, then no local variables will be available to the debugger. In addition, `stage2` and `stage3` objects will have the temporary filenames passed to the assembler embedded in the object file, which means the objects will not compare the same. The `-mno-mips-tfile` switch should only be used when there are bugs in the `mips-tfile` program that prevents compilation. This option is the default in modern versions of GCC, which all use the GNU assembler.

-mno-mips16: Specifying this option tells GCC not to use 16-bit instructions.

-mno-embedded-data: Specifying this option tells GCC to allocate variables in the data section, as usual. This may result in faster code but increases the amount of RAM required to run an application.

-mno-embedded-pic: Specifying this option tells GCC not to make all function calls using PC-relative addresses, and not to address all data using the `$gp` register. This is the default on systems that use the ELF binary output format, or use an assembler or linker/loader other than the GNU tools.

-mno-half-pic: Specifying this option tells GCC to put pointers to extern references in the text section. This option is the default.

-mno-rnames: Specifying this option tells GCC to generate code that uses the hardware names for the registers (i.e., `$4`). This is the default.

-mno-split-addresses: Specifying this option tells GCC not to generate code that loads the high and low parts of address constants separately, which prevents some optimizations but may be necessary if using a non-GNU assembler or linker/loader.

-mno-stats: Specifying this option tells GCC not to emit statistical information when processing noninline functions. This is the default.

-mno-uninit-const-in-rodata: Specifying this option tells GCC to store uninitialized const variables in the data section, as usual. This is the default.

-mrnames: Specifying this option tells GCC to generate code that uses the MIPS software names for the registers, instead of the hardware names (i.e., `A0` instead of `$4`). The only known assembler that supports this option is the Algorithmics assembler.

-msingle-float: Specifying this option tells GCC to assume that the floating-point coprocessor only supports single precision operations, as on the r4650 chip.

-msoft-float: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

-msplit-addresses: Specifying this option tells GCC to generate code that loads the high and low parts of address constants separately. This allows GCC to optimize away redundant loads of the high order bits of addresses. This optimization requires GNU as and GNU ld. This optimization is enabled by default for some embedded targets where GNU as and GNU ld are standard.

-mstats: Specifying this option tells GCC to emit one line to the standard error output for each noninline function processed. This line provides statistics about the program (number of registers saved, stack size, and so on).

-mtune=CPU-type: Specifying this option tells GCC to use the defaults for the specified machine type *CPU-type* when scheduling instructions. Possible values for *CPU-type* are r2000, r3000, r3900, r4000, r4100, r4300, r4400, r4600, r4650, r5000, r6000, r8000, and orion. The r2000, r3000, r4000, r5000, and r6000 values can be abbreviated as r2k (or r2K), r3k, etc.

NOTE Selecting a specific *CPU-type* will schedule things appropriately for that particular chip, but GCC will not generate any code that does not satisfy level 1 of the MIPS ISA without a *-mipsX* or *-mabi* switch being used.

-munit-init-const-in-readonly: Specifying this option tells GCC to store uninitialized const variables in the read-only data section. This option must be used with the *-membedded-data* option.

-no-cpp: Specifying this option instructs the MIPS assembler not to run its preprocessor over user assembler files (with an .s suffix) when assembling them.

-no-crt0: Specifying this option tells GCC not to include the default crt0 C runtime initialization library.

MMIX Options

MIX was a virtual computer system designed for use as an example in Donald Knuth's legendary *The Art of Computer Programming* books. MMIX is a 64-bit RISC-oriented follow-up to MIX.

GCC options available when compiling code for MMIX-based systems are the following:

-mabi-gnu: Specifying this option tells GCC to generate code that is conformant to the GNU ABI and therefore passes function parameters and return values in global registers \$231 and up.

-mabi=mmixware: Specifying this option tells GCC to generate code that passes function parameters and return values that (in the called function) are seen as registers \$0 and up.

-mbranch-predict: Specifying this option tells GCC to use the probable-branch instructions when static branch prediction indicates a probable branch.

-mbase-addresses: Specifying this option tells GCC to generate code that uses `_base_addresses_`. Using a base address automatically generates a request (handled by the assembler and the linker) for a constant to be set up in a global register. The register is used for one or more base address requests within the range 0 to 255 from the value held in the register. This generally leads to short and fast code, but the number of different data items that can be addressed is limited. This means that a program that uses lots of static data may require `-mno-base-addresses`.

-melf: Specifying this option tells GCC to generate code in ELF format, rather than in the default mmo format used by the MMIX simulator.

-mepsilon: Specifying this option tells GCC to generate floating-point comparison instructions that compare with respect to the rE epsilon register.

-mknuthdiv: Specifying this option tells GCC to make the result of a division yielding a remainder have the same sign as the divisor.

-mlibfuncs: Specifying this option tells GCC that all intrinsic library functions are being compiled, passing all values in registers, no matter the size.

-mno-base-addresses: Specifying this option tells GCC not to generate code that uses `_base` addresses. Using a base address generally leads to short and fast code, but limits the number of different data items that can be addressed. Programs that uses significant amounts of static data may require `-mno-base-addresses`.

-mno-branch-predict: Specifying this option tells GCC not to use the probable-branch instructions.

-mno-epsilon: Specifying this option tells GCC not to generate floating-point comparison instructions that use the `rE` epsilon register.

-mno-knuthdiv: Specifying this option tells GCC to make the result of a division yielding a remainder have the same sign as the dividend. This is the default.

-mno-libfuncs: Specifying this option tells GCC that all intrinsic library functions are not being compiled and that values should therefore not be passed in registers.

-mno-toplevel-symbols: Specifying this option tells GCC not to insert a colon (:) at the beginning of all global symbols. This disallows the use of the PREFIX assembly directive with the resulting assembly code.

-mno-zero-extend: Specifying this option tells GCC to use sign-extending load instructions when reading data from memory in sizes shorter than 64 bits.

-mtoplevel-symbols: Specifying this option tells GCC to insert a colon (:) at the beginning of all global symbols, so that the assembly code can be used with the PREFIX assembly directive.

-mzero-extend: Specifying this option tells GCC to use zero-extending load instructions when reading data from memory in sizes shorter than 64 bits.

MN10200 Options

The MN10200 series of 16-bit single-chip microcontrollers are low-power processors with a 16MB address space and fast instruction execution time complemented by a three-stage pipeline.

GCC options available when compiling code for MN10200-based systems are the following:

-mrelax: Specifying this option tells GCC to tell the linker that it should perform a relaxation optimization pass to shorten branches, calls, and absolute memory addresses. This option is only useful when specified on the command line for the final link step. Using this option makes symbolic debugging impossible.

MN10300 Options

The MN10300 series of 32-bit single-chip microcontrollers are the descendants of the MN10200 processors, and add built-in support for multimedia applications to the core capabilities of the MN10200.

GCC options available when compiling code for MN10300-based systems are the following:

-mam33: Specifying this option tells GCC to generate code that uses features specific to the AM33 processor.

-mmult-bug: Specifying this option tells GCC to generate code that avoids bugs in the multiply instructions for the MN10300 processors. This is the default.

-mno-am33: Specifying this option tells GCC not to generate code that uses features specific to the AM33 processor. This is the default.

-mno-crt0: Specifying this option tells GCC not to link in the C runtime initialization object file.

-mno-mult-bug: Specifying this option tells GCC not to generate code that avoids bugs in the multiply instructions for the MN10300 processors. This may result in smaller, faster code if your application does not trigger these bugs.

-mrelax: Specifying this option tells GCC to tell the linker that it should perform a relaxation optimization pass to shorten branches, calls, and absolute memory addresses. This option is only useful when specified on the command line for the final link step. Using this option makes symbolic debugging impossible.

NS32K Options

The National Semiconductor NS32000 (a.k.a. NS32K) family of processors was used in a variety of older computer systems and is still used in embedded and signal processing applications.

GCC options available when compiling code for NS32000-based systems are the following:

-m32032: Specifying this option tells GCC to generate code for a 32032 processor. This is the default when the compiler is configured for 32032 and 32016-based systems.

-m32081: Specifying this option tells GCC to generate code containing 32081 instructions for floating point. This is the default for all systems.

- m32332: Specifying this option tells GCC to generate code for a 32332 processor. This is the default when the compiler is configured for 32332-based systems.
- m32381: Specifying this option tells GCC to generate code containing 32381 instructions for floating point. Specifying this option also implies the -m32081 option. The 32381 processor is only compatible with the 32332 and 32532 CPUs. This is the default for GCC's pc532-netbsd build configuration.
- m32532: Specifying this option tells GCC to generate code for a 32532 processor. This is the default when the compiler is configured for 32532-based systems.
- mbitfield: Specifying this option tells GCC to use bit-field instructions. This is the default for all platforms except the pc532.
- mhimem: Specifying this option tells GCC to generate code that can be loaded above 512MB. Many NS32000 series addressing modes use displacements of up to 512MB. If an address is above 512MB, then displacements from zero cannot be used. This option is often useful for operating systems or ROM code.
- mmulti-add: Specifying this option tells GCC to attempt to generate the multiply-add floating-point instructions polyF and dotF. This option is only available if the -m32381 option also specified. Using these instructions requires changes to register allocation that generally have a negative impact on performance. This option should only be enabled when compiling code that makes heavy use of multiply-add instructions.
- mnoabitfield: Specifying this option tells GCC not to use the bit-field instructions. On some machines, such as the pc532, it is faster to use shifting and masking operations. This is the default for the pc532.
- mnohimem: Specifying this option tells GCC to assume that code will be loaded in the first 512MB of virtual address space. This is the default for all platforms.
- mnomulti-add: Specifying this option tells GCC not to generate code containing the multiply-add floating-point instructions polyF and dotF. This is the default on all platforms.
- mnoregparam: Specifying this option tells GCC not to pass any arguments in registers. This is the default for all targets.

-mno_sb: Specifying this option tells GCC not to use the sb register as an index register. This is usually because it is not present or is not guaranteed to have been initialized. This option is the default for all targets except the pc532-netbsd. This option is also implied whenever the **-mhimem** or **-fpic** options are specified.

-mregparam: Specifying this option tells GCC to use a different function-calling convention where the first two arguments are passed in registers. This calling convention is incompatible with the one normally used on Unix, and therefore should not be used if your application calls libraries that have been compiled with the Unix compiler.

-mrtd: Specifying this option tells GCC to use a function-calling convention where functions that take a fixed number of arguments pop their arguments during the return. This calling convention is incompatible with the one normally used on Unix, and therefore cannot be used if you need to call libraries that have been compiled with generic Unix compilers. When using this option, you must provide function prototypes for all functions that take variable numbers of arguments (including printf); if you do not, incorrect code will be generated for calls to those functions. You must also be careful to never call functions with extra arguments, which would result in seriously incorrect code. This option takes its name from the 680x0 rtd instruction.

-msoft-float: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

-msb: Specifying this option tells GCC that the sb can be used as an index register that is always loaded with zero. This is the default for the pc532-netbsd target.

PDP-11 Options

The options in this section are specific to using GCC to compile applications for the former Digital Equipment Corporations PDP-11 minicomputers. Few of these systems are still in use, aside from some that are still used in older process control applications. However, these options can still be useful if, like one of the authors of this book, you happen to have PDP-11 in the basement, “just in case.”

GCC options available when compiling code for PDP11 systems are the following:

- m10: Specifying this option tells GCC to generate code for a PDP-11/10.
- m40: Specifying this option tells GCC to generate code for a PDP-11/40.
- m45: Specifying this option tells GCC to generate code for a PDP-11/45. This is the default.
- mabshi: Specifying this option tells GCC to use the abshi2 pattern. This is the default.
- mac0: Specifying this option tells GCC to return floating-point results in ac0 (fr0 in Unix assembler syntax).
- mbcopy: Specifying this option tells GCC not to use inline movstrhi patterns for copying memory.
- mbcopy-builtin: Specifying this option tells GCC to use inline movstrhi patterns for copying memory. This is the default.
- mbranch-cheap: Specifying this option tells GCC not to assume that branches are expensive. This is the default.
- mbranch-expensive: Specifying this option tells GCC to assume that branches are expensive. This option is designed for experimenting with code generation, and is not intended for production use.
- mdec-asm: Specifying this option tells GCC to use DEC (Compaq? HP?) assembler syntax. This is the default when GCC is configured for any PDP-11 build target other than pdp11-*-bsd.
- mfloating32 | -mno-float64: Specifying either of these options tells GCC to use 32-bit floats.
- mfloating64 | -mno-float32: Specifying either of these options tells GCC to use 64-bit floats. This is the default.
- mfpu: Specifying this option tells GCC to use hardware FPP floating point. This is the default. (FIS floating point on the PDP-11/40 is not supported.)
- mint16 | -mno-int32: Specifying either of these options tells GCC to use 16-bit ints. This is the default.
- mint32 | -mno-int16: Specifying either of these options tells GCC to use 32-bit ints.
- mno-abshi: Specifying this option tells GCC not to use the abshi2 pattern.

-mno-ac0: Specifying this option tells GCC to return floating-point results in memory. This is the default.

-mno-split: Specifying this option tells GCC to generate code for a system without split Instruction and Data spaces. This is the default.

-msoft-float: Specifying this option tells GCC that floating-point support should not be assumed, and to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC, but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system or these function calls will be identified as unresolved.

-msplit: Specifying this option tells GCC to generate code for a system with split Instruction and Data spaces.

-munix-asm: Specifying this option tells GCC to use Unix assembler syntax. This is the default when GCC is configured for the `pdp11-*-bsd` build target.

PowerPC (PPC) Options

The PowerPC is a RISC microprocessor architecture that was originally created by the 1991 Apple-IBM-Motorola alliance, known as AIM. The PowerPC was the CPU portion of the overall AIM platform, and is the only surviving aspect of the platform. Performance Optimization With Enhanced RISC (POWER) had been introduced (and was doing well) as a multichip processor on the IBM RS/6000 workstation, but IBM was interested in a single-chip version as well as entering other markets. At the same time, Apple was looking for a new processor to replace the aging MC680x0 processors in its Macintosh computers.

GCC provides options that enable you to specify which instructions are available on the processor you are using. GCC supports two related instruction set architectures for the PowerPC and RS/6000. The POWER instruction set consists of those instructions that are supported by the RIOS chip set used in the original RS/6000 systems. The PowerPC instruction set is the architecture of the Motorola MPC5xx, MPC6xx, MPC8xx microprocessors, and the IBM 4xx microprocessors. Neither architecture is a subset of the other, but a large common subset of instructions are supported by both. An MQ register is included in processors supporting the POWER architecture.

TIP *In general, it is easier to use the `-mcpu=CPU-type` option, which implies the appropriate instruction set, rather than trying to remember the appropriate `-mpower*` option.*

IBM's RS64 processor family is a modified PowerPC architecture. These processors are used in the AS/400 computer family, and in some RS/6000 systems. The latest generation of PowerPC processors (the G5) is used in Apple's latest Macintosh computer systems. PowerPC chips based on older cores are tremendously popular in embedded hardware.

GCC options available when compiling code for PowerPC systems are the following:

-G num: Specifying this option on embedded PowerPC systems tells GCC to put global and static objects less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss sections. By default, *num* is 8. The `-G num` switch is also passed to the assembler and linker.

NOTE *All modules should be compiled with the same `-G num` value. Compiling with different values of *num* may or may not work. If it does not, the linker will display an error message and exit.*

-mabi=altivec: Specifying this option tells GCC to extend the current ABI with AltiVec ABI extensions. This does not change the default ABI, but simply adds the AltiVec ABI extensions to the current ABI.

-mabi=no-altivec: Specifying this option tells GCC to disable AltiVec ABI extensions for the current ABI.

-mads: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called `crt0.o` and the standard C libraries are `libads.a` and `libc.a`.

-maix-struct-return: Specifying this option tells GCC to return all structures in memory (as specified by the AIX ABI).

-maix32: Specifying this option tells GCC to use the 32-bit ABI, disabling the 64-bit AIX ABI and calling conventions. This is the default.

-maix64: Specifying this option tells GCC to enable the 64-bit AIX ABI and calling convention: 64-bit pointers, 64-bit long type, and the infrastructure needed to support them. Specifying **-maix64** implies the **-mpowerpc64** and **-mpowerpc** options.

-maltivec: Specifying this option tells GCC to enable the use of built-in functions that provide access to the AltiVec instruction set. You may also need to set **-mabi=altivec** to adjust the current ABI with AltiVec ABI enhancements.

-mbig | -mbig-endian: Specifying this option on SVR4 or embedded PowerPC systems tells GCC to compile code for the processor in big endian mode.

-mbit-align: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to force structures and unions that contain bit fields to be aligned to the base type of the bit field. This option is the default. For example, by default a structure containing nothing but 8 unsigned bit fields of length 1 would be aligned to a 4-byte boundary and have a size of 4 bytes.

-mcall-aix: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions that are similar to those used on AIX. This is the default if you configured GCC using the **powerpc-*-eabiaix** GCC build target.

-mcall-gnu: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions similar to those used by the Hurd-based GNU system.

-mcall-linux: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions similar to those used by the Linux-based GNU system.

-mcall-netbsd: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions similar to those used by the NetBSD operating system.

-mcall-solaris: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions that are similar to those used by the Solaris operating system.

-mcall-sysv: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code using calling conventions that adheres to the March 1995 draft of the System V application binary interface, PowerPC processor supplement. This is the default unless you configured GCC using the **powerpc-*-eabiaix** GCC build target.

- mcall-sysv-eabi: Specifying this option is the same as using both the -mcall-sysv and -meabi options.
- mcall-sysv-noeabi: Specifying this option is the same as using both the -mcall-sysv and -mno-eabi options.
- mcpu=*CPU-type*: Specifying this option tells GCC to set the architecture type, register usage, choice of mnemonics, and instruction scheduling parameters to values associated with the machine type *CPU-type*. Possible values for *CPU-type* are rios, rios1, rsc, rios2, rs64a, 601, 602, 603, 603e, 604, 604e, 620, 630, 740, 7400, 7450, 750, power, power2, powerpc, 403, 505, 801, 821, 823, 860, and common. The -mcpu=power, -mcpu=power2, -mcpu=powerpc, and -mcpu=powerpc64 options specify generic POWER, POWER2, pure 32-bit PowerPC, and 64-bit PowerPC architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes. The other options specify a specific processor. Code generated under those options will run best on that processor, and may not run at all on others.

TIP Specifying the -mcpu=common option selects a completely generic processor. Code generated under this option will run on any POWER or PowerPC processor. GCC will use only the instructions in the common subset of both architectures, and will not use the MQ register. GCC assumes a generic processor model for scheduling purposes.

The -mcpu options automatically enable or disable other -m options as shown in Table B-1.

Table B-1. CPU and Implied Options for PPC Processors

CPU Option	Implied Options
common	-mno-power, -mno-powerc
power, power2, rios1, rios2, rsc	-mpower, -mno-powerpc, -mno-new-mnemonics
powerpc, rs64a, 602, 603, 603e, 604, 620, 630, 740, 7400, 7450, 750, 505	-mno-power, -mpowerpc, -mnew-mnemonics
601	-mpower, -mpowerpc, -mnew-mnemonics
403, 821, 860	-mno-power, -mpowerpc, -mnew-mnemonics, -msoft-float

-meabi: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to adhere to the EABI, which is a set of modifications to the System V.4 specifications. This means that the stack is aligned to an 8-byte boundary, a function `_eabi` is called to from `main()` to set up the EABI environment, and the `-msdata` option can use both `r2` and `r13` to point to two separate small data areas. This option is the default if you configured GCC using one of the `powerpc*-*-eabi*` build targets.

-memb: Specifying this option on embedded PowerPC systems tells GCC to set the `PPC_EMB` bit in the ELF flags header to indicate that EABI-extended relocations are used.

-mfull-toc: Specifying this option modifies the generation of the table of contents (TOC) generated for every PPC executable. If the `-mfull-toc` option is specified, GCC allocates at least one TOC entry for each unique nonautomatic variable reference in a program, and will also place floating-point constants in the TOC. A maximum of 16,384 entries are available in the TOC. The `-mfull-toc` option is the default.

-mfused-madd: Specifying this option tells GCC to generate code that uses the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used.

-mhard-float: Specifying this option tells GCC to generate code that uses the floating-point register set.

-mlittle | -mlittle-endian: Specifying either of these options on SVR4 or embedded PowerPC systems tells GCC to compile code for the processor in little endian mode.

-mminimal-toc: Specifying this option modifies the generation of the TOC that is generated for every PPC executable. The TOC provides a convenient way of looking up the address/entry point of specific functions. This option is a last resort if you see a linker error indicating that you have overflowed the TOC during final linking, and have already tried using the `-no-fp-in-toc` and `-mno-sum-in-toc` options. The `-mminimal-toc` option causes GCC to make only one TOC entry for every file. When you specify this option, GCC produces code that is slower and larger but uses extremely little TOC space. You may wish to use this option only on files that contain code that is infrequently executed.

-mmultiple: Specifying this option tells GCC to generate code that uses the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use **-mmultiple** on little endian PowerPC systems except for the PPC740 and PPC740, since those instructions do not usually work when the processor is in little endian mode. The PPC740 and PPC750 permit the use of these instructions in little endian mode.

-mmvme: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called crt0.o and the standard C libraries are libmvme.a and libc.a.

-mnew-mnemonics: Specifying this option tells GCC to use the assembler mnemonics defined for the PowerPC architecture. Instructions defined in only one architecture have only one mnemonic. GCC uses that mnemonic irrespective of which of these options is specified. GCC defaults to the mnemonics appropriate for the architecture that is in use. Unless you are cross-compiling, you should generally accept the default.

-mno-altivec: Specifying this option tells GCC to disable the use of built-in functions that provide access to the AltiVec instruction set.

-mno-bit-align: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to force structures and unions that contain bit fields to be aligned to the base type of the bit field. For example, by default a structure containing nothing but eight unsigned bit fields of length 1 would be aligned to a 4-byte boundary and have a size of 4 bytes. When using the **-mno-bit-align** option, the structure would be aligned to a 1-byte boundary and be 1 byte in size.

-mno-eabi: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to adhere to the EABI, which is a set of modifications to the System V.4 specifications. This means that the stack is aligned to a 16-byte boundary, an initialization function is not called from `main`, and that the **-msdata** option will only use `r13` to point to a single small data area. This is the default for all GCC build configurations other than the `powerpc*-*-eabi*` build targets.

-mno-fp-in-toc: Specifying this option tells GCC to generate the same TOC as specified by the **-mfull-toc** option, but not to store floating-point constants in the TOC. This option and the **-mno-sum-in-toc** option are typically used if you see a linker error indicating that you have overflowed the TOC during final linking.

-mno-fused-madd: Specifying this option tells GCC to generate code that does not use the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used.

-mno-multiple: Specifying this option tells GCC to generate code that does not use the load multiple word instructions or the store multiple word instructions. This option should not be used on little endian systems, with the exception of the 740 and 750 systems. These instructions are generated by default on POWER systems, and are not generated on PowerPC systems.

-mno-power: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the POWER architecture.

NOTE *If you specify both the -mno-power and -mno-powerpc options, GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register.*

-mno-power2: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the POWER2 architecture.

-mno-powerpc: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC architecture.

NOTE *If you specify both the -mno-power and -mno-powerpc options, GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register.*

-mno-powerpc64: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC-64 architecture.

-mno-powerpc-gpopt: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC architecture, including floating-point square root and the optional PowerPC architecture instructions in the General Purpose group.

-mno-powerpc-gfxopt: Specifying this option prevents GCC from generating code that uses any of the instructions that are specific to the PowerPC architecture, including floating-point select and the optional PowerPC architecture instructions in the Graphics group.

-mno-prototype: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to assume that all calls to variable argument functions are properly prototyped. Only calls to prototyped variable argument functions will set or clear bit 6 of the condition code register (CR) to indicate whether floating-point values were passed in the floating-point registers.

-mno-regnames: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to emit register names in the assembly language output using symbolic forms.

-mno-relocatable: Specifying this option on embedded PowerPC systems tells GCC not to generate code that enables the program to be relocated to a different address at runtime. This minimizes the size of the resulting executable, such as a boot monitor or kernel.

-mno-relocatable-lib: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to generate code that enables the program to be relocated to a different address at runtime.

-mno-strict-align: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to assume that unaligned memory references will be handled by the system.

-mno-string: Specifying this option tells GCC to generate code that does not the load string instructions or the store string word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems.

-mno-sum-in-toc: Specifying this option tells GCC to generate the same TOC as specified by the `-mfull-toc` option, but to generate code to calculate the sum of an address and a constant at runtime instead of putting that sum into the TOC. This option and the `-no-fp-in-toc` option are typically used if you see a linker error indicating that you have overflowed the TOC during final linking.

-mno-toc: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.

-mno-update: Specifying this option tells GCC to generate code that does not use the load or store instructions that update the base register to the address of the calculated memory location. (These instructions are generated by default.) If you use the `-mno-update` option, there is a small window between the time that the stack pointer is updated and when the address of the previous frame is stored, which means that code that walks the stack frame across interrupts or signals may get corrupted data.

-mno-xl-call: Specifying this option when compiling on AIX systems tells GCC not to pass floating-point arguments to prototyped functions beyond the register save area (RSA) on the stack in addition to argument FPRs. This is the default.

-mold-mnemonics: Specifying this option tells GCC to use the assembler mnemonics defined for the POWER architecture. Instructions defined in only one architecture have only one mnemonic. GCC uses that mnemonic irrespective of which of these options is specified. GCC defaults to the mnemonics appropriate for the architecture that is in use. Unless you are cross-compiling, you should generally accept the default.

-mpe: Specifying this option tells GCC to support the IBM RS/6000 SP Parallel Environment (PE). Applications written to use message passing must be linked with special startup code to enable the application to run. The system must have PE installed in the standard location (`/usr/lpp/ppe.poe/`), or GCC's specs file must be overridden by using the `-specs=` option to specify the appropriate directory location. The Parallel Environment does not support threads, so the `-mpe` option and the `-pthread` option are incompatible.

-mpower: Specifying this option tells GCC to generate code using instructions that are found only in the POWER architecture and to use the MQ register.

NOTE *Specifying both of the `-mpower` and `-mpowerpc` options permits GCC to use any instruction from either architecture and to allow use of the MQ register. Both of these options should be specified when generating code for the Motorola MPC601 processor.*

-mpower2: Specifying this option implies the `-mpower` option and also enables GCC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

-mpowerpc: Specifying this option tells GCC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture.

NOTE *Specifying both of the -mpowerpc and -mpower options permits GCC to use any instruction from either architecture and to allow use of the MQ register. Both of these options should be specified when generating code for the Motorola MPC601 processor.*

-mpowerpc64: Specifying the -mpowerpc64 option tells GCC to generate any PowerPC instructions as well as the additional 64-bit instructions that are found in the full PowerPC64 architecture and to treat GPRs as 64-bit, double-word quantities. GCC defaults to -mno-powerpc64.

-mpowerpc-gfxopt: Specifying this option implies the -mpowerpc option and also enables GCC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

-mpowerpc-gpopt: Specifying this option implies the -mpowerpc option and also enables GCC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root.

-mprototype: Specifying this option on SVR4 and embedded PowerPC systems tells GCC not to assume that all calls to variable argument functions are properly prototyped. The compiler will insert an instruction before every nonprototyped call to set or clear bit 6 of the condition code register (CR) to indicate whether floating-point values were passed in the floating-point registers in case the function takes a variable arguments.

-mregnames: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to emit register names in the assembly language output using symbolic forms.

-mrelocatable: Specifying this option on embedded PowerPC systems tells GCC to generate code that enables the program to be relocated to a different address at runtime. If you use the -mrelocatable option on any module, all objects linked together must be compiled with the -mrelocatable or -mrelocatable-lib options.

-mrelocatable-lib: Specifying this option on embedded PowerPC systems tells GCC to generate code that enables the program to be relocated to a different address at runtime. Modules compiled with the **-mrelocatable-lib** option can be linked with either modules compiled without the **-mrelocatable** and **-mrelocatable-lib** options or with modules compiled with the **-mrelocatable** option.

-msdata | -msdata=default: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to compile code as if the **-msdata=sysv** option were specified, unless the **-meabi** option was also specified, in which case GCC compiles code as if the **-msdata=eabi** option were specified.

-msdata-data: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to put small global and static data in the **sdata** section. Small uninitialized global and static data are put in the **sbss** section, but register **r13** is not used to address small data. This is the default behavior unless other **-msdata** options are specified.

-msdata=none | -mno-sdata: Specifying this option on embedded PowerPC systems tells GCC to put all initialized global and static data in the **data** section and all uninitialized data in the **bss** section.

-msdata=sysv: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to put small global and static data in the **sdata** section, which is pointed to by register **r13**. Small uninitialized global and static data are put in the **sbss** section, which is adjacent to the **sdata** section. This option is incompatible with the **-mrelocatable** option.

-msim: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called **sim-crt0.o** and that the standard C libraries are **libsim.a** and **libc.a**. This is the default for the **powerpc-*-eabisim** GCC build configuration.

-msoft-float: Specifying this option tells GCC to generate code that does not use the floating-point register set. Software floating-point emulation is provided if you use this option and pass it to GCC when linking.

-mstrict-align: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to assume that unaligned memory references will be handled by the system.

-mstring: Specifying this option tells GCC to generate code that uses the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use **-mstring** on little endian PowerPC systems except for PPC740 and PPC750 systems, since those instructions usually do not work when the processor is in little endian mode. PPC740 and PPC750 permit the use of these instructions in little endian mode.

-msvr4-struct-return: Specifying this option tells GCC to return structures smaller than 8 bytes in registers, as specified by the SVR4 ABI.

-mtoc: Specifying this option tells GCC to assume that register 2 contains a pointer to a global area pointing to the addresses used in the program on SVR4 and embedded PowerPC systems.

-mtune=*CPU-type*: Specifying this option tells GCC to set the instruction scheduling parameters for the machine type *CPU-type*, but not to set the architecture type, register usage, or choice of mnemonics, as **-mcpu=***CPU-type* would. The same values for *CPU-type* are used for **-mtune** as for **-mcpu**. If both are specified, the code generated will use the architecture, registers, and mnemonics set by **-mcpu**, but the scheduling parameters set by **-mtune**.

-mupdate: Specifying this option tells GCC to generate code that uses the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default.

-mvxworks: Specifying this option on SVR4 and embedded PowerPC systems tells GCC to that you are compiling for a VxWorks system. You have our sympathy.

-mxl-call: Specifying this option when compiling on AIX systems tells GCC to pass floating-point arguments to prototyped functions beyond the register save area (RSA) on the stack in addition to argument FPRs. The AIX calling convention was extended to handle an obscure K&R C case of calling a function that takes the address of its arguments with fewer arguments than declared. When a subroutine is compiled without optimization, AIX XL compilers access floating-point arguments that do not fit in the RSA from the stack. Because always storing floating-point arguments on the stack is inefficient and rarely needed, this option is not enabled by default and is only necessary when calling subroutines compiled by AIX XL compilers without optimization.

-myellowknife: Specifying this option on embedded PowerPC systems tells GCC to assume that the startup module is called crt0.o and the standard C libraries are libyk.a and libc.a. *Yellowknife* is the name of an old Motorola evaluation board.

-pthread: Specifying this option tells GCC to add support for multi-threading through the pthreads library. This option sets flags for both the preprocessor and linker.

RS/6000 Options

See the section “PowerPC (PPC) Options” of this appendix for options relevant to using GCC to compile applications for IBM’s RS/6000 family of workstations.

RT Options

The IBM RT was IBM’s first RISC workstation, and saw little use outside academia. It originally ran an operating system called Academic Operating System (AOS), which was loosely based on BSD Unix, but was later the original deployment platform for IBM’s AIX operating system. The IBM RT was also a primary development platform for the Mach operating system at Carnegie Mellon University.

GCC options available when compiling code for IBM RT systems are the following:

-mcall-lib-mul: Specifying this option tells GCC to call lmul\$\$ for integer multiples.

-mfpx-in-fpregs: Specifying this option tells GCC to use a calling sequence in which floating-point arguments are passed in floating-point registers. This calling sequence is incompatible with the IBM calling convention. Note that varargs.h and stdarg.h will not work with floating-point operands if this option is specified.

-mfpx-in-gregs: Specifying this option tells GCC to use the normal calling convention for floating-point arguments. This is the default.

-mfull-fp-blocks: Specifying this option tells GCC to generate full-size floating-point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

-mhc-struct-return: Specifying this option tells GCC to return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC (hc) compiler. See the **-fpcc-struct-return** option for similar compatibility with the Portable C Compiler (pcc).

-min-line-mul: Specifying this option tells GCC to use an inline code sequence for integer multiplies. This is the default.

-mminimum-fp-blocks: Specifying this option tells GCC not to include extra scratch space in floating-point data blocks. This results in smaller code but slower execution, since scratch space must be allocated dynamically.

-mnohc-struct-return: Specifying this option tells GCC to return some structures of more than one word in registers, when convenient. This is the default.

-mpcc-struct-return: Specifying this option tells GCC to use return conventions that are compatible with pcc.

S/390 and zSeries Options

The S/390 is the latest in IBM's family of 360 and 370 systems that stretches back to the 1960s. The zSeries is IBM's name for their latest series of high-powered servers. S/390 systems can run Linux in virtual machines, just as they run other operating systems. The zSeries machines can run Linux directly.

Linux supports the S/390 and zSeries processor architecture and some devices that are specific to S/390 and zSeries environments. Linux on these systems enables you to take advantage of the fast I/O and reliability that are traditional features of S/390 and zSeries mainframe hardware.

GCC options available when compiling code for S/390 and zSeries systems are the following:

-m31: Specifying this option tells GCC to generate code that is compliant to the Linux for S/390 ABI. This is the default for s390 targets.

-m64: Specifying this option tells GCC to generate code that is compliant to the Linux for zSeries ABI. This allows GCC to generate 64-bit instructions. This is the default for s390x build targets.

-mbackchain: Specifying this option tells GCC to generate code that maintains an explicit backchain within the stack frame. This backchain points to the caller's frame, and is currently needed to allow debugging. This is the default.

- mdebug: Specifying this option tells GCC to print additional debug information when compiling.
- mhard-float: Specifying this option tells GCC to use the hardware floating-point instructions and registers for floating-point operations. GCC generates the appropriate IEEE floating-point instructions. This is the default.
- mmvcle: Specifying this option tells GCC to generate code using the mvcle instruction to perform block moves.
- mno-backchain: Specifying this option tells GCC not to generate code that maintains an explicit backchain within the stack frame that points to the caller's frame. Not maintaining a backchain prevents debugging.
- mno-debug: Specifying this option tells GCC not to print additional debug information when compiling. This is the default.
- mno-mvcle: Specifying this option tells GCC not to generate code using the mvcle instruction to perform block moves, but to use an mvc loop instead. This is the default.
- mno-small-exec: Specifying this option tells GCC not to generate code using the bras instruction to do subroutine calls. This is the default, causing programs compiled with GCC to use the basr instruction instead, which does not have a 64K limitation.
- msmall-exec: Specifying this option tells GCC to generate code using the bras instruction to do subroutine calls. This only works reliably if the total executable size does not exceed 64K.
- msoft-float: Specifying this option tells GCC not to use the hardware floating-point instructions and registers for floating-point operations. Functions in libgcc.a will be used to perform floating-point operations.

SH Options

SuperH (SH) processors from Hitachi and others are powerful and extremely popular processors for use in embedded systems. The SH-1 and SH-2 processors are 16-bit processors, the SH-3 and SH-4 are 32-bit processors, and the new SH-5 is a fast 64-bit processor.

GCC options available when compiling code for systems using SH processors are the following:

- m1: Specifying this option tells GCC to generate code for the SH1 processor.
- m2: Specifying this option tells GCC to generate code for the SH2 processor.
- m3: Specifying this option tells GCC to generate code for the SH3 processor.
- m3e: Specifying this option tells GCC to generate code for the SH3e processor.
- m4-nofpu: Specifying this option tells GCC to generate code for SH4 processors without a floating-point unit.
- m4-single-only: Specifying this option tells GCC to generate code for SH4 processors with a floating-point unit that only supports single precision arithmetic.
- m4-single: Specifying this option tells GCC to generate code for SH4 processors, assuming the floating-point unit is in single precision mode by default.
- m4: Specifying this option tells GCC to generate code for the SH4 processor.
- mb: Specifying this option tells GCC to compile code for an SH processor in big endian mode.
- mbigtable: Specifying this option tells GCC to use 32-bit offsets in switch tables. The default is to use 16-bit offsets.
- mdalign: Specifying this option tells GCC to align doubles at 64-bit boundaries. This changes the calling conventions, and therefore some functions from the standard C library will not work unless you also recompile the standard C library with the -mdalign option.
- mfmove: Specifying this option tells GCC to enable the use of the fmove instruction.
- mhitachi: Specifying this option tells GCC to comply with the SH calling conventions defined by Hitachi.
- mieee: Specifying this option tells GCC to generate IEEE-compliant floating-point code.
- misize: Specifying this option tells GCC to include instruction size and location in the assembly code.
- ml: Specifying this option tells GCC to compile code for an SH processor in little endian mode.
- nomacsave: Specifying this option tells GCC to mark the MAC register as call-clobbered, even if the -mhitachi option is also used.

- mpadstruct: This is a deprecated option that pads structures to multiple of 4 bytes, which is incompatible with the SH ABI.
- mprefforgot: Specifying this option tells GCC to emit function calls using the global offset table instead of the procedure linkage table when generating position-independent code.
- mrelax: Specifying this option tells GCC to shorten some address references at link time, when possible. Specifying this option passes the -relax option to the linker.
- mspace: Specifying this option tells GCC to optimize for size instead of speed. This option is implied by the -Os optimization option.
- musermode: Specifying this option tells GCC to generate a library function call that invalidates instruction cache entries after fixing up a trampoline. The function call does not assume that it can write to the whole memory address space. This is the default when GCC was built for the sh-*-linux* target.

SPARC Options

The SPARC is a fast 32-bit processor architecture originally developed by Sun Microsystems and used in all of their workstations, single-board computers (SBCs), and other embedded hardware since the late 1980s. The SPARC processor has also been licensed to a number of other workstation and embedded hardware vendors. The latest generation of SPARC processors, the UltraSPARC family, are 64-bit processors.

GCC options available when compiling code for systems using SPARC processors are the following:

- mapp-reg: Specifying this option tells GCC to be fully SVR4 ABI compliant at the cost of some performance loss. Libraries and system software should be compiled with this option to maximize compatibility.
- mcpu=*CPU-type*: Specifying this option tells GCC to set the instruction set, register set, and instruction scheduling parameters for machine type *CPU-type*. Supported values for *CPU-type* are v7, cypress, v8, supersparc, sparclite, hypersparc, sparclite86x, f930, f934, sparclet, tsc701, v9, and ultrasparc. Default instruction scheduling parameters are used for values that select an architecture and not an implementation. These are v7, v8, sparclite, sparclet, and v9.

Table B-2 shows each supported architecture and their supported implementations.

Table B-2. SPARC Architectures and Associated CPU-type Values

Values	Architecture
v7	cypress
v8	supersparc, hypersparc
sparclite	f930, f934, sparclite86x
sparclet	tsc701
v9	ultrasparc

-mcypress: Specifying this option tells GCC to optimize code for the Cypress CY7C602 chip, as used in the SparcStation/SparcServer 3xx series. This is also appropriate for the older SparcStation 1, 2, IPX, and so on. This is the default. This option is deprecated—the more general **-mCPU=CPU-type** option should be used instead.

-mfaster structs: Specifying this option tells GCC to assume that structures should have 8-byte alignment. This enables the use of pairs of ldd and std instructions for copies in structure assignment, instead of twice as many ld and st pairs. However, the use of this changed alignment directly violates the Sparc ABI and is therefore intended only for use on targets where developers acknowledge that their resulting code will not be directly in line with the rules of the ABI.

-mflat: Specifying this option tells GCC not to generate save/restore instructions and instead use a “flat,” or single-register window, calling convention. This model uses %i7 as the frame pointer and is compatible with code that does not use this calling convention. Regardless of the calling conventions used, the local registers and the input registers (0–5) are still treated as “call-saved” registers and will be saved on the stack as necessary.

-mhard-float | -mfpu: Specifying this option tells GCC to generate output containing floating-point instructions. This is the default.

-mhard-quad-float: Specifying this option tells GCC to generate output that contains quad-word (`long double`) floating-point instructions.

NOTE No current SPARC implementations provide hardware support for the quad-word floating-point instructions. They all invoke a trap handler for one of these instructions, where the trap handler then emulates the effect of the instruction. Because of the overhead of the trap handler, this is much slower than calling the ABI library routines. For this reason, the `-msoft-quad-float` option is the default.

-mno-app-reg: Specifying this option tells GCC to generate output using the global registers 2 through 4, which the SPARC SVR4 ABI reserves for applications. This is the default.

-mno-faster-structs: Specifying this option tells GCC not to make any assumptions about structure alignment, and to use the `ld` and `st` instructions when making copies during structure assignment.

-mno-flat: Specifying this option tells GCC to use save/restore instructions as its calling convention (except for leaf functions). This option is the default.

-mno-unaligned-doubles: Specifying this option tells GCC to assume that doubles have 8-byte alignment. This is the default.

-msoft-float | -mno-fpu: Specifying this option tells GCC to generate output that contains library calls for floating-point operations. These libraries are not provided as part of GCC (except for the embedded GCC build targets `sparc-*-aout` and `sparclite-*-*-*`), but are normally found on the target system, and should be resolved by the C loader on the target machine if they are available. When using this option and cross-compiling, you must provide suitable libraries containing at least function stubs on the host system, or these function calls will be identified as unresolved.

TIP Specifying this option also changes the calling convention used in the output file. You must therefore compile all of the modules of your program with this option, including any libraries that you reference. You must also compile `libgcc.a`, the library that comes with GCC, with this option in order to be able to use this option.

-msoft-quad-float: Specifying this option tells GCC to generate output containing library calls for quad-word (`long double`) floating-point instructions. The functions called are those specified in the SPARC ABI. This is the default.

-msparclite: Specifying this option tells GCC to generate code for SPARClite processors. This adds the integer multiply, integer divide step and scan (ffs) instructions that exist in SPARClite but not in SPARC v7. This option is deprecated; the more general `-mCPU=CPU-type` option should be used instead.

-msupersparc: Specifying this option tells GCC to optimize code for the SuperSparc CPU, as used in the SparcStation 10, 1000, and 2000 series. This option also enables use of the full SPARC v8 instruction set. This option is deprecated; the more general `-mCPU=CPU-type` option should be used instead.

-mtune=CPU-type: Specifying this option tells GCC to set the instruction scheduling parameters for machine type *CPU-type*, but not to set the instruction set or register set as the option `-mcpu=CPU-type` would. The same values for `-mcpu=CPU-type` can be used for `-mtune=CPU-type`, but the only useful values are those that select a particular CPU implementation: cypress, supersparc, hypersparc, f930, f934, sparclite86x, tsc701, and ultrasparc.

-munaligned-doubles: Specifying this option causes GCC not to assume that doubles are 8-byte aligned. GCC assumes that doubles have 8-byte alignment only if they are contained in another type, or if they have an absolute address. Otherwise, it assumes they have 4-byte alignment. Specifying this option avoids some rare compatibility problems with code generated by other compilers, but results in a performance loss, especially for floating-point code.

-mv8: Specifying this option tells GCC to generate code for SPARC v8 processors. The only difference from v7 code (the default) is that the compiler emits the integer multiply and integer divide instructions that exist in SPARC v8 but not in SPARC v7. This option is deprecated; the more general `-mCPU=CPU-type` option should be used instead.

SPARClet Processor Option

The SPARClet is a SPARC processor designed for use in embedded or SBC environments. The SPARClet processor uses an embedded parallel RISC architecture and combines an expanded SPARC instruction set with integrated support for digital signal processing.

You can use all of the standard SPARC GCC options when compiling code for SPARClet processors. GCC also provides some options that are specific for use when compiling code for this processor. GCC options specific to compiling code for SPARClet processors are the following:

-mbroken-saverestore: Specifying this option tells GCC to generate code that does not use nontrivial forms of the save and restore instructions. Early versions of the SPARClet processor do not correctly handle save and restore instructions used with arguments, but do correctly handle them when used without arguments. A save instruction used without arguments increments the current window pointer but does not allocate a new stack frame. It is assumed that the window overflow trap handler and interrupt handlers will properly handle this case.

-mlittle-endian: Specifying this option tells GCC to generate code for a processor running in little endian mode.

-mlive-go: Specifying this option tells GCC to treat register %g0 as a normal register. GCC will clobber the register as necessary, but will not assume it always reads as 0.

64-Bit SPARC Options

UltraSPARC is the first 64-bit SPARC processor. You can use all of the standard SPARC GCC options when compiling code for UltraSPARC processors. GCC also provides some options that are specific for use when compiling code for this processor.

GCC options specific to compiling code for UltraSPARC processors are the following:

-m32: Specifying this option tells GCC to generate code for a 32-bit environment. The 32-bit environment sets int, long, and pointer to 32 bits.

-m64: Specifying this option tells GCC to generate code for a 64-bit environment. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits.

-mcmodel=embmedany: Specifying this option tells GCC to generate code for the Medium/Anywhere code model for embedded systems, which assumes a 32-bit text and a 32-bit data segment, both starting anywhere (determined at link time). Register %g4 points to the base of the data segment. Pointers are still 64 bits. Programs are statically linked; PIC is not supported.

-mcmodel=medany: Specifying this option tells GCC to generate code for the Medium/Anywhere code model, where the program may be linked anywhere in the address space, the text segment must be less than 2GB, and data segment must be within 2GB of the text segment.

Pointers are 64 bits.

-mmodel=medlow: Specifying this option tells GCC to generate code for the Medium/Low code model, where a program must be linked in the low 32 bits of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked.

-mmodel=medmid: Specifying this option tells GCC to generate code for the Medium/Middle code model, where the program must be linked in the low 44 bits of the address space, the text segment must be less than 2GB, and the data segment must be within 2GB of the text segment. Pointers are 64 bits.

-mlittle-endian: Specifying this option tells GCC to generate code for a processor running in little endian mode.

-mno-stack-bias: Specifying this option tells GCC to assume that no offset is used when making stack frame references.

-mstack-bias: Specifying this option tells GCC to assume that the stack pointer and frame pointer (if present) are offset by -2047, which must be added back when making stack frame references.

System V Options

System V Release 4, or SVR4, was a landmark release of the Unix operating system from AT&T, and is the conceptual foundation for Unix-like operating systems such as Solaris and Linux. Alas, poor BSD, I knew thee well . . .

GCC options specific to compiling code for systems running SVR4 are the following:

-G: Specifying this option tells GCC to create a shared object. This option is deprecated, as it is easily confused with other **-G** options. You should use the more comprehensible **-symbolic** or **-shared** options instead.

-Qn: Specifying this option tells GCC not to add **.ident** directives identifying tool versions in the output assembly file. This is the default.

-Qy: Specifying this option tells GCC to identify the versions of each tool used by the compiler, by adding a **.ident** assembler directive in the assembler output.

-Ym,DIR: Specifying this option tells GCC to look in the directory DIR to find the m4 preprocessor. The assembler uses this option.

-YP,DIRS: Specifying this option tells GCC to search the directories DIRS, and no others, for libraries specified with **-l**.

TMS320C3x/C4x Options

The TMS320C3x and TMS320C4x processors are digital signal processors from Texas Instruments.

GCC options specific to compiling code for systems running TMS320C3x and TMS320C4x processors are the following:

-mbig-memory | -mbig: Specifying this option tells GCC to generate code for the big memory model. Using the big memory model makes no assumptions about code data size, and requires reloading of the DP register for every direct memory access. This is the default.

-mbk: Specifying this option tells GCC to allow allocation of general integer operands into the block count register BK.

-mcpu=CPU-type: Specifying this option tells GCC to set the instruction set, register set, and instruction scheduling parameters for machine type *CPU-type*. Supported values for *CPU-type* are c30, c31, c32, c40, and c44. The default is c40, which generates code for the TMS320C40.

-mdb: Specifying this option tells GCC to enable generating code that uses the decrement and branch, DBcond(D), instructions. This is the default for TMS320C4x processors.

NOTE *GCC will try to reverse a loop so that it can utilize the decrement and branch instruction, but will give up if there is more than one memory reference in the loop. Thus a loop where the loop counter is decremented can generate slightly more efficient code, in cases where the RPTB instruction cannot be utilized.*

-mdp-isr-reload | -mparanoid: Specifying either of these options tells GCC to force the DP register to be saved on entry to an interrupt service routine (ISR), reloaded to point to the data section, and restored on exit from the ISR. This should not be necessary unless someone has violated the small memory model by modifying the DP register, such as within an object library.

-mfast-fix: Specifying this option tells GCC to accept the results of the C3x/C4x FIX instruction which, when converting a floating-point value to an integer value, chooses the nearest integer less than or equal to the floating-point value rather than to the nearest integer. Thus if the floating-point number is negative, the result will be incorrectly truncated and additional code will be necessary to detect and correct this case.

-mloop-unsigned: Specifying this option tells GCC to use an unsigned iteration count. This limits loop iterations to $2^{31} + 1$, since these instructions test if the iteration count is negative in order to terminate a loop.

-mmemparm: Specifying this option tells GCC to generate code that uses the stack for passing arguments to functions.

-mmpyi: Specifying this option when compiling code for TMS320C3x processors tells GCC to use the 24-bit MPYI instruction for integer multiplies instead of a library call to guarantee 32-bit results. Note that if one of the operands is a constant, then the multiplication will be performed using shifts and adds.

-mno-bk: Specifying this option tells GCC not to allow allocation of general integer operands into the block count register BK.

-mno-db: Specifying this option tells GCC not to generate code that uses the decrement and branch, DBcond(D), instructions. This is the default for TMS320C3x processors.

-mno-fast-fix: Specifying this option tells GCC to generate additional code to correct the results of the C3x/C4x FIX instruction which, when converting a floating-point value to an integer value, chooses the nearest integer less than or equal to the floating-point value rather than to the nearest integer. Thus if the floating-point number is negative, the result will be incorrectly truncated. Specifying this option generates the additional code necessary to detect and correct this case.

-mno-loop-unsigned: Specifying this option tells GCC not to use an unsigned iteration count, which might artificially limit loop iterations to $2^{31} + 1$, since these instructions test if the iteration count is negative in order to terminate a loop.

-mno-mpyi: Specifying this option tells GCC to use a library call for integer multiplies. When compiling code for the TMS320C3x processor, squaring operations are performed inline instead of through a library call.

-mno-parallel-insns: Specifying this option tells GCC not to generate parallel instructions.

-mno-parallel-mpy: Specifying this option tells GCC not to generate MPY | ADD and MPY | SUB parallel instructions. This generally minimizes code size, though the lack of parallelism may negatively impact performance.

-mno-rptb: Specifying this option tells GCC not to generate repeat block sequences using the RPTB instruction for zero overhead looping.

-mno-rpts: Specifying this option tells GCC not to use the single-instruction repeat instruction RPTS. This is the default, because interrupts are disabled by the RPTS instruction.

-mparallel-insns: Specifying this option tells GCC to enable generating parallel instructions. This option is implied when the -O2 optimization option is specified.

-mparallel-mpy: Specifying this option tells GCC to generate MPY || ADD and MPY || SUB parallel instructions if the -mparallel-insns option is also specified. These instructions have tight register constraints that can increase code size for large functions.

-mregparm: Specifying this option tells GCC to generate code that uses registers (whenever possible) for passing arguments to functions. This is the default.

-mrptb: Specifying this option tells GCC to enable the generation of repeat block sequences using the RPTB instruction for zero overhead looping. The RPTB construct is only used for innermost loops that do not call functions or jump across the loop boundaries. There is no advantage to having nested RPTB loops due to the overhead required to save and restore the RC, RS, and RE registers. This is option enabled by default with the -O2 optimization option.

-mrpts=*count*: Specifying this option tells GCC to enable the use of the single instruction repeat instruction RPTS. If a repeat block contains a single instruction, and the loop count can be guaranteed to be less than the value *count*, GCC will emit a RPTS instruction instead of an RPTB instruction. If no value is specified, then a RPTS will be emitted even if the loop count cannot be determined at compile time. Note that the repeated instruction following the RPTS instruction does not have to be reloaded from memory during each iteration, thus freeing up the CPU buses for operands.

-msmall-memory | -msmall: Specifying this option tells GCC to generate code for the small memory model. The small memory model assumes that all data fits into one 64K word page. At runtime, when using the small memory model, the data page (DP) register must be set to point to the 64K page containing the bss and data program sections.

-mti: Specifying this option tells GCC to try to emit an assembler syntax that the TI assembler (asm30) is happy with. This also enforces compatibility with the API employed by the TI C3x C compiler. For example, long doubles are passed as structures rather than in floating-point registers.

V850 Options

NEC's V850 family of 32-bit RISC microcontrollers is designed for embedded real-time applications such as motor control, process control, industrial measuring equipment, automotive systems, and so on.

GCC options specific to compiling code for systems running V850 processors are the following:

-mbig-switch: Specifying this option tells GCC to generate code suitable for big switch tables. You should only use this option if the assembler or linker complains about out-of-range branches within a switch table.

-mep: Specifying this option tells GCC to optimize basic blocks that use the same index pointer four or more times to copy pointers into the ep register, using the shorter sld and sst instructions instead. This option is on by default if you specify any of the optimization options.

-mlong-calls: Specifying this option tells GCC to treat all calls as being far. If calls are assumed to be far, GCC will always load the function's address into a register, and make an indirect call through the pointer.

-mno-ep: Specifying this option tells GCC not to optimize basic blocks that use the same index pointer four or more times to copy pointers into the ep register.

-mno-long-calls: Specifying this option tells GCC to treat all calls as being near, requiring no special handling.

-mno-prolog-function: Specifying this option tells GCC not to use external functions to save and restore registers at the prolog and epilog of a function.

-mprolog-function: Specifying this option tells GCC to use external functions to save and restore registers at the prolog and epilog of a function. These external functions are slower, but use less code space if more than one function saves the same number of registers. This option is enabled by default if you specify any of the optimization options.

-msda=n: Specifying this option tells GCC to put static or global variables whose size is *n* bytes or less into the small data area that register gp points to. The small data area can hold up to 64 kilobytes.

-mspace: Specifying this option tells GCC to try to make the code as small as possible by activating the -mep and -mprolog-function options.

-mtda=n: Specifying this option tells GCC to put static or global variables whose size is *n* bytes or less into the tiny data area that register ep points to. The tiny data area can hold up to 256 bytes in total (128 bytes for byte references).

-mv850: Specifying this option tells GCC that the target processor is the V850.

-mzda=n: Specifying this option tells GCC to put static or global variables whose size is *n* bytes or less into the first 32 kilobytes of memory.

VAX Options

VAX was Digital Equipment Corporation's (DEC) workstations and mini-computers for years, and shipped with a quaint operating system called VMS.

GCC options available when compiling code for VAX systems are the following:

-mg: Specifying this option tells GCC to generate code for g-format floating-point numbers instead of d-format floating-point numbers.

-mgnu: Specifying this option tells GCC to generate all possible jump instructions. This option assumes that you will assemble your code using the GNU assembler.

-munix: Specifying this option tells GCC not to generate certain jump instructions (aobeq and so on) that the Unix assembler for the VAX cannot handle across long ranges.

Xstormy16 Options

Sanyo's Xstormy16 processor is designed for memory-constrained applications, and is often used in home appliances and audio systems.

The sole GCC option available when compiling code for XStormy16 processors is the following:

-msim: Specifying this option tells GCC to choose startup files and linker scripts that are suitable for an XStormy16 simulator.

Xtensa Options

The Xtensa architecture is designed to support many different configurations—there is no such thing as a generic Xtensa processor. Each instance of the Xtensa processor architecture is uniquely tuned by the system designer to ideally fit the

SOC's targeted application, by using the Xtensa Processor Generator or by selecting from a broad selection of predefined standard RISC microprocessor features,

GCC's default options can be set to match a particular Xtensa configuration by copying a configuration file into the GCC sources when building GCC.

GCC options available to override the default values specified in the configuration file when compiling code for Xtensa processors are the following:

-mbig-endian: Specifying this option tells GCC to generate code using big endian byte ordering.

-mbooleans: Specifying this option tells GCC to enable support for the Boolean register file used by Xtensa coprocessors. This is not typically useful by itself but may be required for other options that make use of the Boolean registers (such as the floating-point options).

-mdensity: Specifying this option tells GCC to enable the use of the optional Xtensa code density instructions.

-mfused-madd: Specifying this option tells GCC to enable the use of fused multiply/add and multiply/subtract instructions in the floating-point option. This has no effect if the floating-point option is not also enabled. This option should not be used when strict IEEE 754-compliant results are required, since the fused multiply/add and multiply/subtract instructions do not round the intermediate result, and therefore produce results with more precision than is specified by the IEEE standard.

-mhard-float: Specifying this option tells GCC to enable the use of the floating-point option. GCC generates floating-point instructions for 32-bit float operations. 64-bit double operations are always emulated with calls to library functions.

-mlittle-endian: Specifying this option tells GCC to generate code using little endian byte ordering.

-mlongcalls: Specifying this option tells GCC to instruct the assembler to translate direct calls to indirect calls unless it can determine that the target of a direct call is in the range allowed by the call instruction. This translation typically occurs for calls to functions in other source files. This option should be used in programs where the call target can potentially be out of range. This option is implemented in the assembler, not the compiler, so the assembly code generated by GCC will still show direct call instructions; you will have to examine disassembled object code in order to see the actual instructions. This option also causes the assembler to use an indirect call for every cross-file call, not just those that really will be out of range.

-mmac16: Specifying this option tells GCC to enable the use of the Xtensa MAC16 option. GCC will generate MAC16 instructions from standard C code, with the limitation that it will use neither the MR register file nor any instruction that operates on the MR registers.

-mminmax: Specifying this option tells GCC to enable the use of the optional minimum and maximum value instructions.

-mmul16: Specifying this option tells GCC to enable the use of the 16-bit integer multiplier option. GCC will generate 16-bit multiply instructions for multiplications of 16 bits or smaller in standard C code.

-mmul32: Specifying this option tells GCC to enable the use of the 32-bit integer multiplier option. GCC will generate 32-bit multiply instructions for multiplications of 32 bits or smaller in standard C code.

-mno-booleans: Specifying this option tells GCC to disable support for the Boolean register file used by Xtensa coprocessors.

-mno-density: Specifying this option tells GCC to disable the use of the optional Xtensa code density instructions.

-mno-longcalls: Specifying this option tells GCC not to translate direct calls to indirect calls. This is the default. See the description of the -mlongcalls option for implementation details.

-mno-mac16: Specifying this option tells GCC to disable the use of the Xtensa MAC16 option. GCC will translate 16-bit multiply/accumulate operations to a combination of core instructions and library calls, depending on whether any other multiplier options are enabled.

-mno-fused-madd: Specifying this option tells GCC to disable the use of fused multiply/add and multiply/subtract instructions in the floating-point option. Disabling fused multiply/add and multiply/subtract instructions forces the compiler to use separate instructions for the multiply and add/subtract operations. This may be desirable in some cases where strict IEEE 754-compliant results are required, since the fused multiply/add and multiply/subtract instructions do not round the intermediate result, thereby producing results with more precision than is specified by the IEEE standard. Disabling fused multiply/add and multiply/subtract instructions also ensures that the program output is not sensitive to the compiler's ability to combine multiply and add/subtract operations.

-mno-minmax: Specifying this option tells GCC to disable the use of the optional minimum and maximum value instructions.

-mno-mul16: Specifying this option tells GCC to disable the use of the 16-bit integer multiplier option. GCC will either use 32-bit multiply or MAC16 instructions if they are available or generate library calls to perform the multiply operations using shifts and adds.

-mno-mul32: Specifying this option tells GCC to disable the use of the 32-bit integer multiplier option. GCC will generate library calls to perform the multiply operations using either shifts and adds or 16-bit multiply instructions if they are available.

-mno-nsa: Specifying this option tells GCC to disable the use of the optional normalization shift amount (NSA) instructions to implement the built-in `ffs` function.

-mno-serialize-volatile: Specifying this option tells GCC not to use MEMW instructions before volatile memory references in order to guarantee sequential consistency, resulting in decreased code size.

-mno-sext: Specifying this option tells GCC to disable the use of the optional sign extend (SEXT) instruction.

-mno-target-align: Specifying this option tells GCC to not have the assembler automatically align instructions, increasing code density. Specifying this option does not affect the treatment of auto-aligned instructions like LOOP, which the assembler will always align, either by widening density instructions or by inserting NOOP instructions.

-mno-text-section-literals: Specifying this option tells GCC to place literals in a separate section in the output file. This allows the literal pool to be placed in a data RAM/ROM, and also allows the linker to combine literal pools from separate object files to remove redundant literals and improve code size. This is the default.

-mnsa: Specifying this option tells GCC to enable the use of the optional NSA instructions to implement the built-in `ffs` function.

-mserialize-volatile: Specifying this option tells GCC to insert MEMW instructions before volatile memory references in order to guarantee sequential consistency. This is the default.

-msext: Specifying this option tells GCC to enable the use of the optional SEXT instruction.

-msoft-float: Specifying this option tells GCC to disable use of the floating-point option. GCC generates library calls to emulate 32-bit floating-point operations using integer instructions. 64-bit double operations are always emulated with calls to library functions.

-mtarget-align: Specifying this option tells GCC to instruct the assembler to automatically align instructions in order to reduce branch penalties at the expense of some code density. The assembler attempts to widen density instructions to align branch targets and the instruction's following call instructions. If there are not enough preceding safe density instructions to align a target, no widening will be performed. This is the default.

-mtext-section-literals: Specifying this option tells GCC to intersperse literals in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files.

Index

See Appendix B for a comprehensive list of machine and processor-specific options for GCC.

Symbols and Numbers

- ### command-line option, description of, 284
- #####, appearance in profiling output, 159
- \$ preprocessor option, description of, 338
- %% predefined substitution spec, description of, 107
- %1 spec processing instruction, description of, 108
- %2 spec processing instruction, description of, 108
- * label notation, meaning of, 276
- @-commands, location of, 266
- 0 an-01, optimizations enabled with, 139
- 02 optimization, optimizations enabled with, 140–141
 - _ (underscores), using with `typeof`, 115
 - `_cyg_profile_func_enter()` function, example of, 182–183
- 00 command-line option, description of, 315
- 0|-01 command-line option, description of, 315
- 02 command-line option, description of, 316
- 03 command-line option, description of, 316
- 64-bit SPARC options, overview of, 486–487
- \ (backslash), significance of, 9
- , (comma) search function in GNU Info, description of, 275
- 0 (parentheses), appearance in autoconf macros, 204
- ? (question mark) search function in GNU Info, description of, 275

A

- A and a arguments for `dump` option, descriptions of, 97
- a command-line option in `gprof`, 168
 - advisory about, 96

- %a spec processing instruction, description of, 108
- A-* preprocessor options, descriptions of, 338
- A[*symbol-specification*] command-line option in `gprof`, description of, 169, 178–182
- ABI (application binary interface), availability of, 27
- `abort()` library function, relationship to function attributes, 125
- AC_* files in autoconf, descriptions of, 203–204
- aclocal script, using with autoconf and automake, 212, 214
- Ada, downloading source for, 5–7
- `addr2line()` function, example of, 183
- addresses, mapping to function names, 183
- Agent newsreader, downloading, 349
- AIX systems, troubleshooting problems on, 254–255
- aliases, troubleshooting, 249
- aligned (*n*) attribute, example of, 128
- all edges coverage, role in test coverage, 149
- all Makefile produced by automake, description of, 208
- all makefile target, description of, 19
- Allegro DJGPP component, description of, 51
- `alloca()` function, using with arrays of variable length, 118
- all-static mode for Libtool, working in, 237
- Alpha options, overview of, 405–411
- Alpha/VMS options, overview of, 411
- Alt versus Esc key in GNU Info, advisory about, 269
- AMD x86-64 options, overview of, 412
- AMD29K options, overview of, 413–415
- annotated source code, displaying, 178–182
- anonymous FTP, using to download files for Glibc from, 379–380
- ANSI C, concerns associated with, 69–70

- ansi command-line option, description of, 70, 73, 284, 338
- a.out directory, significance of, 64–65
- applications, configuring with autoconf and automake, 212
- Apredicate* preprocessor option, description of, 338
- apropos=*word* command-line option in GNU Info, description of, 278
- ARC options, overview of, 415–416
- arguments, passing to assembler, linker, and preprocessor, 252
- arithmetic on void and function pointers, relationship to C and extensions, 120
- ARM #pragmas, features of, 133
- ARM options, overview of, 416–422
- array indexes, specifying with designated initializers, 120–121
- arrays
 - using with `typeof`, 114–115
 - of variable length, 117–119
 - zero-length arrays used with C and extensions, 115–117
- asm* built-in spec strings, descriptions of, 106
- assemblers
 - passing arguments to, 252
 - passing options to, 86
 - running preprocessed files through, 67
 - using IBM assemblers on AIX systems, 254
- _attribute_keyword*, using, 123–124, 126–127
- attributes
 - declaring function attributes, 123–127
 - specifying variable attributes, 127–130
- AUTHORS auxiliary file in automake
 - description of, 209
 - example of, 215
- autoconf program, 196–197
 - building and installing, 192–197
 - creating `configure.ac` files for, 201–206
 - development of, 188–189
 - downloading, 6
 - files required for, 200–201
 - generating configuration scripts with, 212
 - macros in, 203
 - mailing lists for, 193
 - obtaining current information about, 218
 - obtaining online, 193–194
 - running, 211–218
- upgrade versus replacing, 191–192
- using Libtool with, 239–243
- autoheader utility in autoconf package
 - description of, 196
 - using, 212
- autom4te utility in autoconf package, description of, 197
- automake program
 - auxiliary files in, 209
 - creating `Makefile.am` files for, 207–211
 - downloading, 6
 - features of, 190–191
 - files required for, 200–201
 - mailing lists for, 198
 - Makefiles produced by, 208
 - obtaining and installing, 197–200
 - obtaining current information about, 218
 - primaries in, 207
 - process of, 210–211
 - running, 211–218
 - upgrade versus replacing, 191–192
 - using Libtool with, 239–243
- autoreconf utility in autoconf package, description of, 197
- autoscan utility in autoconf package
 - description of, 197
 - using, 201–202
- autoupdate utility in autoconf package, description of, 197
- aux-info* command-line option, description of, 70–71, 284
- avoid-version* mode for Libtool, working in, 237
- AVR options, overview of, 422–423

B

- B** and **b** arguments for `dump` option, descriptions of, 97
- b** GNU Info navigation command, description of, 268, 273
- b** command-line option
 - in `gprof`, 169
 - in `gcov`, 154, 159–161, 177–178
- %B** and **% b** predefined substitution specs, descriptions of, 107
- B** *prefix* option, description of, 79
- backslash (\), significance of, 9
- bash: `gcc: Permission denied` message, generating, 247
- basic blocks, relationship to optimization, 136
- .bb files
 - creation of, 163
 - example of, 156
 - role in test coverage analysis, 153

.bbg files, creation of, 163
 Berkeley DB NSS (name Service Switch)
 Glibc add-on, features of, 373
 --bindir configuration option,
 description of, 12
 binutils package, contents of, 165
 Bison, downloading, 6
 block reordering optimization, features
 of, 24
 -bmachine command-line option,
 description of, 284
 bootstrap process, duration of, 17
 bounds-checking mode, availability of, 29
 -Bprefix command-line option,
 description of, 285
 branch counts, showing in
 fibonacci.c.gcov file, 161–162
 branch coverage, role in test coverage,
 149
 branch percentages, showing in
 fibonacci.c.gcov file, 160–161
 BSD systems
 Glibc support for, 364
 troubleshooting code profiling on,
 255
 bubblestrap makefile target, description
 of, 19
 bug reports, submitting with gccbug
 scripts, 31
 bugs and misfeatures, coping with,
 245–247
 build directory, making current
 directory, 9
 build problems, resolving, 262–263
 build process
 optimizing, 17
 previewing, 2–3
 build system versus host and target
 systems, 9–10
 builds, testing, 20–23
 _builtin_expect optimization, features
 of, 26
 BusyBox, resolving Glibc upgrade
 problems with, 394–396

C

-c
 command-line option, 285
 command-line option in gprof, 169
 option for gcov program, 154
 output option, 64, 66
 C and c arguments for dump option,
 descriptions of, 97
 %c and C spec processing instructions,
 descriptions of, 108
 .c and .C suffixes, operations associated
 with, 63

C and extensions. *See* GCC's C and
 extensions
 C argument for dump option,
 description of, 99
 C dialects, compiling, 69–74
 C language
 embedding decisions with, 151
 options for, 344
 C library
 in DJGPP, 48–49
 installing Glibc as, 391–393
 -C preprocessor option, description of,
 338
 -C[*symbol-specification*] command-line
 option in gprof, description of, 169
 C_INCLUDE_PATH environment
 variable, purpose of, 102
 C++ code, compiling with GCC, 74–76
 C++ compilers, testing, 22
 C++ language
 downloading source for, 5–7
 embedding decisions with, 151
 options in, 344
 C99 standard
 explanation of, 69–70
 selecting for C dialect options, 73–74
 using flexible array members with,
 116
 calc_fib() function, displaying
 annotated source code for, 178–179
 calc_fib.c function sample code, 156
 call graphs
 identifying sequences of function
 calls with, 147
 providing with gprof applications,
 164
 providing with gprof code profiler,
 177
 Canadian compiler, purpose of, 10
 Cannot execute binary file,
 troubleshooting, 249–250
 case ranges, relationship to C and
 extensions, 123
 .cc suffix, operation associated with, 63
 cci* built-in spec strings, descriptions
 of, 106
 ChangeLog auxiliary file in automake
 description of, 209
 example of, 215
 char type, using with -signed and -
 unsigned C options, 71–72
 check Makefile
 producing with automake, 208
 target for, 19
 Chill compiler, treatment of, 37
 clean Makefile
 producing with automake, 208
 target for, 19

clean mode for Libtool, working in, 233
 cleanstrap makefile target, description of, 19
 Clipper options, overview of, 424
 COBOL for GCC mailing lists, description of, 357
 code continuation, indicating with backslash (\), 9
 code coverage analysis, explanation of, 148
 code generation options, list of, 344
 code hoisting, explanation of, 25
 code profile analysis, compiling code for, 166–167
 code profiling

- common errors in, 183–185
- explanation of, 148
- troubleshooting on BSD systems, 255
- using gcc for, 182–185
- using gprof for, 163–164

 code unification, explanation of, 25
 comma (.) search function in GNU Info, description of, 275
 command line, using Libtool from, 232–239
 command multipliers, using with GNU Info, 280
`%command` spec file directive, description of, 104
 command-line options. *See* GCC command-line options
 common subexpression, explanation and example of, 136–137
 compare makefile target, description of, 19
 compile mode for Libtool, working in, 233–234
 compiler optimization theory

- eliminating redundant computations, 136–138
- overview of, 135–138

 The Compiler Resources Page Web address, 358–359
 COMPILER_PATH environment variable, purpose of, 101
 compilers, types of, 10
 compiling

- C dialects, 69–74
- example of, 64
- source files, 65
- stages of, 62–63

 computed goto statements, using with labels as values, 111–112
 config.* files and options in Libtool, descriptions of, 231, 233
 configuration options, overview of, 11–15

configure scripts

- advisory about, 11
- executing after generating with autoconf, 205
- generating with autoconf, 205, 212
- running, 218–219
- using with autoconf and automake programs, 191–192
- using with autoconf program, 194–195
- using with automake, 199

 configure.ac files

- creating for autoconf, 201–206, 213
- integrating Libtool with, 240
- use by autoconf program, 189
- using with Fibonacci application, 240

 control flow analysis, relationship to optimization, 135–136
 Convex options, overview of, 424–425
 COPYING auxiliary file in automake, description of, 209
 copy-propagation transformations, eliminating redundant code with, 137–138
 CPATH environment variable, purpose of, 102
 CPLUS_INCLUDE_PATH environment variable, purpose of, 102
 cpp built-in spec string, description of, 106
 cpp C preprocessor, enhancements made to, 28
`.cpp` suffix, operation associated with, 63
 CRIS options, overview of, 425–428
 crossback compiler, purpose of, 10
 cross-compilers, purpose of, 10, 249
 csh (C shell), updating PATH environment variables for, 248
 Ctrl-*

- GNU Info navigation commands, 267–268
- search functions in GNU Info, 275

 cube() function, example of, 126
 CVS snapshots, requirements for building GCC from, 6
`.cxx` suffix, operation associated with, 63
 Cygwin

- downloading and installing, 39–45
- overview of, 39
- specifying Internet connection for, 42
- using in DOS and Windows, 45–47

D

D and d arguments for dump option, descriptions of, 97
 -D command-line option in gprof, description of, 170

%d predefined substitution spec, description of, 107
-d[mod] debugging option, description of, 95
-d[num] command-line option in gprof, description of, 169
D30V options, overview of, 428–429
.da files, creation of, 163
Darwin #pragmas, features of, 133–134
data flow analysis, relationship to optimization, 136
DATA primary in automake, description of, 207
-dchars preprocessor option, description of, 339
dead code elimination pass optimization, features of, 25
debug command-line option for Libtool, description of, 233
debuggers, troubleshooting, 255
debugging information, adding, 94–99
debugging options, table of, 95–96
debugging unnumbered options, list of, 344
decision coverage, role in test coverage, 149
decisions, embedding with C and C++, 151
DejaGNU test harness, downloading, 20–21
DEJAGNULIBS environment variable, setting, 21
--demangle[=style] command-line option in gprof, description of, 170
depcomp auxiliary file in automake, description of, 209
DEPENDENCIES_OUTPUT environment variable, purpose of, 102
dependency output, generating for Make utility, 102
deprecated attribute, purpose of, 125, 128
designated initializers, relationship to C and extensions, 120–122
DEUTSCH preprocessor macro, defining, 77–78
diagnostic options, list of, 345
directory search options, list of, 345
directory search path, modifying, 78–82
--disable configuration options, descriptions of, 12–16
dist* Makefiles produced by automake, descriptions of, 208
distclean makefile target, description of, 19

DJGPP (DJ's GNU Programming Platform)
components of, 51
configuring, 54–56
downloading and installing, 49–54
overview of, 47–49
using in DOS and Windows, 56
DL (dynamically loaded) libraries, features of, 225–226
-dletters command-line options, descriptions of, 285–287
-dlopen *file* mode for Libtool, working in, 237
-dlreopen *file* mode for Libtool, working in, 237
-dmod debugging option, description of, 96–97
-Dname* preprocessor options, descriptions of, 77, 338
documentation, enhancements made to, 32
DOS environments
 using Cygwin in, 45–47
 using DJGPP in, 56
Down GNU Info navigation command, description of, 268
DPMI (DOS Protected Mode Interface), relationship to DJGPP, 47
dragon book, obtaining, 136
--dribble=*file* command-line option in GNU Info, description of, 278
dry-run command-line option for Libtool, description of, 233
-dump* options
 arguments for, 97–98
 description of, 96–97
 descriptions of, 60–62, 287
dvi makefile target, description of, 19
DWARF2 debugging format, default of ELF to, 37

E

E and e arguments for dump option, descriptions of, 97
-E command-line option, description of, 287
e GNU Info navigation command, description of, 268, 273
-E output option
 advisory about using with linker, 83
 description of, 64–66
 significance of, 69
%E spec processing instruction, description of, 108
echo area in GNU Info, explanation of, 271

- eh-frame-hdr link option, lack of support for, 86
 - ELF binary format, treatment of, 37
 - elinks browsers Web address, 358
 - enable configuration options, descriptions of, 12–16
 - endfile built-in spec string, description of, 106
 - enscript, using with GNU Info, 279
 - enumerations, treatment of, 36
 - environment variables
 - customizing GCC with, 101–103
 - modifying in Makefiles, 252
 - specifying for DJGPP, 54
 - error messages, troubleshooting, 258–259
 - ERROR test suite result code, description of, 22
 - errors versus warning messages, 18
 - Esc versus Alt key in GNU Info, advisory about, 269
 - %estr predefined substitution spec, description of, 107
 - Eunice emulation layer, purpose of, 189
 - exec configuration option, description of, 12
 - execute mode for Libtool, working in, 235
 - exit() library function, relationship to function attributes, 125
 - Expect tool, downloading and checking for installation of, 20
 - export* modes for Libtool, working in, 237–238
 - extensions, treatment of, 37
 - extraclean makefile target, description of, 19
- F**
- F and f arguments for dump option, descriptions of, 97
 - f *file* command-line option in GNU Info, description of, 278–279
 - f option for gcov program
 - description of, 154
 - example of, 162
 - FAIL test suite result code, description of, 22
 - falign* command-line options, descriptions of, 140–142, 287–288
 - fallow-single-precision command-line option, description of, 70, 287
 - falt-external-templates command-line option, description of, 288
 - fargument-alias command-line option, description of, 287
- fbounds-check command-line option, description of, 144, 289
 - fbranch-probabilities command-line option, description of, 289
 - fbuiltin command-line option in C dialect, description of, 70, 73
 - fcall* command-line options, descriptions of, 289
 - fcaller-saves level 2 optimization, description of, 140
 - fcheck-new command-line option, description of, 75, 290
 - fcond-mismatch command-line option, description of, 70, 290
 - fconserve-space command-line option, description of, 75, 290
 - fconstant-string command-line option, description of, 290
 - fcprop-registers level 1 optimization optimization, description of, 139
 - fcross-jumping level 1 optimization, description of, 139
 - fcse* command-line options, descriptions of, 140–142, 290
 - fdata-sections command-line option, description of, 291
 - fdefault-inline GCC optimization, description of, 144
 - fdefer-pop level 1 optimization, description of, 139
 - fdelayed-branch command-line option, description of, 139–140, 291
 - fdelete-null-pointer-checks command-line option, description of, 140, 291
 - fdiagnostics-show-location command-line option, description of, 291
 - fdollars-in-identifiers command-line option, description of, 75, 291
 - fdump* command-line options, descriptions of, 95, 291, 292
 - features command-line option for Libtool, description of, 233
 - Feldman, Stu and development of make program, 190
 - fexceptions command-line option, description of, 293
 - fexpensive-optimizations command-line option, description of, 141, 293
 - fexternal-templates command-line option, description of, 293
 - ffast-math command-line option, description of, 144, 293
 - ffgcse* level 2 optimizations, descriptions of, 141
 - ffinite-math-only GCC optimization, description of, 144

- ffixed-reg command-line option, description of, 293
- ffloat-store command-line option, description of, 144, 294
- fforce* command-line options, descriptions of, 141–143, 144, 294
- ffor-scope command-line option, description of, 294
- ffreestanding command-line option, description of, 70, 72, 294
- ffunction command-line option, description of, 144, 295
- fgcse* command-line options, descriptions of, 295
- fgnu-runtime command-line option, description of, 295
- fguess-branch-probability level 1 optimization, description of, 139
- fhosted command-line option, description of, 70, 72, 296
- Fibonacci application, using `configure.ac` file with, 240
- `fibonacci.c` sample code, 156
 - output file produced by `gcov` for, 158–159
 - running with `gcov` and `-b` option, 160–161
 - setting `no_instrument_function()` attribute in, 184–185
- fif-conversion* level 1 optimizations, descriptions of, 139
- file=*file* command-line option in GNU Info, description of, 278
- filename suffixes, list of, 63
- filename symbol specification in `gprof`, description of, 168
- filename-line-numbersymbol specification in `gprof`, description of, 168
- file-ordering *map_file* command-line option in `gprof`, description of, 170
- fin* command-line options, descriptions of, 296–297
- finhibit-size-directive command-line option, description of, 296
- finish mode for Libtool, working in, 233, 235
- finline* GCC optimizations, descriptions of, 144
- finstrument-functions option in gcc compiler, purpose of, 182, 185
- fixincludes script in GCC 3.3.1, bugs in, 246
- fixproto script in GCC 3.3.1, bugs in, 247
- fkeep*
 - command-line options, 297
 - GCC optimizations, 144
- flat profiles
 - providing with `gprof` applications, 164
 - providing with `gprof` code profiler, 177
- fleading-underscore command-line option, description of, 297
- flexible array members, using with C99 standard, 116
- floop-optimize level 1 optimization, description of, 139–140
- fmath-errno command-line option, description of, 145, 297
- fmem-report command-line option, description of, 95, 297
- fmerge* command-line options, descriptions of, 139, 145, 297
- fmessage-length=*n* command-line option, description of, 298
- fmove-all-movables command-line option, description of, 298
- fms-extensions command-line option, description of, 75, 298
- fnext-runtime command-line option, description of, 298
- fno* command-line options
 - in C dialect, 70, 73
 - in C++, 75–76
 - description of, 339
 - descriptions of, 298–303
- fnon-call-exceptions command-line option, description of, 303
- fomit-frame-pointer command-line option, description of, 139–140, 303
- foptimize* command-line options, descriptions of, 141, 143, 303–304
- format string exploit, explanation of, 91
- FORTRAN, downloading source for, 5–7
- fpack-struct command-line option, description of, 304
- fpcc-struct-return command-line option, description of, 304
- fpeephole2 level 2 optimization, description of, 141–142
- fpermissive command-line option, description of, 76, 304
- fpic command-line option, description of, 304
- fPIC command-line option, description of, 304
- fprefetch-loop-arrays command-line option, description of, 305
- fpreprocessed preprocessor option, description of, 339
- fpretend-float command-line option, description of, 95, 305
- fprintf() statement, relationship to test coverage, 150

- fprofile-arcs command-line option,
 - description of, 153, 305
- freduce-all-givs command-line option,
 - description of, 305
- fremove command-line option,
 - description of, 141, 306
- freg-struct-return command-line option,
 - description of, 305
- frename-registers command-line option,
 - description of, 306
- freorder-functions command-line option,
 - description of, 305
- frepo command-line option in C++,
 - description of, 76
- frerun* command-line options,
 - descriptions of, 141, 306
- fsched* command-line options,
 - descriptions of, 141, 306
- fshared-data command-line option,
 - description of, 307
- fshort* command-line options,
 - descriptions of, 70, 307
- fsigned* command-line options,
 - descriptions of, 70–71, 307
- fsingle-precision-constant command-line option,
 - description of, 307
- fssa* command-line options,
 - descriptions of, 307–308
- fstack* command-line options,
 - descriptions of, 308
- fstats command-line option,
 - description of, 308
- fstrength-reduce command-line option,
 - description of, 141, 308
- fstrict-aliasing command-line option,
 - description of, 141, 308
- fsynchronous-unwind command-line option,
 - description of, 289
- fsyntax-only command-line option,
 - description of, 308
- fsyntax-only warning option,
 - description of, 87
- ftabstop=*width* preprocessor option,
 - description of, 339
- ftemplate-depth-*n* command-line option,
 - description of, 309
- ftemplate-depth-*N* command-line option in C++,
 - description of, 76
- ftest-coverage command-line option,
 - description of, 95, 153, 309
- fthread-jumps command-line option,
 - description of, 139, 309
- ftime-report command-line option,
 - description of, 95, 309
- ftls-model=*model* command-line option,
 - description of, 309
- FTP clients, downloading source for Glibc from, 379–380
- ftracer command-line option,
 - description of, 309
- ftrapping-math GCC optimization,
 - description of, 145
- ftrapv
 - command-line option, 309
 - GCC optimization, 145
- function attributes, declaring, 123–127
- function calls
 - analyzing behavior, performance, and interaction between, 148
 - constructing with C and extensions, 113–114
- function indexes, providing with gprof code profiler, 177
- function names
 - mapping addresses to, 183
 - as strings, 131–133
- _FUNCTION_ identifier, purpose of, 131–133
- function-name symbol specification in gprof, description of, 168
- function-ordering command-line option in gprof, description of, 170
- funroll* command-line options,
 - descriptions of, 309
- unsafe-math-optimizations
 - command-line option, description of, 145, 310
- funsigned* command-line options,
 - descriptions of, 71–72, 310
- funwind-tables command-line option,
 - description of, 310
- fuse-cxa-atexit command-line option,
 - description of, 76, 310
- fverbose-asm command-line option,
 - description of, 311
- fvolatile* command-line options,
 - descriptions of, 311
- fvtable-command-line option,
 - description of, 311
- fwritable strings command-line option in C dialect, description of, 70, 311

G

- G and g arguments for dump option,
 - descriptions of, 97
- g command-line option, description of, 94–95, 311–312
- %G spec processing instruction,
 - description of, 108
- g[n] debugging option, description of, 95
- g++ compiler, enhancements made to, 27
- G++ troubleshooting mangling, in, 255

GCC
 building from CVS, 6
 code profiling with, 163–164
 compiling, 16–20
 compiling C++ code with, 74–76
 downloading source code, 5–7
 as driver program, 103
 finding current running versions of, 249
 installing, 23–24
 invoking with options and arguments, 59–60
 preparing installation system, 3–5
 previewing build process of, 2–3
 rationale for building from source, 1–2
 requirements for build process, 3–4
 using multiple versions on single systems, 248
 verifying software requirements for, 2–3
 Web site, 358
 GCC 3.0.1, changes in, 32
 GCC 3.0.2, changes in, 32–33
 GCC 3.0.3, changes in, 33
 GCC 3.0.4, changes in, 33
 GCC 3.1.1, changes in, 35
 GCC 3.3.1, problems identified in release notes for, 246–247
 GCC 3.3.2, architecture and system-specific configuration files in, 403–405
 GCC 3 enhancements
 to documentation, 32
 garbage collection versus obstacks, 31
 gccbug script, 31
 language-independent warning options, 31
 Lengauer and Tarjan algorithm, 31
 to libgcc library, 31
 new languages and language-specific improvements, 26–30
 new targets and target-specific improvements, 30–31
 optimizer improvements, 24–26
 target-independent code generation options, 31
 to test suite, 31
 GCC 3.1, changes in, 34–35
 GCC 3.2, changes in, 35–36
 GCC 3.2 or newer, using with Glibc, 377
 GCC 3.3, changes in, 36
 gcc, adding profiling code with, 182–185
 GCC assembler option, *-Wa,option*, 336
 gcc binary, troubleshooting, 251
 GCC command-line options, 59–62
 `-###`, 284
 `--00`, 315
 `--0|-01`, 315
 `--02`, 316
 `--03`, 316
 `-ansi`, 284
 `-aux-info`, 284
 `-bmachine`, 285
 `-Bprefix`, 285
 `-c`, 285
 `-delete-null-pointer-checks`, 291
 `-dletters`, 285–287
 `-dump*`, 287
 `-E`, 287
 `-falign*`, 287–288
 `-fallow-single-precision`, 287
 `-falt-external-templates`, 288
 `-fargument-alias`, 287
 `-fbounds-check`, 289
 `-fbranch-probabilities`, 289
 `-fcall*`, 289
 `-fcheck-new` command-line option, 290
 `-fcond-mismatch`, 290
 `-fconserve-space`, 290
 `-fconstant-string`, 290
 `-fcse*`, 290
 `-fdata-sections`, 291
 `-fdelayed-branch`, 291
 `-fdiagnostics-show-location`, 291
 `-fdollars-in-identifiers`, 291
 `-fdump*`, 292
 `-fdump-class-hierarchy`, 291
 `-fexceptions`, 293
 `-fexpensive-optimizations`, 293
 `-fexternal-templates`, 293
 `-ffast-math`, 293
 `-ffixed-reg`, 293
 `-ffloat-store`, 294
 `-fforce*`, 294
 `-ffor-scope`, 294
 `-ffreestanding`, 294
 `-ffunction`, 295
 `-fgcse*`, 295
 `-fgnu-runtime`, 295
 `-fhosted`, 296
 `-fin*`, 296–297
 `-finhibit-size-directive`, 296
 `-fkeep*`, 297
 `-fleading-underscore`, 297
 `-fmath-errno`, 297
 `-fmem-report`, 297
 `-fmerge*`, 297
 `-fmessage-length=n`, 298
 `-fmove-all-movables`, 298
 `-fms-extensions`, 298
 `-fnext-runtime`, 298
 `-fno*`, 298–303
 `-fnon-call-exceptions`, 303
 `-fomit-frame-pointer`, 303

- GCC command-line options, *(continued)*
- foptimize-register-move, 303
 - fpack-struct, 304
 - fpcc-struct-return, 304
 - fpermissive, 304
 - fpic command-line option, 304
 - fPIC command-line option, 304
 - fprefetch-loop-arrays, 305
 - fpretend-float, 305
 - fprofile-arcs, 305
 - freduce-all-givs, 305
 - fregmove command-line option, 306
 - freg-struct-return, 305
 - frename-registers command-line option, 306
 - freorder-functions, 305
 - frerun* command-line options, 306
 - fsched* command-line options, 306
 - fshared-data, 307
 - fshort*, 307
 - fsigned*, 307
 - fsingle-precision-constant, 307
 - fssa*, 307
 - fstack*, 308
 - fstats, 308
 - fstrength-reduce, 308
 - fstrict-aliasing, 308
 - fsynchronous-unwind, 289
 - fsyntax-only, 308
 - ftemplate-depth-*n*, 309
 - ftest-coverage, 309
 - fthread-jumps, 309
 - ftime-report, 309
 - ftls-model=*model*, 309
 - ftracer, 309
 - ftrapv, 309
 - funroll-all-loops, 309–310
 - unsafe-math-optimizations, 310
 - funsigned*, 310
 - funwind-tables, 310
 - fuse-cxa-atexit, 310
 - fverbose-asm, 311
 - fvolatile*, 311
 - fvtable-gc, 311
 - fwritable-strings, 311
 - g, 311–312
 - gcoff, 312
 - gdwarf*, 312
 - gen-decls, 312
 - ggdb, 313
 - glevel, 312
 - gstabs*, 313
 - gvms, 313
 - gxcoff*, 313
 - help, 313
 - I-, 314
 - Idir, 314
 - Ldir, 315
 - no-integrated-cpp, 315
 - nostdinc*, 315
 - o *file*, 316
 - p, 316
 - param *name=value*, 317
 - pass-exit-codes, 316
 - pedantic*, 317–318
 - pg, 318
 - pipe, 318
 - print*, 318–320
 - Q, 320
 - S, 320
 - save-temp, 320
 - std=std, 320–321
 - target-help, 321
 - time, 321
 - traditional, 321–322
 - trigraphs, 323
 - v, 323
 - V *version*, 323
 - W, 323–324
 - Wabi, 324
 - Waggregate-return, 324
 - Wall, 324–325
 - Wbad-function-cast, 325
 - Wcast*, 325
 - Wchar-subscripts, 325
 - Wcomment, 325
 - Wconversion, 326
 - Wctor-dtor-privacy, 326
 - Wdisabled-optimization, 326
 - Wdiv-by-zero, 326
 - Weffc++, 326–327
 - Werror*, 327
 - Wfloat-equal, 327
 - Wformat*, 327–328
 - Wimplicit*, 328
 - Winline, 328
 - Wlarger-than-len, 328
 - Wlong-long, 328
 - Wmain, 328
 - Wmissing*, 329
 - Wmultichar, 329
 - Wnested-externs, 329
 - Wno*, 330–331
 - Wnon*, 331
 - Wold-style-cast, 331
 - Woverloaded-virtual, 332
 - Wpacked, 331
 - Wpadded, 331
 - Wparentheses, 332
 - Wpointer-arith, 332
 - Wredundant-decls, 332
 - Wreorder, 332
 - Wreturn-type, 332
 - Wselector, 332

- Wsequence, 333
- Wshadow, 333
- Wsign*, 333
- Wstrict-prototypes, 333
- Wswitch, 333
- Wsynth, 333
- Wsystem-headers, 334
- Wtraditional, 334
- Wtrigraphs, 335
- Wun*, 335–336
- GCC customization**
 - with environment variables, 101–103
 - with spec files and spec strings, 103–108
- GCC linker options**
 - library, 336
 - lobjc, 336
 - no*, 337
 - nodefaultlibs, 336
 - s, 337
 - shared*, 337
 - static, 337
 - symbolic, 337
 - u *symbol*, 337
 - Wl,*option*, 337
 - Xlinker *option*, 337
- GCC newsgroups, descriptions of,** 351–352
- GCC optimizations, list of,** 144–145
- GCC option reference**
 - C language options, 344
 - C++ language options, 344
 - code generation options, 344
 - debugging unnumbered options, 344
 - diagnostic options, 345
 - directory search options, 345
 - linker options, 345
 - Objective C language options, 345
 - optimization options, 345
 - output and processing options, 345
 - preprocessor options, 345
 - target options, 345
- GCC output options, list of,** 64
- gcc preprocessor option, description of, 340
- GCC preprocessor options**
 - \$, 338
 - A*, 338
 - ansi, 338
 - Apredicate, 338
 - C, 338
 - dchars, 339
 - Dname*, 338
 - fno-show-column, 339
 - fpreprocessed, 339
 - ftabstop=*width*, 339
 - gcc, 340
- H, 340
- help, 340
- idirafter *dir*, 340
- imacros *file*, 340
- include *file*, 340
- iprefix *suffix*, 340
- isystem *dir*, 340
- iwithprefix *dir*, 341
- M, 341
- MD, 341
- MF *file*, 342
- MG, 342
- MM, 342
- MMD, 342
- MP, 342
- MQ *target*, 342
- MT *target*, 342
- o *file*, 342
- P, 343
- remap, 343
- U *name*, 343
- undef, 343
- v, 343
- version|, 343
- w, 343
- W*option*, 343
- Wp,*option*, 343
- x *lang*, 343
- GCC problems**
 - abuse of _STDC_ definition, 261–262
 - build and installation problems, 262–263
 - compatibility issues, 254–256
 - include files or libraries, 257–258
 - incompatibilities between GNU C and K&R C, 259–261
 - mixing GNU with other toolchains, 251–254
 - moving GCC after installation, 250–251
 - optimization, 256–257
 - program execution, 247–251
 - programs compiled with GCC, 249–250
 - running out of memory, 250
 - warning and error messages, 258–259
- GCC releases, obtaining change information about,** 246
- GCC resources**
 - mailing lists, 352–358
 - publications, 359–361
 - Usenet, 347–352
 - on Web, 358–359
- GCC test suite, running,** 20–23
- GCC_EXEC_PREFIX environment variable, purpose of,** 101–102
- gccbug script, availability of, 31

- `gcc/config` subdirectory, contents of, 403
- `gcc.gnu.org`, mailing lists at, 353–357
- GCC-related mailing lists
 - netiquette for, 356–357
 - posting to, 353
- GCC's C and extensions
 - arithmetic on void and function pointers, 120
 - arrays of variable length, 117–119
 - and case ranges, 123
 - declaring function attributes, 123–127
 - designated initializers, 120–122
 - function calls, 113–114
 - function names as strings, 131–133
 - inline functions, 130–131
 - labels as values, 111–112
 - locally declared variables, 110–111
 - macros with variable number of arguments, 119
 - mixed declarations and code, 123
 - nested functions, 112–113
 - nonconstant initializers, 120
 - overview of, 108–109
 - #pragmas, 133–134
 - specifying variable attributes, 127–130
 - subscripting non-Lvalue arrays, 119–120
 - `typeof` used with types, 114–115
 - zero-length arrays, 115–117
- GCC's output, controlling, 62–69
- GCC's search path, controlling
 - modification of, 101
- `gcj` GNU Compiler for Java, integration of, 26
- `-gcoff` command-line option,
 - description of, 95, 312
- `gcov` session, example of, 155–163. *See also* test coverage
- GDB versions, advisory about, 37, 255
- `-gdwarf*` command-line options,
 - descriptions of, 95, 312
- `-gen-decls` command-line option,
 - description of, 312
- generated-manpages makefile target,
 - description of, 19
- Gettext, downloading, 6
- `-ggdb` command-line option,
 - description of, 94–95, 313
- `-glevel` command-line option,
 - description of, 312
- Glibc. *See also* libraries
 - alternatives to, 365–367
 - backing out of upgrades, 397–399
 - building, 385–388
 - compiling, 388–389
 - configuring for compilation, 386–388
 - documentation for, 400–401
 - downloading and installing source code for, 379–382
 - error associated with `ld.map` file, 389
 - features of, 363–365
 - getting information about versions of, 371–373
 - identifying on systems, 370–371
 - installing, 390–393
 - installing alternative version of, 399–400
 - installing source code archives for, 382–384
 - mailing lists for, 401
 - potential problems in upgrading of, 369–370
 - previewing build process of, 374–376
 - rationale for building from source, 367–369
 - reporting problems with, 402
 - resolving upgrade problems using BusyBox, 394–396
 - suggested location for, 385
 - testing build of, 390
 - tool recommendations, 376–378
 - troubleshooting installation of, 394–399
 - Web address for, 379
 - Web sites related to, 401
- Glibc add-ons
 - Berkeley DB NSS (name Service Switch), 373
 - `glIBC-linuxthreads`, 373
 - `nss-lwres` module, 373
- Glibc configuration, kernel optimization support for, 388
- Glibc source code directory, integrating add-ons into, 384–385
- `glIBC-compat` add-on, discontinuation of, 373
- `glIBC-crypt` add-on, discontinuation of, 373
- `glIBC-linuxthreads` Glibc add-on, features of, 373
- global null pointer test elimination optimization, features of, 25–26
- global transformations, relationship to optimization, 136
- `gmon.out` file, creating with `gprof` code profiler, 174
- Gnatsweb Web address, 355
- GNU awk 3.0, using with Glibc, 377
- GNU binutils 2.3 or later, using with Glibc, 377
- GNU build tools, resource for, 218

GNU C and K&R C, incompatibilities between, 259–261
 GNU FTP site, accessing, 5–7
 GNU Info
 accessing documentation for, 270
 anatomy of screens in, 269–271
 displaying first page of, 272
 echo area in, 271
 following xrefs (cross-references) in, 276–277
 getting help on, 281
 invoking, 278–279
 menu references in, 271, 276
 moving around in, 272–274
 note references in, 276
 overview of, 265–267
 performing searches in, 274–275
 printing nodes in, 277–278
 selecting nodes in, 271
 tips and tricks for use of, 279–281
 using, 267–269
 using command multipliers with, 280
 working with multiple windows in, 280–281
 GNU libc 2.2.3, advisory about building of, 3
 GNU m4 macro preprocessor Web address, 193
 GNU mailing lists, subscribing to, 352
 GNU Makefile conventions. *See* Makefiles
 GNU, mixing with other toolchains, 251–254
 GNU newsgroups, descriptions of, 350–351
 GNU Press Web address, 359
 GNU ‘sed’ 3.02 or newer, using with Glibc, 378
 GNU ‘texinfo’ 3.12f, using with Glibc, 378
 GNU utilities, updating for use with Glibc, 379
 GNUMake 3.79 or newer, using with Glibc, 377
 goto statements, using with labels as values, 111–112
 Gperf, downloading, 6
 gprof code profiler
 obtaining and compiling, 164–166
 symbol specifications in, 168
 using, 163–164, 167–173
 gprof fibonacci file, default output from, 174–177
 gprof session, example of, 173–178
 GRX DJGPP component, description of, 51
 -gstabs* command-line options, descriptions of, 95–96, 313

%gsuffix predefined substitution spec, description of, 107

-gvms command-line option, description of, 96, 313
 -gcoff* command-line options, descriptions of, 96, 313

H

-h
 configure script option, 219
 option for gcov program, 155
 H and h arguments for dump option, descriptions of, 97
 -h command-line option in GNU Info, description of, 278
 -H preprocessor option, description of, 340
 .h suffix, operation associated with, 63
 H8/300 options, overview of, 429
 hallo.c sample code, 77–78
 header files, regenerating, 257
 HEADERS primary in automake, description of, 207
 --help command-line option
 description of, 60, 313, 340
 in GNU Info, 278
 for Libtool, 233
 help, obtaining in GNU Info, 281
 host system versus build and target systems, 9–10
 hosted environment, explanation of, 72
 HP/PA (PA/RISC) options, overview of, 430–431

I

I argument for dump option, description of, 97
 -I- command-line option, description of, 79–82, 314
 -i command-line option in gprof, description of, 170
 -I dirs command-line option
 description of, 79–80
 in gprof, 170
 %i predefined substitution spec, description of, 107
 i search function in GNU Info, description of, 275
 .i suffix, operation associated with, 63
 i386 and AMD x86-64 options, overview of, 432–438
 IA-64 options, overview of, 438–440
 IBM assembler, using on AIX systems, 254
 -Idir command-line option, description of, 314

- idirafter *dir* preprocessor option, description of, 340
 - if-conversion pass optimization, features of, 26
 - ifnames utility in autoconf package, description of, 197
 - .ii suffix, operation associated with, 63
 - imacros *file* preprocessor option, description of, 340
 - .in suffix, using with autoconf program, 201
 - #include directives, example of, 79–80
 - include *file* preprocessor option, description of, 340
 - %include *file* spec file command, description of, 105
 - include files, troubleshooting problems with, 257–258
 - %include_noerr *file* spec file command, description of, 105
 - index-search *string* command-line option in GNU Info, description of, 278
 - info command, using, 266
 - infodir configuration option, description of, 12
 - initializers, relationship to C and extensions, 120–123
 - inline functions, relationship to C and extensions, 130–131
 - input files, treating as source code files, 66
 - INSTALL file
 - in automake, 209
 - in Glibc, 376
 - install Makefile
 - producing with automake, 208
 - target for, 19
 - install mode for Libtool, working in, 236
 - installation problems, resolving, 262–263
 - installation system, preparing, 3–5
 - installing GCC, 23–24
 - install-sh auxiliary file in automake, description of, 209
 - int len declarations, using with arrays of variable length, 118–119
 - Intel 960 options, overview of, 440–442
 - internationalization, relationship to Glibc, 389
 - Internet connection, specifying for Cygwin, 42
 - iostream methods, treatment of, 37
 - IPC (interprocess communication) mechanisms, pipes as, 62
 - iprefix *suffix* preprocessor option, description of, 340
 - ISO C, Glibc support for, 364
 - ISO C99 features, additional support for, 28
 - ISO/ANSI C, concerns associated with, 69–70
 - isystem *dir* preprocessor option, description of, 340
 - iwithprefix *dir* preprocessor option, description of, 341
- J**
- j argument for dump option, description of, 97
 - J[*symbol-specification*] command-line option in gprof, description of, 171
- K**
- k argument for dump option, description of, 97
 - k command-line option in gprof, description of, 171
 - K&R C and GNU C, incompatibilities between, 259–261
 - Knews newsreader, downloading, 349
- L**
- L and l arguments for dump option, descriptions of, 97–98
 - L and l command-line options in gprof, descriptions of, 171
 - L *dir* option, description of, 79, 82
 - l GNU Info navigation command, description of, 268
 - l option for gcov program, description of, 155
 - %l spec processing instruction, description of, 108
 - %L spec processing instruction, description of, 108
 - labels as values, using with C and extensions, 111–112
 - LANG environment variable, purpose of, 103
 - languages, specifying in input files, 63
 - LC_ALL environment variable, purpose of, 103
 - LC_CTYPE environment variable, purpose of, 103

- `LC_MESSAGES` environment variable,
purpose of, 103
- `ldconfig` command
issuing, 23
relationship to shared libraries, 224
- `ldd` command, example of, 224
- `-Ldir` command-line option, description of, 315
- `Left GNU Info navigation command`,
description of, 268
- Lengauer and Tarjan algorithm,
availability of, 31
- level 1 optimizations, examples of, 138–140
- level 2 optimizations, enabling with `-O3`, 144
- level 2 optimizations, overview of, 140–143
- `-lgls` option, troubleshooting on SGI systems, 256
- `lib` built-in spec string, description of, 106
- `--libdir` configuration option,
description of, 12
- `libgcc` built-in spec string, description of, 106
- libraries. *See also* Glibc
dynamically loaded libraries, 225–226
linking with Libtool, 243–244
multilibs, 319
overview of, 221
shared libraries, 222–225
static libraries, 222
- `LIBRARIES` primary in automake,
description of, 207
- libraries, troubleshooting problems with, 257–258
- `LIBRARY_PATH` environment variable,
purpose of, 103
- Libtool
downloading, 228
features of, 226–227
files installed by, 230–231
getting more information about, 244
installing, 228–230
integrating with `configure.ac` file, 240
integrating with `Makefile.am` file, 241
troubleshooting problems with, 243–244
using from command line, 232–239
using with autoconf and automake programs, 239–243
- `link` built-in spec string, description of, 106
- link mode for Libtool, working in, 236–237
- linker
controlling, 82–86
passing arguments to, 252
preventing from using standard libraries, 85
- linker built-in spec string, description of, 106
- linker name, relationship to shared libraries, 225
- linker options, list of, 345
- `-Llibdir` mode for Libtool, working in, 238
- `-library` linker option, description of, 336
- `-lname`
link option, 83–85
mode for Libtool, working in, 238
- `-lobjc` linker option, description of, 336
- local transformations, relationship to optimization, 136
- localedata Glibc add-on,
discontinuation of, 373
- locally declared variables, using with C and extensions, 110–111
- `longjmp()` function, setting, 260
- `ltdl.*` files in Libtool, descriptions of, 232
- `LTLIBRARIES` primary in automake,
description of, 207
- `ltmain.sh` file in Libtool, description of, 232
- lvalue arrays, subscripting non-lvalue arrays, 119–120

M

- `M` and `m` arguments for `dump` option, descriptions of, 98
- `-m num` command-line option in `gprof`, description of, 171
- `-M` preprocessor option, description of, 341
- `.m` suffix, operation associated with, 63
- `m4` macro preprocessor Web address, 193
- M32R options, overview of, 442–443
- M68hc1x options, overview of, 447
- M88K options, overview of, 447–451
- M680x0 options, overview of, 443–446
- MacKenzie, David and development of `autoconf` program, 188–189
- macros
using with `autoconf` program, 204
with variable number of arguments, 119
- mailing lists
for `autoconf` program, 193
for `automake` program, 193, 198

- for GCC, 352–358
- for Glibc, 401
- maintainer-clean makefile target, description of, 19
- make bootstrap command, issuing, 3–4, 17–18
- make check command
 - using with autoconf, 195–196
 - using with automake, 200
- make commands
 - building Libtool with, 229–230
 - issuing, 16–17
- make install command
 - using with autoconf, 196
 - using with automake, 200
- make program, features of, 190
- Make utility, generating dependency output for, 102
- makeables, overview of, 18–19
- Makefile conventions, obtaining information about, 191
- Makefile templates, creating for use with autoconf program, 206
- Makefile.am files
 - creating for use with automake, 207–211
 - example of, 208
 - purpose of, 206
 - using with Fibonacci application, 240–241
- Makefiles
 - contents of, 190
 - generating, 215–216
 - modifying environment variables in, 252
 - using, 216–217
- malloc attribute
 - obtaining replacements for, 250
 - purpose of, 125
- mandir configuration option, description of, 12
- MANS primary in automake, description of, 207
- MCORE options, overview of, 451–452
- MD preprocessor option, description of, 341
- memory, running out of, 250
- menu references in GNU Info
 - example of, 276
 - explanation of, 271
- metaconfig program, early Configure scripts in, 189
- Meta-v GNU Info navigation command, description of, 268
- MF *file* preprocessor option, description of, 342
- MG preprocessor option, description of, 342
- MinGW, features of, 57
- MIPS options, overview of, 452–459
- misfeatures and bugs, coping with, 245–247
- missing auxiliary file in automake, description of, 210
- mkinstalldirs auxiliary file in automake, description of, 209
- MM preprocessor option, description of, 342
- MMD preprocessor option, description of, 342
- MMIX options, overview of, 460–461
- MN10200 options, overview of, 461
- MN10300 options, overview of, 462
- mode (*m*) attribute, purpose of, 128
- mode=MODE command-line option for Libtool, description of, 233
- moderated newsgroup, explanation of, 348
- modified condition decision coverage, role in test coverage, 150
- module mode for Libtool, working in, 238
- mortal user, building GCC as, 3–4
- mostlyclean makefile target, description of, 19
- MP preprocessor option, description of, 342
- MQ *target* preprocessor option, description of, 342
- MT *target* preprocessor option, description of, 342
- multilibs
 - Newlib support for, 367
 - overview of, 319
- M-x print-node command in GNU Info, using, 277–278
- myfile.c sample code, 81
- myprog.c source file, compiling, 64

N

- N and n arguments for dump option, descriptions of, 98
- n GNU Info navigation command, description of, 268, 272
- n option
 - for configure scripts, 219
 - for gcov program, description of, 155
- n[*symbol-specification*] command-line option in gprof, description of, 171
- N[*symbol-specification*] command-line option in gprof, description of, 171
- %(*name*) predefined substitution spec, description of, 107
- native compiler, purpose of, 10

nested functions, using with C and extensions, 112–113
 netnews, selecting software for reading of, 349
 Newlib, using as alternative to Glibc, 366–367
 NEWS auxiliary file in automake description of, 210 example of, 215
 news servers, specifying for newsgroups, 347–348
 newsgroups, moderated and unmoderated types of, 348
 NewsXpress newsreader, downloading, 349
-no*
 linker options, 337
 modes for Libtool, 238
-nodefaultlibs link option, description of, 83, 85, 336
 nodes, selecting in GNU Info, 271, 273, 278
-no-integrated-cpp command-line option, description of, 70, 315
 noreturn keyword, using with function attributes, 126
-nostartfiles link option, description of, 83, 85
-nstdinc* command-line options, descriptions of, 76, 82, 315
-nostdlib link option, description of, 83, 85
 note references in GNU Info, example of, 276–277
 NS32K options, overview of, 462–464
 nss-lwres module Glibc add-on, features of, 373
O
-o argument for dump option, description of, 98
-O command-line option in GNU Info, description of, 279
-o *directory|file* option for gcov program, description of, 155
-o *file* command-line option description of, 316, 342
 in GNU Info, 278
 using, 64–66, 68
.o files versus .so files, relationship to linker, 83
-o *output-file* mode for Libtool, working in, 238
%O predefined substitution spec, description of, 107
%o predefined substitution spec, description of, 107
OBJC_INCLUDE_PATH environment variable, purpose of, 102
 object filenames, specifying, 84–85
 object files
 finding with **GCC_EXEC_PREFIX**, 101–102
 linking with Libtool shared library objects, 243
 producing from compiled assembly code, 68
--object-directory *directory* option for gcov program, description of, 155
--object-file *file* option for gcov program, description of, 155
 Objective C language
 downloading source for, 5–7
 options for, 345
-Oname command-line option in gprof, description of, 171
 optimization
 explanation of, 135
 options for, 345
 processor-independent optimization, 138–145
 processor-specific optimizations, 145–146
 troubleshooting problems with, 256–257
 optimizer, enhancements made to, 24–26
 options. *See* GCC command-line options
 output and processing options, list of, 345
--output *file* command-line option in GNU Info, description of, 278
 output files, specifying names of, 65
 output redirection, example of, 65–66

P

P and **p** arguments for dump option, descriptions of, 98
-p command-line option, description of, 96, 316
p GNU Info navigation command, description of, 268, 272
-P preprocessor option, description of, 343
-p[*symbol-specification*] command-line option in gprof, description of, 171
-P[*symbol-specification*] command-line option in gprof, description of, 171
 packages, selecting for Cygwin, 43
 packed attribute, purpose of, 129
 Page Down GNU Info navigation command, description of, 268
 Page Up GNU Info navigation command, description of, 268

- Pakke DJGPP component, description of, 51
 -param *name=value* command-line option, description of, 317
 parameter forward declarations, using with arrays of variable length, 118–119
 parentheses (), appearance in autoconf macros, 204
 PASS test suite result code, description of, 22
 --pass-exit-codes option, description of, 60, 62, 316
 path coverage, role in test coverage, 149
 PATH environment variable, setting for different command shells, 248
 PDP-11 options, overview of, 464–466
 -pedantic* warning options, descriptions of, 87, 90, 246, 317–318
 Perl 5 or better, using with Glibc, 378
 Perl interpreter, relationship to automake program, 198
 -pg command-line option, description of, 96, 173, 318
 PIC (position-independent code), generating, 255
 pilot.programmer.gnu GNU newsgroups, description of, 351
 --pipe command-line option, description of, 60, 318
 platform ports, availability of, 30–31
 POSIX, Glibc support for, 364
 PowerPC (PPC) options, overview of, 466–478
 _Pragma, relationship to function attributes, 127
 #pragmas, support for, 133–134
 predefines built-in spec string, description of, 106
 --prefix option, using to configure source code, 9, 11–12
 --prefix=*PREFIX* configure script option, description of, 219
 preprocessing, example of, 67
 preprocessor controlling, 77–78
 options for, 345
 passing options to, 252
 primaries in automake, list of, 207
 -print* command-line options, descriptions of, 61–62, 318–320
 printf() function
 analyzing applications with, 147
 problems associated with, 90–91
 using bounds-checking mode with, 29
 printme.c sample code, 91–93
 processor-independent optimization, performing, 138–145
 profiling. *See* code profiling
 --program configuration options, descriptions of, 12, 16
 program execution, troubleshooting, 247–251
 PROGRAMS primary in automake, description of, 207
 pure function attribute, using, 126
- ## Q
- Q command-line option, description of, 96, 320
 - q[*symbol-specification*] command-line option in gprof, description of, 172
 - Q[*symbol-specification*] command-line option in gprof, description of, 172
 - question mark (?) search function in GNU Info, description of, 275
 - quickstrap makefile target, description of, 19
 - quiet command-line option for Libtool, description of, 233
- ## R
- R and r arguments for dump option, descriptions of, 98
 - R *libdir* mode for Libtool, working in, 238
 - RamFloppy rescue disk, downloading, 393
 - README auxiliary file in automake description of, 210
 example of, 215
 - redundant compilations, eliminating, 136–138
 - release *release* mode for Libtool, working in, 238
 - remap preprocessor option, description of, 343
 - %rename *old new* spec file command, description of, 105
 - rescue disks
 creating when installing Glibc, 392–393
 resolving Glibc upgrade problems with, 396–397
 - resources
 mailing lists, 352–358
 publications, 359–361
 Usenet, 347–352
 on Web, 358–359
 - restageN makefile target, description of, 19

- restore=*file* command-line option in GNU Info, description of, 279
- Right GNU Info navigation command, description of, 268
- RIP rescue disk, downloading, 393
- root directory, selecting for Cygwin, 41
- rpath *libdir* mode for Libtool, working in, 238
- RS/6000 options, overview of, 478
- RSX DJGPP component, description of, 51
- RSXNTDJ, obtaining information about, 57
- RT options, overview of, 478–479

- S**
- S and s arguments for dump option, descriptions of, 98
- .S and .s suffixes, operations associated with, 63
- S command-line option, description of, 320
- s command-line option in gprof, description of, 172
- S and s GNU Info navigation commands, descriptions of, 268
- s link options, descriptions of, 83, 337
- S output option
 - advisory about using with linker, 83
 - description of, 64
 - significance of, 66, 69
- S and s search functions in GNU Info, descriptions of, 275
- %\$ spec processing instruction, description of, 108
- S390 and zSeries options, overview of, 479–480
- save-temps command-line option, description of, 61, 320
- scanf() function
 - problems associated with, 90–91
 - using bounds-checking mode with, 29
- SCRIPTS primary in automake, description of, 207
- search path, controlling modification of, 101
- searches, performing in GNU Info, 274–275
- section attribute, purpose of, 129–130
- setjmp() function, setting, 260
- setup.exe file, saving for Cygwin installation, 39–40
- SGI systems, troubleshooting -lbl_s option on, 256
- SH options, overview of, 480–482
- shared libraries, features of, 222–225
- shared* link options, descriptions of, 83, 85–86, 337
- show-options command-line option in GNU Info, description of, 279
- sibling call elimination, explanation of, 24
- signed_char built-in spec string, description of, 106
- sigsetjmp() and siglongjmp() functions, bugs associated with, 247
- silent command-line option for Libtool, description of, 233
- slrn (s-lang read news) newsreader, downloading, 349
- software requirements, verifying for GCC, 2–3
- Solaris #pragmas, features of, 134
- Solaris systems, troubleshooting malloc() function on, 256
- soname, relationship to shared libraries, 225
- source code
 - configuring, 9–11
 - configuring for Glibc, 385–388
 - displaying annotated source code, 178–182
 - downloading, 5–7
 - installing, 7–9
- source files
 - compiling, 65
 - treating input files as, 66
- SOURCES primary in automake, description of, 207
- Spacebar GNU Info navigation command, description of, 268
- SPARC options, overview of, 482–485
- SPARC systems, troubleshooting double values on, 256
- SPARClet Processor options, overview of, 485–486
- spec file commands, list of, 105
- spec file directives, list of, 104
- spec files
 - customizing GCC with, 103–108
 - troubleshooting, 249
- spec strings
 - built-in spec strings, 106
 - purpose of, 61
- **spec_name* spec file directive, description of, 104–105
- specs=*file* option, description of, 79, 82
- srcdir hint, using to configure source code, 9
- SSA (static single assignment)
 - optimization, features of, 25
- stage*N* makefile target, description of, 19

- standards compliance, applying to C dialects, 74
 - startfile built-in spec string, description of, 106
 - statement coverage, role in test coverage, 149
 - static libraries, features of, 222
 - static linker option, description of, 337
 - static mode for Libtool, working in, 238
 - static* link options, descriptions of, 83, 85–86
 - std=*std* command-line option, description of, 320
 - _STDC_ definition, abuse of, 261–262
 - std=*value* command-line option in C dialect, description of, 70, 73
 - strfmon() function, problems associated with, 90–91
 - strftime() function, problems associated with, 90–91
 - strings
 - customizing GCC with, 103–108
 - function names as, 131–133
 - subnodes command-line option in GNU Info, description of, 279
 - subscripting non-lvalue arrays, relationship to C and extensions, 119–120
 - substitution, relationship to spec strings, 106–107
 - suffix* spec file directive, description of, 104–105
 - swapme.c sample code, 81
 - symbol specifications in gprof, overview of, 168
 - symbolic linker option, description of, 337
 - symbolic links, troubleshooting in Glibc upgrades, 396
 - system header files, regenerating, 257
 - system names, significance of, 10–11
 - System V options, overview of, 487
 - system variables, specifying for DJGPP, 55
 - systems, types of, 9–10
 - SYSV Unix, Glibc support for, 365
- T**
- t argument for dump option, description of, 98
 - T command-line option in gprof, description of, 172
 - tail call elimination optimization, features of, 24–25
- tarballs, extracting when installing source code, 7
 - target options, list of, 345
 - target system versus build and host systems, 9–10
 - target-help command line option, description of, 321
 - Tcl (Tool Command Language), downloading and checking for installation of, 20
 - TCL_LIBRARY environment variable, setting, 21
 - tcsh (TOPS-20/TENEX C shell), updating PATH environment variables for, 248
 - test coverage. *See also* gcov session
 - overview of, 148–152
 - performing with gcov tool, 153–154
 - resources for, 152
 - test coverage analysis
 - compiling code for, 152–153
 - files used and produced during, 163
 - test_summary script, using, 23
 - TESTS primary in automake, description of, 207
 - TEXINFOS primary in automake, description of, 207
 - Textinfo source files
 - contents of, 266
 - example of, 266–267
 - time command-line option, description of, 61–62, 96, 321
 - TimeStorm tool, downloading, 46
 - tin (Tass + Iaian's) newsreader, downloading, 349
 - TMPDIR environment variable, purpose of, 103
 - TMS320C3x/C4x options, overview of, 488–490
 - toolchains
 - mixing GNU with, 251–254
 - relationship to Cygwin, 39
 - traditional* command line options
 - descriptions of, 71, 73, 321–322
 - treatment of, 37
 - using, 258–259
 - transformations
 - relationship to optimization, 135
 - using copy-propagation transformations, 137–138
 - transparent_union attribute, purpose of, 130
 - trigraphs command-line option, description of, 70, 323
 - troubleshooting. *See* GCC problems
 - Tru64 #pragmas, features of, 134
 - types, using `typeof` to refer to, 114–115

U

- u GNU Info navigation command, description of, 268, 272
- U *name* preprocessor option, description of, 343
- u *sym* link option, description of, 83
- u *symbol* linker option, description of, 337
- uClibc micro C library, using as alternative to Glibc, 366
- undef preprocessor option, description of, 77, 343
- underscores (`_`), using with `typeof`, 115
- uninstall Makefile
 - producing with automake, 208
 - target for, 19
- uninstall mode for Libtool, working in, 239
- Unix Curses Emulator DJGPP
 - component, description of, 51
- Unix, development of, 187–188
- unmoderated newsgroup, explanation of, 348
- unstageN makefile target, description of, 19
- UNSUPPORTED test suite result code, description of, 22
- unused function attribute, purpose of, 126
- unused.c sample code, 93–94
- unzip32.exe file, using with DJGPP, 52
- Up GNU Info navigation command, description of, 268
- usage command-line option in GNU Info, description of, 279
- Usenet news, selecting software for reading of, 349
- %usuffix predefined substitution spec, description of, 107
- U/WIN project, features of, 57

V

- v argument for dump option, description of, 98–99
- v command-line option
 - description of, 323
 - in gcov program, 155
 - in gprof, 172
- V configure script option, description of, 219
- v option, description of, 61, 343
- V *ver* option, description of, 61
- V *version* command-line option, description of, 323
- %v1-v3 predefined substitution specs, descriptions of, 107

- V850 options, overview of, 491–492
- variable attributes, specifying, 127–130
- VAX options, overview of, 492
- VCG (Visualization of Compiler Graphs), Web address for, 99
- verbose modes, using with GCC and other toolchains, 252
- version command-line option for Libtool, description of, 233
- version option, description of, 61–62
- version| preprocessor option, description of, 343
- version-info *current[]/[]* mode for Libtool, working in, 238
- vi-keys command-line option in GNU Info, description of, 279

W

- W and -w warning options, descriptions of, 87–93, 323–324, 343
- w argument for dump option, description of, 98
- %w predefined substitution specs, descriptions of, 107
- w *width* command-line option in gprof, description of, 172
- Wabi command-line option, description of, 324
- Waggregate-return command-line option, description of, 324
- Wall command-line option, description of, 30, 87, 324–325
- Wa,*option* GCC assembler option, description of, 336
- warning messages
 - displaying during build process, 18
 - enabling and disabling, 86–94
 - versus errors, 18
 - troubleshooting, 258–259
- WARNING test suite result code, description of, 22
- Wbad-function-cast command-line function, description of, 325
- Wcast* command-line functions, descriptions of, 325
- Wchar-subscripts command-line function, description of, 325
- Wcomment command-line function, description of, 325
- Wconversion command-line option, description of, 326
- Wctor-dtor-privacy command-line option, description of, 326
- Wdisabled-optimization command-line option, description of, 326

- Wdiv-by-zero command-line option, description of, 326
- Web addresses
 - Agent newsreader, 349
 - autoconf home page, 192
 - automake page on GNU and Red Hat sites, 197
 - binutils package, 165
 - The Compiler Resources Page, 358–359
 - Cygwin, 39
 - DJGPP (DJ's GNU Programming Platform), 48–49
 - elinks browsers, 358
 - Expect tool, 20
 - GCC, 358
 - GCC resources, 358–359
 - GCC-release change information, 246
 - Glibc download site, 379
 - Gnatsweb, 355
 - GNU m4 macro preprocessor, 193
 - GNU mailing lists, 352
 - GNU Makefile conventions, 191
 - GNU Press, 359
 - internationalization open standards, 389
 - Knews newsreader, 349
 - Makefile conventions, 191
 - malloc replacements, 250
 - Newlib, 366
 - NewsXpress newsreader, 349
 - Perl home page, 198
 - RSXNTDJ information, 57
 - slrn (s-lang read news) newsreader, 349
 - Tcl (Tool Command Language), 20
 - TimeStorm tool, 46
 - tin (Tass + Iain's) newsreader, 349
 - uClibc project, 366
 - U/WIN project, 57
 - VCG (Visualization of Compiler Graphs), 99
 - Weffc++ command-line option, description of, 326–327
 - Werror* command-line options, descriptions of, 327
 - Wfloat-equal command-line option, description of, 327
 - Wformat* command-line options, descriptions of, 327–328
 - which gcc command, using, 249
 - Wimplicit* command-line options, descriptions of, 328
 - Windows environments
 - using Cygwin in, 45–47
 - using DJGPP in, 56–57
 - windows in GNU Info
 - features of, 270
 - working in, 280–281
 - Winline command-line option, description of, 328
 - with configuration options, descriptions of, 12–16
 - Wlarger-than-*len* command-line option, description of, 328
 - Wlong-long command-line option, description of, 328
 - Wl,*opt* link option, description of, 83
 - Wl,*option* linker option, description of, 337
 - Wmain command-line option, description of, 328
 - Wmissing* command-line option, description of, 329
 - Wmultichar command-line option, description of, 329
 - Wnested-externs command-line option, description of, 329
 - Wno* command-line options, descriptions of, 330–331
 - Wnon* command-line options, descriptions of, 331
 - Wold-style-cast command-line option, description of, 331
 - Woption preprocessor option, description of, 343
 - Woverloaded-virtual command-line option, description of, 332
 - Wpacked command-line option, description of, 331
 - Wpadded command-line option, description of, 331
 - Wparentheses command-line option, description of, 332
 - Wpointer-arith command-line option, description of, 332
 - Wp,*option* preprocessor option, description of, 343
 - Wredundant-decls command-line option, description of, 332
 - Wreorder command-line option, description of, 332
 - Wreturn-type command-line option, description of, 332
 - Wselector command-line option, description of, 332
 - Wsequence command-line option, description of, 333
 - Wshadow command-line option, description of, 333
 - Wsign* command-line options, descriptions of, 333

- Wstrict-prototypes command-line option, description of, 333
- Wswitch command-line option, description of, 333
- Wsynth command-line option, description of, 333
- Wsystem-headers command-line option, description of, 334
- Wtraditional command-line option, description of, 334
- Wtrigraphs command-line option, description of, 335
- Wun* command-line options, descriptions of, 335–336

X

- X and x arguments for dump option, descriptions of, 98
- x command-line option in gprof, description of, 172
- x *lang*
 - output option, 64
 - preprocessor option, 343
- x none output option, description of, 64
- x option, example of, 66–68
- Xa option, passing arguments to assembler with, 252
- XFAIL test suite result code, description of, 22
- XL option, passing arguments to linker with, 252

- Xlinker,*opt* link option, description of, 83
- Xlinker,*option* linker option, description of, 337
- Xp option, passing options to preprocessor with, 252
- XPASS test suite result code, description of, 22
- XPG, Glibc support for, 365
- xrefs (cross-references), following in GNU Info, 276–277
- Xtensa options, overview of, 492–496

Y

- Y and y arguments for dump option, descriptions of, 98
- y command-line option in gprof, description of, 172

Z

- Z [*symbol-specification*] command-line option in gprof, description of, 173
- z argument for dump option, description of, 98
- z command-line option in gprof, description of, 172
- zero-length arrays, using with C and extensions, 115–117
- Zip Picker, using with DJGPP, 50, 53



forums.apress.com

FOR PROFESSIONALS BY PROFESSIONALS™

JOIN THE APRESS FORUMS AND BE PART OF OUR COMMUNITY. You'll find discussions that cover topics of interest to IT professionals, programmers, and enthusiasts just like you. If you post a query to one of our forums, you can expect that some of the best minds in the business—especially Apress authors, who all write with *The Expert's Voice™*—will chime in to help you. Why not aim to become one of our most valuable participants (MVPs) and win cool stuff? Here's a sampling of what you'll find:

DATABASES

Data drives everything.

Share information, exchange ideas, and discuss any database programming or administration issues.

PROGRAMMING/BUSINESS

Unfortunately, it is.

Talk about the Apress line of books that cover software methodology, best practices, and how programmers interact with the "suits."

INTERNET TECHNOLOGIES AND NETWORKING

Try living without plumbing (and eventually IPv6).

Talk about networking topics including protocols, design, administration, wireless, wired, storage, backup, certifications, trends, and new technologies.

WEB DEVELOPMENT/DESIGN

Ugly doesn't cut it anymore, and CGI is absurd.

Help is in sight for your site. Find design solutions for your projects and get ideas for building an interactive Web site.

JAVA

We've come a long way from the old Oak tree.

Hang out and discuss Java in whatever flavor you choose: J2SE, J2EE, J2ME, Jakarta, and so on.

SECURITY

Lots of bad guys out there—the good guys need help.

Discuss computer and network security issues here. Just don't let anyone else know the answers!

MAC OS X

All about the Zen of OS X.

OS X is both the present and the future for Mac apps. Make suggestions, offer up ideas, or boast about your new hardware.

TECHNOLOGY IN ACTION

Cool things. Fun things.

It's after hours. It's time to play. Whether you're into LEGO® MINDSTORMS™ or turning an old PC into a DVR, this is where technology turns into fun.

OPEN SOURCE

Source code is good; understanding (open) source is better.

Discuss open source technologies and related topics such as PHP, MySQL, Linux, Perl, Apache, Python, and more.

WINDOWS

No defenestration here.

Ask questions about all aspects of Windows programming, get help on Microsoft technologies covered in Apress books, or provide feedback on any Apress Windows book.

HOW TO PARTICIPATE:

Go to the Apress Forums site at <http://forums.apress.com/>.

Click the New User link.