Beginning Linux® Programming
4th Edition

Neil Matthew
Richard Stones

Wiley Publishing, Inc.

Beginning Linux® Programming 4th Edition

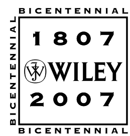
Acknowledgements x
Foreword xxiii
Introduction xxv
Chapter 1: Getting Started 1
Chapter 2: Shell Programming 17
Chapter 3: Working with Files 93
Chapter 4: The Linux Environment 137
Chapter 5: Terminals 175
Chapter 6: Managing Text-Based Screens with curses 211
Chapter 7: Data Management 255
Chapter 8: MySQL 311
Chapter 9: Development Tools 377
Chapter 10: Debugging 429
Chapter 11: Processes and Signals 461
Chapter 12: POSIX Threads 495
Chapter 13: Inter-Process Communication: Pipes 525
Chapter 14: Semaphores, Shared Memory, and Message Queues. 577
Chapter 15: Sockets 607
Chapter 16: Programming GNOME Using GTK+ 645
Chapter 17: Programming KDE Using Qt 701
Chapter 18: Standards for Linux 747
Index 761

Beginning
Linux® Programming
4th Edition

Beginning Linux® Programming

4th Edition

Neil Matthew
Richard Stones



Wiley Publishing, Inc.

Beginning Linux® Programming, 4th Edition

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2008 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-14762-7

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data is available from the publisher.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at http://www.wiley.com/go/permissions.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Linux is a trademark of Linus Torvalds. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

About the Authors

Neil Matthew has been interested in and has programmed computers since 1974. A mathematics graduate from the University of Nottingham, Neil is just plain keen on programming languages and likes to explore new ways of solving computing problems. He's written systems to program in BCPL, FP (Functional Programming), Lisp, Prolog, and a structured BASIC. He even wrote a 6502 microprocessor emulator to run BBC microcomputer programs on UNIX systems.

In terms of UNIX experience, Neil has used almost every flavor since the late 1970s, including BSD UNIX, AT&T System V, Sun Solaris, IBM AIX, many others, and of course Linux. He can claim to have been using Linux since August 1993 when he acquired a floppy disk distribution of Soft Landing (SLS) from Canada, with kernel version 0.99.11. He's used Linux-based computers for hacking C, C++, Icon, Prolog, Tcl, and Java at home and at work.

All of Neil's "home" projects are developed using Linux. He says Linux is much easier because it supports quite a lot of features from other systems, so that both BSD- and System V-targeted programs will generally compile with little or no change.

Neil is currently working as an Enterprise Architect specializing in IT strategy at Celesio AG. He has a background in technical consultancy, software development techniques, and quality assurance. Neil has also programmed in C and C++ for real-time embedded systems.

Neil is married to Christine and has two children, Alexandra and Adrian. He lives in a converted barn in Northamptonshire, England. His interests include solving puzzles by computer, music, science fiction, squash, mountain biking, and not doing it yourself.

Rick Stones started programming at school (more years ago than he cares to remember) on a 6502-powered BBC micro, which, with the help of a few spare parts, continued to function for the next 15 years. He graduated from Nottingham University with a degree in Electronic Engineering, but decided software was more fun.

Over the years he has worked for a variety of companies, from the very small with just a dozen employees, to the very large, including the IT services giant EDS. Along the way he has worked on a range of projects, from real-time communications to accounting systems, to very large help desk systems. He is currently working as an IT architect, acting as a technical authority on various major projects for a large pan-European company.

A bit of a programming linguist, he has programmed in various assemblers, a rather neat proprietary telecommunications language called SL-1, some FORTRAN, Pascal, Perl, SQL, and smidgeons of Python and C++, as well as C. (Under duress he even admits that he was once reasonably proficient in Visual Basic, but tries not to advertise this aberration.)

Rick lives in a village in Leicestershire, England, with his wife Ann, children Jennifer and Andrew, and a cat. Outside work his main interests are classical music, especially early religious music, and photography, and he does his best to find time for some piano practice.

Credits

Acquisitions Editor
Jenny Watson

Development Editor
Sara Shlaer

Technical Editor
Timothy Boronczyk

Production Editor
William A. Barton

Copy Editor
Kim Cofer

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Joseph B. Wikert

Project Coordinator, Cover
Adrienne Martinez

Graphics and Production Specialists
Mike Park, Happenstance-Type-O-Rama
Craig Woods, Happenstance-Type-O-Rama

Proofreader
Amy McCarthy, Word One

Indexer
Johnna VanHoose Dinse

Anniversary Logo Design
Richard Pacifico

Acknowledgments

The authors would like to record their thanks to the many people who helped to make this book possible.

Neil would like to thank his wife, Christine, for her understanding and children Alex and Adrian for not complaining too loudly at Dad spending so long in The Den writing.

Rick would like to thank his wife, Ann, and their children, Jennifer and Andrew, for their very considerable patience during the evenings and weekends while Dad was yet again "doing book work."

As for the publishing team, we'd like to thank the folks at Wiley who helped us get this fourth edition into print. Thanks to Carol Long for getting the process started and sorting out the contracts, and especially to Sara Shlaer for her exceptional editing work and Timothy Boronczyk for his excellent technical reviews. We also wish to thank Jenny Watson for chasing down all those odd bits of extras and generally guiding the book through the administrative layers, Bill Barton for ensuring proper organization and presentation, and Kim Cofer for a thorough copyedit. We are very grateful also to Eric Foster-Johnson for his fantastic work on Chapters 16 and 17. We can say that this is a better book than it would have been without the efforts of all of you.

We would also like to thank our employers, Scientific Generics, Mobicom, and Celesio for their support during the production of all four editions of this book.

Finally we would also like to pay homage to two important motivators who have helped make this book possible. Firstly, Richard Stallman for the excellent GNU tools and the idea of a free software environment, which is now a reality with GNU/Linux, and secondly, Linus Torvalds for starting and continuing to inspire the co-operative development that gives us the ever-improving Linux kernel.

Contents

Acknowledgements x
Foreword xxiii
Introduction xxv

Chapter 1: Getting Started 1

An Introduction to UNIX, Linux, and GNU **1**
What Is UNIX? 1
What Is Linux? 3
The GNU Project and the Free Software Foundation 3
Linux Distributions 4
Programming Linux **4**
Linux Programs 5
Text Editors 6
The C Compiler 7
Development System Roadmap 8
Getting Help **14**
Summary **16**

Chapter 2: Shell Programming 17

Why Program with a Shell? **18**
A Bit of Philosophy **18**
What Is a Shell? **19**
Pipes and Redirection **21**
Redirecting Output 21
Redirecting Input 22
Pipes 22
The Shell as a Programming Language **23**
Interactive Programs 23
Creating a Script 24
Making a Script Executable 25
Shell Syntax **27**
Variables 27
Conditions 31
Control Structures 34
Functions 46
Commands 49
Command Execution 68

Here Documents 73
Debugging Scripts 74
Going Graphical — The dialog Utility 75
Putting It All Together 81
Requirements 82
Design 82
Summary 91

Chapter 3: Working with Files 93

Linux File Structure 94
Directories 94
Files and Devices 95
System Calls and Device Drivers 96
Library Functions 97
Low-Level File Access 98
write 98
read 99
open 100
Initial Permissions 101
Other System Calls for Managing Files 106
The Standard I/O Library 109
fopen 110
fread 110
fwrite 111
fclose 111
fflush 111
fseek 112
fgetc, getc, and getchar 112
fputc, putc, and putchar 112
fgets and gets 113
Formatted Input and Output 113
printf, fprintf, and sprintf 113
scanf, fscanf, and sscanf 115
Other Stream Functions 117
Stream Errors 119
Streams and File Descriptors 119
File and Directory Maintenance 120
chmod 120
chown 120
unlink, link, and symlink 121
mkdir and rmdir 121
chdir and getcwd 122

Scanning Directories **122**
opendir 123
readdir 123
telldir 123
seekdir 124
closedir 124
Errors **127**
strerror 127
perror 127
The /proc File System **128**
Advanced Topics: fcntl and mmap **132**
fcntl 132
mmap 133
Summary **135**

Chapter 4: The Linux Environment **137**

Program Arguments **137**
getopt 140
getopt_long 142
Environment Variables **144**
Use of Environment Variables 146
The environ Variable 147
Time and Date **148**
Temporary Files **156**
User Information **158**
Host Information **161**
Logging **163**
Resources and Limits **167**
Summary **173**

Chapter 5: Terminals **175**

Reading from and Writing to the Terminal **175**
Talking to the Terminal **180**
The Terminal Driver and the General Terminal Interface **182**
Overview 183
Hardware Model 183
The termios Structure **184**
Input Modes 186
Output Modes 186
Control Modes 187
Local Modes 188

Special Control Characters 188
Terminal Speed 192
Additional Functions 192
Terminal Output 196
Terminal Type 197
Identify Your Terminal Type 197
Using terminfo Capabilities 200
Detecting Keystrokes 205
Virtual Consoles 207
Pseudo-Terminals 208
Summary 209

Chapter 6: Managing Text-Based Screens with curses 211

Compiling with curses 212
Curses Terminology and Concepts 213
The Screen 216
Output to the Screen 216
Reading from the Screen 217
Clearing the Screen 218
Moving the Cursor 218
Character Attributes 218
The Keyboard 221
Keyboard Modes 221
Keyboard Input 222
Windows 224
The WINDOW Structure 224
Generalized Functions 225
Moving and Updating a Window 225
Optimizing Screen Refreshes 229
Subwindows 230
The Keypad 232
Using Color 235
Redefining Colors 238
Pads 238
The CD Collection Application 240
Starting a New CD Collection Application 240
Looking at main 243
Building the Menu 243
Database File Manipulation 245
Querying the CD Database 250
Summary 254

Chapter 7: Data Management 255

Managing Memory 255
Simple Memory Allocation 256
Allocating Lots of Memory 257
Abusing Memory 260
The Null Pointer 261
Freeing Memory 262
Other Memory Allocation Functions 264
File Locking 264
Creating Lock Files 265
Locking Regions 268
Use of read and write with Locking 271
Competing Locks 276
Other Lock Commands 280
Deadlocks 280
Databases 281
The dbm Database 281
The dbm Routines 283
dbm Access Functions 283
Additional dbm Functions 287
The CD Application 289
Updating the Design 289
The CD Database Application Using dbm 290
Summary 309

Chapter 8: MySQL 311

Installation 312
MySQL Packages 312
Post-Install Configuration 314
Post-Installation Troubleshooting 319
MySQL Administration 320
Commands 320
Creating Users and Giving Them Permissions 325
Passwords 327
Creating a Database 328
Data Types 329
Creating a Table 330
Graphical Tools 333
Accessing MySQL Data from C 335
Connection Routines 337
Error Handling 341

Executing SQL Statements 342
Miscellaneous Functions 357
The CD Database Application **358**
Creating the Tables 359
Adding Some Data 362
Accessing the Application Data from C 364
Summary **375**

Chapter 9: Development Tools **377**

Problems of Multiple Source Files **377**
The make Command and Makefiles **378**
The Syntax of Makefiles 378
Options and Parameters to make 379
Comments in a Makefile 382
Macros in a Makefile 382
Multiple Targets 384
Built-in Rules 387
Suffix and Pattern Rules 388
Managing Libraries with make 389
Advanced Topic: Makefiles and Subdirectories 391
GNU make and gcc 391
Source Code Control **392**
RCS 393
SCCS 399
Comparing RCS and SCCS 399
CVS 400
CVS Front Ends 404
Subversion 405
Writing a Manual Page **406**
Distributing Software **409**
The patch Program 410
Other Distribution Utilities 411
RPM Packages **413**
Working with RPM Package Files 414
Installing RPM Packages 415
Building RPM Packages 415
Other Package Formats **424**
Development Environments **424**
KDevelop 425
Other Environments 425
Summary **427**

Chapter 10: Debugging 429

Types of Errors 429
General Debugging Techniques 430
A Program with Bugs 430
Code Inspection 433
Instrumentation 434
Controlled Execution 436
Debugging with gdb 437
Starting gdb 437
Running a Program 438
Stack Trace 438
Examining Variables 439
Listing the Program 440
Setting Breakpoints 441
Patching with the Debugger 444
Learning More about gdb 445
More Debugging Tools 445
Lint: Removing the Fluff from Your Programs 446
Function Call Tools 449
Execution Profiling with prof/gprof 451
Assertions 452
Memory Debugging 453
ElectricFence 454
valgrind 455
Summary 459

Chapter 11: Processes and Signals 461

What Is a Process? 461
Process Structure 462
The Process Table 463
Viewing Processes 463
System Processes 464
Process Scheduling 467
Starting New Processes 468
Waiting for a Process 475
Zombie Processes 477
Input and Output Redirection 479
Threads 480

Signals **481**
Sending Signals 484
Signal Sets 489
Summary **493**

Chapter 12: POSIX Threads **495**

What Is a Thread? **495**
Advantages and Drawbacks of Threads **496**
A First Threads Program **497**
Simultaneous Execution **501**
Synchronization **503**
Synchronization with Semaphores 503
Synchronization with Mutexes 508
Thread Attributes **512**
Canceling a Thread **517**
Threads in Abundance **520**
Summary **524**

Chapter 13: Inter-Process Communication: Pipes **525**

What Is a Pipe? **525**
Process Pipes **526**
Sending Output to popen **528**
Passing More Data 529
How popen Is Implemented 530
The Pipe Call **531**
Parent and Child Processes **535**
Reading Closed Pipes 536
Pipes Used as Standard Input and Output 537
Named Pipes: FIFOs **540**
Accessing a FIFO 542
Advanced Topic: Client/Server Using FIFOs 549
The CD Database Application **553**
Aims 554
Implementation 555
Client Interface Functions 558
The Server Interface, server.c 565
The Pipe 569
Application Summary 574
Summary **575**

Chapter 14: Semaphores, Shared Memory, and Message Queues 577

Semaphores 577
Semaphore Definition 579
A Theoretical Example 579
Linux Semaphore Facilities 580
Using Semaphores 582
Shared Memory 586
shmget 588
shmat 588
shmdt 589
shmctl 589
Message Queues 594
msgget 594
msgsnd 595
msgrcv 595
msgctl 596
The CD Database Application 599
Revising the Server Functions 600
Revising the Client Functions 602
IPC Status Commands 604
Displaying Semaphore Status 604
Displaying Shared Memory Status 604
Displaying Message Queue Status 605
Summary 605

Chapter 15: Sockets 607

What Is a Socket? 608
Socket Connections 608
Socket Attributes 612
Creating a Socket 614
Socket Addresses 615
Naming a Socket 616
Creating a Socket Queue 617
Accepting Connections 617
Requesting Connections 618
Closing a Socket 619
Socket Communications 619
Host and Network Byte Ordering 622
Network Information 624
The Internet Daemon (xinetd/inetd) 629
Socket Options 631

Multiple Clients **632**
select 635
Multiple Clients 638
Datagrams **642**
Summary **644**

Chapter 16: Programming GNOME Using GTK+ **645**

Introducing X **645**
X Server 646
X Client 646
X Protocol 646
Xlib 647
Toolkits 647
Window Managers 647
Other Ways to Create a GUI — Platform-Independent Windowing APIs 648
Introducing GTK+ **648**
GLib Type System 649
GTK+ Object System 650
Introducing GNOME 651
Installing the GNOME/GTK+ Development Libraries 652
Events, Signals, and Callbacks **655**
Packing Box Widgets **658**
GTK+ Widgets **661**
GtkWindow 662
GtkEntry 663
GtkSpinButton 666
GtkButton 668
GtkTreeView 672
GNOME Widgets **676**
GNOME Menus **677**
Dialogs **682**
GtkDialog 682
Modal Dialog Box 684
Nonmodal Dialogs 685
GtkMessageDialog 686
CD Database Application **687**
Summary **699**

Chapter 17: Programming KDE Using Qt **701**

Introducing KDE and Qt **701**
Installing Qt **702**
Signals and Slots **705**
Qt Widgets **712**
QLineEdit 712
Qt Buttons 716
QComboBox 721
QListView 724
Dialogs **727**
QDialog 728
QMessageBox 730
QInputDialog 731
Using qmake to Simplify Writing Makefiles 733
Menus and Toolbars with KDE **733**
CD Database Application Using KDE/Qt **738**
MainWindow 738
AddCdDialog 742
LogonDialog 743
main.cpp 745
Summary **746**

Chapter 18: Standards for Linux **747**

The C Programming Language **748**
A Brief History Lesson 748
The GNU Compiler Collection 749
gcc Options 749
Interfaces and the Linux Standards Base **751**
LSB Standard Libraries 752
LSB Users and Groups 754
LSB System Initialization 754
The Filesystem Hierarchy Standard **755**
Further Reading about Standards **758**
Summary **759**

Index **761**

Foreword

All computer programmers have their own piles of notes and scribbles. They have their code examples saved from the past heroic dive into the manuals or from Usenet, where sometimes even fools fear to follow. (The other body of opinion is that fools all get free Usenet access and use it nonstop.) It is therefore perhaps strange that so few books follow such a style. In the online world there are a lot of short, to-the-point documents about specific areas of programming and administration. The Linux documentation project released a whole pile of documents covering everything from installing Linux and Windows on the same machine to wiring your coffee machine to Linux. Seriously. Take a look at The Linux Documentation Project on `http://www.tldp.org`.

The book world, on the other hand, seems to consist mostly of either learned tomes, detailed and very complete works that you don't have time to read, or books for complete beginners that you buy for friends as a joke. There are very few books that try to cover the basics of a lot of useful areas. This book is one of them, a compendium of those programmers' notes and scribbles, deciphered (try reading a programmer's handwriting), edited, and brought together coherently as a book.

This edition of *Beginning Linux Programming* has been reviewed and updated to reflect today's Linux developments.

—*Alan Cox*

Introduction

Welcome to *Beginning Linux Programming*, 4th Edition, an easy-to-use guide to developing programs for Linux and other UNIX-style operating systems.

In this book we aim to give you an introduction to a wide variety of topics important to you as a developer using Linux. The word *Beginning* in the title refers more to the content than to your skill level. We've structured the book to help you learn more about what Linux has to offer, however much experience you have already. Linux programming is a large field and we aim to cover enough about a wide range of topics to give you a good "beginning" in each subject.

Who's This Book For?

If you're a programmer who wishes to get up to speed with the facilities that Linux (or UNIX) offers software developers, to maximize your programming time and your application's use of the Linux system, you've picked up the right book. Clear explanations and a tried and tested step-by-step approach will help you progress rapidly and pick up all the key techniques.

We assume you have some experience in C and/or C++ programming, perhaps in Windows or some other system, but we try to keep the book's examples simple so that you don't need to be an expert C coder to follow this book. Where direct comparisons exist between Linux programming and C/C++ programming, these are indicated in the text.

Watch out if you're totally new to Linux. This isn't a book on installing or configuring Linux. If you want to learn more about administering a Linux system, you may wish to look at some complementary books such as *Linux Bible 2007 Edition*, by Christopher Negus (Wiley, ISBN 978-0470082799).

Because it aims to be a tutorial guide to the various tools and sets of functions/libraries available to you on most Linux systems as well as a handy reference you can return to, this book is unique in its straightforward approach, comprehensive coverage, and extensive examples.

What's Covered in the Book

The book has a number of aims:

- To teach the use of the standard Linux C libraries and other facilities as specified by the various Linux and UNIX standards.
- To show how to make the most of the standard Linux development tools.

- To give a concise introduction to data storage under Linux using both the DBM and MySQL database systems.
- To show how to build graphical user interfaces for the X Window System. We will use both the GTK (the basis of the GNOME environment) and Qt (the basis of the KDE environment) libraries.
- To encourage and enable you to develop your own real-world applications.

As we cover these topics, we introduce programming theory and then illustrate it with appropriate examples and a clear explanation. In this way you can learn quickly on a first read and look back over things to brush up on all the essential elements if you need to.

Though the small examples are designed mainly to illustrate a set of functions or some new theory in action, throughout the book lies a larger sample project: a simple database application for recording audio CD details. As your knowledge expands, you can develop, re-implement, and extend the project to your heart's content. That said, however, the CD application doesn't dominate any chapter, so you can skip it if you want to, but we feel that it provides additional useful, in-depth examples of the techniques that we discuss. It certainly provides an ideal way to illustrate each of the more advanced topics as they are introduced. Our first discussion of this application occurs at the end of Chapter 2 and shows how a fairly large shell script is organized, how the shell deals with user input, and how it can construct menus and store and search data.

After recapping the basic concepts of compiling programs, linking to libraries, and accessing the online manuals, you will take a sojourn into shells. You then move into C programming, where we cover working with files, getting information from the Linux environment, dealing with terminal input and output, and the `curses` library (which makes interactive input and output more tractable). You're then ready to tackle re-implementing the CD application in C. The application design remains the same, but the code uses the `curses` library for a screen-based user interface.

From there, we cover data management. Meeting the `dbm` database library is sufficient cause for us to re-implement the application, but this time with a design that will re-emerge in some later chapters. In a later chapter we look at how the data could be stored in a relational database using MySQL, and we also reuse this data storage technique later in the chapter, so you can see how the techniques compare. The size of these recent applications means that we then need to deal with such nuts-and-bolts issues as debugging, source code control, software distribution, and makefiles.

You will also look at how different Linux processes can communicate, using a variety of techniques, and at how Linux programs can use sockets to support TCP/IP networking to different machines, including the issues of talking to machines that use different processor architectures.

After getting the foundations of Linux programming in place, we cover the creation of graphical programs. We do this over two chapters, looking first at the GTK+ toolkit, which underlies the GNOME environment, and then at the Qt toolkit, which underlies the KDE environment.

We finish off with a brief look at the standards that keep Linux systems from different vendors similar enough that we can move between them easily and write programs that will work on different distributions of Linux.

As you'd expect, there's a fair bit more in between, but we hope that this gives you a good idea of the material we'll be discussing.

What You Need to Use This Book

In this book, we'll give you a taste of programming for Linux. To help you get the most from the chapters, you should try out the examples as you read. These also provide a good base for experimentation and will hopefully inspire you to create programs of your own. We hope you will read this book in conjunction with experimenting on your own Linux installation.

Linux is available for many different systems. Its adaptability is such that enterprising souls have persuaded it to run in one form or another on just about anything with a processor in it! Examples include systems based on the Alpha, ARM, IBM Cell, Itanium, PA-RISC, PowerPC, SPARC, SuperH, and 68k CPUs as well as the various *x*86-class processors, in both 32- and 64-bit versions.

We wrote this book and developed the examples on two Linux systems with different specifications, so we're confident that if you can run Linux, you can make good use of this book. Furthermore, we tested the code on other versions of Linux during the book's technical review.

To develop this book we primarily used *x*86-based systems, but very little of what we cover is *x*86 specific. Although it is possible to run Linux on a 486 with 8MB RAM, to run a modern Linux distribution successfully and follow the examples in this book, we recommend that you pick a recent version of one of the more popular Linux distributions such as Fedora, openSUSE, or Ubuntu and check the hardware recommendations they give.

As for software requirements, we suggest that you use a recent version of your preferred Linux distribution and apply the current set of updates, which most vendors make available online by way of automated updates, to keep your system current and up-to-date with the latest bug fixes. Linux and the GNU toolset are released under the GNU General Public License (GPL). Most other components of a typical Linux distribution use either the GPL or one of the many other Open Source licenses, and this means they have certain properties, one of which is freedom. They will always have the source code available, and no one can take that freedom away. See `http://www.gnu.org/licenses/` for more details of the GPL, and `http://www.opensource.org/` for more details of the definition of Open Source and the different licenses in use. With GNU/Linux, you will always have the option of support — either doing it yourself with the source code, hiring someone else, or going to one of the many vendors offering pay-for support.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at `http://www.wrox.com`. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-14762-7.

Once you download the code, just decompress it with your favorite compression tool. Alternatively, you can go to the main Wrox code download page at `http://www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

A Note on the Code Downloads

We have tried to provide example programs and code snippets that best illustrate the concepts being discussed in the text. Please note that, in order to make the new functionality being introduced as clear as possible, we have taken one or two liberties with coding style.

In particular, we do not always check that the return results from every function we call are what we expect. In production code for real applications we would certainly do this check, and you too should adopt a rigorous approach toward error handling. (We discuss some of the ways that errors can be caught and handled in Chapter 3.)

The GNU General Public License

The source code in the book is made available under the terms of the GNU General Public License version 2, `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`. The following permission statement applies to all the source code available in this book:

```
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book:

> **Boxes like this one hold important, not-to-be-forgotten, mission-critical information that is directly relevant to the surrounding text.**

Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

When we introduce them, we highlight *important words* in italics. Characters we want you to **type** are in bold font. We show keyboard strokes like this: Ctrl+A.

We present code and terminal sessions in three different ways:

```
$ who
root     tty1         Sep 10 16:12
rick     tty2         Sep 10 16:10
```

When the command line is shown, it's in the style at the top of the code, whereas output is in the regular style. The $ is the prompt (if the superuser is required for the command, the prompt will be a # instead) and the bold text is what you type in and press Enter (or Return) to execute. Any text following that in the same font but in non-bold is the output of the bolded command. In the preceding example you type in the command `who`, and you see the output below the command.

Prototypes of Linux-defined functions and structures are shown in bold as follows:

```
#include <stdio.h>

int printf (const char *format, ...);
```

In our code examples, the code foreground style shows new, important material, such as

```
/* This is what new, important, and pertinent code looks like. */
```

whereas code that looks like this (code background style) is less important:

```
/* This is what code that has been seen before looks like. */
```

And often when a program is added to throughout a chapter, code that is added later is in foreground style first and background style later. For example, a new program would look like this:

```
/* Code example */
/* That ends here. */
```

And if we add to that program later in the chapter, it looks like this instead:

```
/* Code example */
/* New code added */
/* on these lines */
/* That ends here. */
```

The last convention we'll mention is that we presage example code with a "Try It Out" heading that aims to split the code up where it's helpful, highlight the component parts, and show the progression of the application. When it's important, we also follow the code with a "How It Works" section to explain any salient points of the code in relation to previous theory. We find these two conventions help break up the more formidable code listings into palatable morsels.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty

piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to `http://www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com` you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Getting Started

In this chapter, you discover what Linux is and how it relates to its inspiration, UNIX. You take a guided tour of the facilities provided by a Linux development system, and write and run your first program. Along the way, you'll be looking at

- UNIX, Linux, and GNU
- Programs and programming languages for Linux
- How to locate development resources
- Static and shared libraries
- The UNIX philosophy

An Introduction to UNIX, Linux, and GNU

In recent years Linux has become a phenomenon. Hardly a day goes by without Linux cropping up in the media in some way. We've lost count of the number of applications that have been made available on Linux and the number of organizations that have adopted it, including some government departments and city administrations. Major hardware vendors like IBM and Dell now support Linux, and major software vendors like Oracle support their software running on Linux. Linux truly has become a viable operating system, especially in the server market.

Linux owes its success to systems and applications that preceded it: UNIX and GNU software. This section looks at how Linux came to be and what its roots are.

What Is UNIX?

The UNIX operating system was originally developed at Bell Laboratories, once part of the telecommunications giant AT&T. Designed in the 1970s for Digital Equipment PDP computers, UNIX has become a very popular multiuser, multitasking operating system for a wide variety of hardware platforms, from PC workstations to multiprocessor servers and supercomputers.

A Brief History of UNIX

Strictly, UNIX is a trademark administered by The Open Group, and it refers to a computer operating system that conforms to a particular specification. This specification, known as The Single UNIX Specification, defines the names of, interfaces to, and behaviors of all mandatory UNIX operating system functions. The specification is largely a superset of an earlier series of specifications, the P1003, or POSIX (Portable Operating System Interface) specifications, developed by the IEEE (Institute of Electrical and Electronic Engineers).

Many UNIX-like systems are available commercially, such as IBM's AIX, HP's HP-UX, and Sun's Solaris. Some have been made available for free, such as FreeBSD and Linux. Only a few systems currently conform to The Open Group specification, which allows them to be marketed with the name UNIX.

In the past, compatibility among different UNIX systems has been a real problem, although POSIX was a great help in this respect. These days, by following a few simple rules it is possible to create applications that will run on all UNIX and UNIX-like systems. You can find more details on Linux and UNIX standards in Chapter 18.

UNIX Philosophy

In the following chapters we hope to convey a flavor of Linux (and therefore UNIX) programming. Although programming in C is in many ways the same whatever the platform, UNIX and Linux developers have a special view of program and system development.

The UNIX operating system, and hence Linux, encourages a certain programming style. Following are a few characteristics shared by typical UNIX programs and systems:

- **Simplicity:** Many of the most useful UNIX utilities are very simple and, as a result, small and easy to understand. KISS, "Keep It Small and Simple," is a good technique to learn. Larger, more complex systems are guaranteed to contain larger, more complex bugs, and debugging is a chore that we'd all like to avoid!
- **Focus:** It's often better to make a program perform one task well than to throw in every feature along with the kitchen sink. A program with "feature bloat" can be difficult to use and difficult to maintain. Programs with a single purpose are easier to improve as better algorithms or interfaces are developed. In UNIX, small utilities are often combined to perform more demanding tasks when the need arises, rather than trying to anticipate a user's needs in one large program.
- **Reusable Components:** Make the core of your application available as a library. Well-documented libraries with simple but flexible programming interfaces can help others to develop variations or apply the techniques to new application areas. Examples include the `dbm` database library, which is a suite of reusable functions rather than a single database management program.
- **Filters:** Many UNIX applications can be used as filters. That is, they transform their input and produce output. As you'll see, UNIX provides facilities that allow quite complex applications to be developed from other UNIX programs by combining them in novel ways. Of course, this kind of reuse is enabled by the development methods that we've previously mentioned.
- **Open File Formats:** The more successful and popular UNIX programs use configuration files and data files that are plain ASCII text or XML. If either of these is an option for your program development, it's a good choice. It enables users to use standard tools to change and search for configuration items and to develop new tools for performing new functions on the data files. A good example of this is the `ctags` source code cross-reference system, which records symbol location information as regular expressions suitable for use by searching programs.

- **Flexibility:** You can't anticipate exactly how ingeniously users will use your program. Try to be as flexible as possible in your programming. Try to avoid arbitrary limits on field sizes or number of records. If you can, write the program so that it's network-aware and able to run across a network as well as on a local machine. Never assume that you know everything that the user might want to do.

What Is Linux?

As you may already know, Linux is a freely distributed implementation of a UNIX-like kernel, the low-level core of an operating system. Because Linux takes the UNIX system as its inspiration, Linux and UNIX programs are very similar. In fact, almost all programs written for UNIX can be compiled and run on Linux. Also, some commercial applications sold for commercial versions of UNIX can run unchanged in binary form on Linux systems.

Linux was developed by Linus Torvalds at the University of Helsinki, with the help of UNIX programmers from across the Internet. It began as a hobby inspired by Andy Tanenbaum's Minix, a small UNIX-like system, but has grown to become a complete system in its own right. The intention is that the Linux kernel will not incorporate proprietary code but will contain nothing but freely distributable code.

Versions of Linux are now available for a wide variety of computer systems using many different types of CPUs, including PCs based on 32-bit and 64-bit Intel *x*86 and compatible processors; workstations and servers using Sun SPARC, IBM PowerPC, AMD Opteron, and Intel Itanium; and even some handheld PDAs and Sony's Playstations 2 and 3. If it's got a processor, someone somewhere is trying to get Linux running on it!

The GNU Project and the Free Software Foundation

Linux owes its existence to the cooperative efforts of a large number of people. The operating system kernel itself forms only a small part of a usable development system. Commercial UNIX systems traditionally come bundled with applications that provide system services and tools. For Linux systems, these additional programs have been written by many different programmers and have been freely contributed.

The Linux community (together with others) supports the concept of free software, that is, software that is free from restrictions, subject to the GNU General Public License (the name GNU stands for the recursive *GNU's Not Unix*). Although there may be a cost involved in obtaining the software, it can thereafter be used in any way desired and is usually distributed in source form.

The Free Software Foundation was set up by Richard Stallman, the author of GNU Emacs, one of the best-known text editors for UNIX and other systems. Stallman is a pioneer of the free software concept and started the GNU Project, an attempt to create an operating system and development environment that would be compatible with UNIX, but not suffer the restrictions of the proprietary UNIX name and source code. GNU may one day turn out to be very different from UNIX in the way it handles the hardware and manages running programs, but it will still support UNIX-style applications.

The GNU Project has already provided the software community with many applications that closely mimic those found on UNIX systems. All these programs, so-called GNU software, are distributed under the terms of the GNU General Public License (GPL); you can find a copy of the license at `http://www.gnu.org`. This license embodies the concept of *copyleft* (a takeoff on "copyright"). Copyleft is intended to prevent others from placing restrictions on the use of free software.

A few major examples of software from the GNU Project distributed under the GPL follow:

- GCC: The GNU Compiler Collection, containing the GNU C compiler
- G++: A C++ compiler, included as part of GCC
- GDB: A source code–level debugger
- GNU make: A version of UNIX `make`
- Bison: A parser generator compatible with UNIX `yacc`
- bash: A command shell
- GNU Emacs: A text editor and environment

Many other packages have been developed and released using free software principles and the GPL, including spreadsheets, source code control tools, compilers and interpreters, Internet tools, graphical image manipulation tools such as the Gimp, and two complete object-based environments: GNOME and KDE. We discuss GNOME and KDE in Chapters 16 and 17.

There is now so much free software available that with the addition of the Linux kernel it could be said that the goal of a creating GNU, a free UNIX-like system, has been achieved with Linux. To recognize the contribution made by GNU software, many people now refer to Linux systems in general as GNU/Linux.

You can learn more about the free software concept at `http://www.gnu.org`.

Linux Distributions

As we have already mentioned, Linux is actually just a kernel. You can obtain the sources for the kernel to compile and install it on a machine and then obtain and install many other freely distributed software programs to make a complete Linux installation. These installations are usually referred to as *Linux systems*, because they consist of much more than just the kernel. Most of the utilities come from the GNU Project of the Free Software Foundation.

As you can probably appreciate, creating a Linux system from just source code is a major undertaking. Fortunately, many people have put together ready-to-install distributions (often called *flavors*), usually downloadable or on CD-ROMs or DVDs, that contain not just the kernel but also many other programming tools and utilities. These often include an implementation of the X Window System, a graphical environment common on many UNIX systems. The distributions usually come with a setup program and additional documentation (normally all on the CD[s]) to help you install your own Linux system. Some well-known distributions, particularly on the Intel *x*86 family of processors, are Red Hat Enterprise Linux and its community-developed cousin Fedora, Novell SUSE Linux and the free openSUSE variant, Ubuntu Linux, Slackware, Gentoo, and Debian GNU/Linux. Check out the DistroWatch site at `http://distrowatch.com` for details on many more Linux distributions.

Programming Linux

Many people think that programming Linux means using C. It's true that UNIX was originally written in C and that the majority of UNIX applications are written in C, but C is not the only option available to

Linux programmers, or UNIX programmers for that matter. In the course of the book, we'll mention a couple of the alternatives.

> **In fact, the first version of UNIX was written in PDP 7 assembler language in 1969. C was conceived by Dennis Ritchie around that time, and in 1973 he and Ken Thompson rewrote essentially the entire UNIX kernel in C, quite a feat in the days when system software was written in assembly language.**

A vast range of programming languages are available for Linux systems, and many of them are free and available on CD-ROM collections or from FTP archive sites on the Internet. Here's a partial list of programming languages available to the Linux programmer:

Ada	C	C++
Eiffel	Forth	Fortran
Icon	Java	JavaScript
Lisp	Modula 2	Modula 3
Oberon	Objective C	Pascal
Perl	PostScript	Prolog
Python	Ruby	Smalltalk
PHP	Tcl/Tk	Bourne Shell

We show how you can use a Linux shell (`bash`) to develop small- to medium-sized applications in Chapter 2. For the rest of the book, we mainly concentrate on C. We direct our attention mostly toward exploring the Linux programming interfaces from the perspective of the C programmer, and we assume knowledge of the C programming language.

Linux Programs

Linux applications are represented by two special types of files: *executables* and *scripts*. Executable files are programs that can be run directly by the computer; they correspond to Windows `.exe` files. Scripts are collections of instructions for another program, an interpreter, to follow. These correspond to Windows `.bat` or `.cmd` files, or interpreted BASIC programs.

Linux doesn't require executables or scripts to have a specific filename or any extension whatsoever. File system attributes, which we discuss in Chapter 2, are used to indicate that a file is a program that may be run. In Linux, you can replace scripts with compiled programs (and vice versa) without affecting other programs or the people who call them. In fact, at the user level, there is essentially no difference between the two.

When you log in to a Linux system, you interact with a shell program (often `bash`) that runs programs in the same way that the Windows command prompt does. It finds the programs you ask for by name by

searching for a file with the same name in a given set of directories. The directories to search are stored in a shell variable, `PATH`, in much the same way as with Windows. The search path (to which you can add) is configured by your system administrator and will usually contain some standard places where system programs are stored. These include:

- `/bin`: Binaries, programs used in booting the system
- `/usr/bin`: User binaries, standard programs available to users
- `/usr/local/bin`: Local binaries, programs specific to an installation

An administrator's login, such as `root`, may use a `PATH` variable that includes directories where system administration programs are kept, such as `/sbin` and `/usr/sbin`.

Optional operating system components and third-party applications may be installed in subdirectories of `/opt`, and installation programs might add to your `PATH` variable by way of user install scripts.

> **It's not a good idea to delete directories from `PATH` unless you are sure that you understand what will result if you do.**

Note that Linux, like UNIX, uses the colon (`:`) character to separate entries in the `PATH` variable, rather than the semicolon (`;`) that MS-DOS and Windows use. (UNIX chose `:` first, so ask Microsoft why Windows is different, not why UNIX is different!) Here's a sample `PATH` variable:

```
/usr/local/bin:/bin:/usr/bin:.:/home/neil/bin:/usr/X11R6/bin
```

Here the `PATH` variable contains entries for the standard program locations, the current directory (`.`), a user's home directory, and the X Window System.

> **Remember, Linux uses a forward slash (/) to separate directory names in a filename rather than the backslash (\) of Windows. Again, UNIX got there first.**

Text Editors

To write and enter the code examples in the book, you'll need to use an editor. There are many to choose from on a typical Linux system. The vi editor is popular with many users.

Both of the authors like Emacs, so we suggest you take the time to learn some of the features of this powerful editor. Almost all Linux distributions have Emacs as an optional package you can install, or you can get it from the GNU website at `http://www.gnu.org` or a version for graphical environments at the XEmacs site at `http://www.xemacs.org`.

To learn more about Emacs, you can use its online tutorial. To do this, start the editor by running the `emacs` command, and then type Ctrl+H followed by t for the tutorial. Emacs also has its entire manual available. When in Emacs, type Ctrl+H and then i for information. Some versions of Emacs may have menus that you can use to access the manual and tutorial.

The C Compiler

On POSIX-compliant systems, the C compiler is called `c89`. Historically, the C compiler was simply called `cc`. Over the years, different vendors have sold UNIX-like systems with C compilers with different facilities and options, but often still called `cc`.

When the POSIX standard was prepared, it was impossible to define a standard `cc` command with which all these vendors would be compatible. Instead, the committee decided to create a new standard command for the C compiler, `c89`. When this command is present, it will always take the same options, independent of the machine.

On Linux systems that do try to implement the standards, you might find that any or all of the commands `c89`, `cc`, and `gcc` refer to the system C compiler, usually the GNU C compiler, or `gcc`. On UNIX systems, the C compiler is almost always called `cc`.

In this book, we use `gcc` because it's provided with Linux distributions and because it supports the ANSI standard syntax for C. If you ever find yourself using a UNIX system without `gcc`, we recommend that you obtain and install it. You can find it at `http://www.gnu.org`. Wherever we use `gcc` in the book, simply substitute the relevant command on your system.

Try It Out — Your First Linux C Program

In this example you start developing for Linux using C by writing, compiling, and running your first Linux program. It might as well be that most famous of all starting points, Hello World.

1. Here's the source code for the file `hello.c`:

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

int main()
{
    printf("Hello World\n");
    exit(0);
}
```

2. Now compile, link, and run your program.

```
$ gcc -o hello hello.c
$ ./hello
Hello World
$
```

How It Works

You invoked the GNU C compiler (on Linux this will most likely be available as `cc` too) that translated the C source code into an executable file called `hello`. You ran the program and it printed a greeting. This is just about the simplest example there is, but if you can get this far with your system, you should be able to compile and run the remainder of the examples in the book. If this did not work for you, make sure that the C compiler is installed on your system. For example, many Linux distributions have an install option called Software Development (or something similar) that you should select to make sure the necessary packages are installed.

Because this is the first program you've run, it's a good time to point out some basics. The `hello` program will probably be in your home directory. If `PATH` doesn't include a reference to your home directory, the shell won't be able to find `hello`. Furthermore, if one of the directories in `PATH` contains another program called `hello`, that program will be executed instead. This would also happen if such a directory is mentioned in `PATH` before your home directory. To get around this potential problem, you can prefix program names with `./` (for example, `./hello`). This specifically instructs the shell to execute the program in the current directory with the given name. (The dot is an alias for the current directory.)

If you forget the `-o name` option that tells the compiler where to place the executable, the compiler will place the program in a file called `a.out` (meaning assembler output). Just remember to look for an `a.out` if you think you've compiled a program and you can't find it! In the early days of UNIX, people wanting to play games on the system often ran them as `a.out` to avoid being caught by system administrators, and some UNIX installations routinely delete all files called `a.out` every evening.

Development System Roadmap

For a Linux developer, it can be important to know a little about where tools and development resources are located. The following sections provide a brief look at some important directories and files.

Applications

Applications are usually kept in directories reserved for them. Applications supplied by the system for general use, including program development, are found in `/usr/bin`. Applications added by system administrators for a specific host computer or local network are often found in `/usr/local/bin` or `/opt`.

Administrators favor `/opt` and `/usr/local`, because they keep vendor-supplied files and later additions separate from the applications supplied by the system. Keeping files organized in this way may help when the time comes to upgrade the operating system, because only `/opt` and `/usr/local` need be preserved. We recommend that you compile your applications to run and access required files from the `/usr/local` hierarchy for system-wide applications. For development and personal applications it's best just to use a folder in your `home` directory.

Additional features and programming systems may have their own directory structures and program directories. Chief among these is the X Window System, which is commonly installed in the `/usr/X11` or `/usr/bin/X11` directory. Linux distributions typically use the X.Org Foundation version of the X Window System, based on Revision 7 (X11R7). Other UNIX-like systems may choose different versions of the X Window System installed in different locations, such as `/usr/openwin` for Sun's Open Windows provided with Solaris.

The GNU compiler system's driver program, gcc (which you used in the preceding programming example), is typically located in /usr/bin or /usr/local/bin, but it will run various compiler-support applications from another location. This location is specified when you compile the compiler itself and varies with the host computer type. For Linux systems, this location might be a version-specific subdirectory of /usr/lib/gcc/. On one of the author's machines at the time of writing it is /usr/lib/gcc/i586-suse-linux/4.1.3. The separate passes of the GNU C/C++ compiler, and GNU-specific header files, are stored here.

Header Files

For programming in C and other languages, you need header files to provide definitions of constants and declarations for system and library function calls. For C, these are almost always located in /usr/include and subdirectories thereof. You can normally find header files that depend on the particular incarnation of Linux that you are running in /usr/include/sys and /usr/include/linux.

Other programming systems will also have header files that are stored in directories that get searched automatically by the appropriate compiler. Examples include /usr/include/X11 for the X Window System and /usr/include/c++ for GNU C++.

You can use header files in subdirectories or nonstandard places by specifying the -I flag (for include) to the C compiler. For example,

```
$ gcc -I/usr/openwin/include fred.c
```

will direct the compiler to look in the directory /usr/openwin/include, as well as the standard places, for header files included in the fred.c program. Refer to the manual page for the C compiler (man gcc) for more details.

It's often convenient to use the grep command to search header files for particular definitions and function prototypes. Suppose you need to know the name of the #defines used for returning the exit status from a program. Simply change to the /usr/include directory and grep for a probable part of the name like this:

```
$ grep EXIT_ *.h
...
stdlib.h:#define        EXIT_FAILURE    1       /* Failing exit status.  */
stdlib.h:#define        EXIT_SUCCESS    0       /* Successful exit status.  */
...
$
```

Here grep searches all the files in the directory with a name ending in .h for the string EXIT_. In this example, it has found (among others) the definition you need in the file stdlib.h.

Library Files

Libraries are collections of precompiled functions that have been written to be reusable. Typically, they consist of sets of related functions to perform a common task. Examples include libraries of screen-handling functions (the curses and ncurses libraries) and database access routines (the dbm library). We show you some libraries in later chapters.

Standard system libraries are usually stored in /lib and /usr/lib. The C compiler (or more exactly, the linker) needs to be told which libraries to search, because by default it searches only the standard C library. This is a remnant of the days when computers were slow and CPU cycles were expensive. It's not enough to put a library in the standard directory and hope that the compiler will find it; libraries need to follow a very specific naming convention and need to be mentioned on the command line.

A library filename always starts with lib. Then follows the part indicating what library this is (like c for the C library, or m for the mathematical library). The last part of the name starts with a dot (.), and specifies the type of the library:

- .a for traditional, static libraries
- .so for shared libraries (see the following)

The libraries usually exist in both static and shared formats, as a quick ls /usr/lib will show. You can instruct the compiler to search a library either by giving it the full path name or by using the -l flag. For example,

```
$ gcc -o fred fred.c /usr/lib/libm.a
```

tells the compiler to compile file fred.c, call the resulting program file fred, and search the mathematical library in addition to the standard C library to resolve references to functions. A similar result is achieved with the following command:

```
$ gcc -o fred fred.c -lm
```

The -lm (no space between the l and the m) is shorthand (shorthand is much valued in UNIX circles) for the library called libm.a in one of the standard library directories (in this case /usr/lib). An additional advantage of the -lm notation is that the compiler will automatically choose the shared library when it exists.

Although libraries are usually found in standard places in the same way as header files, you can add to the search directories by using the -L (uppercase letter) flag to the compiler. For example,

```
$ gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11
```

will compile and link a program called x11fred using the version of the library libX11 found in the /usr/openwin/lib directory.

Static Libraries

The simplest form of library is just a collection of object files kept together in a ready-to-use form. When a program needs to use a function stored in the library, it includes a header file that declares the function. The compiler and linker take care of combining the program code and the library into a single executable program. You must use the -l option to indicate which libraries other than the standard C runtime library are required.

Static libraries, also known as *archives*, conventionally have names that end with .a. Examples are /usr/lib/libc.a and /usr/lib/libX11.a for the standard C library and the X11 library, respectively.

You can create and maintain your own static libraries very easily by using the `ar` (for archive) program and compiling functions separately with `gcc -c`. Try to keep functions in separate source files as much as possible. If functions need access to common data, you can place them in the same source file and use static variables declared in that file.

Try It Out Static Libraries

In this example, you create your own small library containing two functions and then use one of them in an example program. The functions are called `fred` and `bill` and just print greetings.

1. First, create separate source files (imaginatively called `fred.c` and `bill.c`) for each function. Here's the first:

```
#include <stdio.h>


void fred(int arg)
{
    printf("fred: we passed %d\n", arg);
}
```

   And here's the second:

```
#include <stdio.h>


void bill(char *arg)
{
    printf("bill: we passed %s\n", arg);
}
```

2. You can compile these functions individually to produce object files ready for inclusion into a library. Do this by invoking the C compiler with the `-c` option, which prevents the compiler from trying to create a complete program. Trying to create a complete program would fail because you haven't defined a function called `main`.

```
$ gcc -c bill.c fred.c
$ ls *.o
bill.o  fred.o
```

3. Now write a program that calls the function `bill`. First, it's a good idea to create a header file for your library. This will declare the functions in your library and should be included by all applications that want to use your library. It's a good idea to include the header file in the files `fred.c` and `bill.c` too. This will help the compiler pick up any errors.

```
/*
    This is lib.h. It declares the functions fred and bill for users
```

```
*/

void bill(char *);
void fred(int);
```

4. The calling program (`program.c`) can be very simple. It includes the library header file and calls one of the functions from the library.

```
#include <stdlib.h>
#include "lib.h"

int main()
{
    bill("Hello World");
    exit(0);
}
```

5. You can now compile the program and test it. For now, specify the object files explicitly to the compiler, asking it to compile your file and link it with the previously compiled object module `bill.o`.

```
$ gcc -c program.c
$ gcc -o program program.o bill.o
$ ./program
bill: we passed Hello World
$
```

6. Now you'll create and use a library. Use the `ar` program to create the archive and add your object files to it. The program is called `ar` because it creates archives, or collections, of individual files placed together in one large file. Note that you can also use `ar` to create archives of files of any type. (Like many UNIX utilities, `ar` is a generic tool.)

```
$ ar crv libfoo.a bill.o fred.o
a - bill.o
a - fred.o
```

7. The library is created and the two object files added. To use the library successfully, some systems, notably those derived from Berkeley UNIX, require that a table of contents be created for the library. Do this with the `ranlib` command. In Linux, this step isn't necessary (but it is harmless) when you're using the GNU software development tools.

```
$ ranlib libfoo.a
```

Your library is now ready to use. You can add to the list of files to be used by the compiler to create your program like this:

```
$ gcc -o program program.o libfoo.a
$ ./program
bill: we passed Hello World
$
```

You could also use the `-l` option to access the library, but because it is not in any of the standard places, you have to tell the compiler where to find it by using the `-L` option like this:

```
$ gcc -o program program.o -L. -lfoo
```

The `-L.` option tells the compiler to look in the current directory (`.`) for libraries. The `-lfoo` option tells the compiler to use a library called `libfoo.a` (or a shared library, `libfoo.so`, if one is present). To see which functions are included in an object file, library, or executable program, you can use the `nm` command. If you take a look at `program` and `lib.a`, you see that the library contains both `fred` and `bill`, but that `program` contains only `bill`. When the program is created, it includes only functions from the library that it actually needs. Including the header file, which contains declarations for all of the functions in the library, doesn't cause the entire library to be included in the final program.

If you're familiar with Windows software development, there are a number of direct analogies here, illustrated in the following table.

Item	UNIX	Windows
object module	`func.o`	`FUNC.OBJ`
static library	`lib.a`	`LIB.LIB`
program	`program`	`PROGRAM.EXE`

Shared Libraries

One disadvantage of static libraries is that when you run many applications at the same time and they all use functions from the same library, you may end up with many copies of the same functions in memory and indeed many copies in the program files themselves. This can consume a large amount of valuable memory and disk space.

Many UNIX systems and Linux-support shared libraries can overcome this disadvantage. A complete discussion of shared libraries and their implementation on different systems is beyond the scope of this book, so we'll restrict ourselves to the visible implementation under Linux.

Shared libraries are stored in the same places as static libraries, but shared libraries have a different filename suffix. On a typical Linux system, the shared version of the standard math library is `/lib/libm.so`.

When a program uses a shared library, it is linked in such a way that it doesn't contain function code itself, but references to shared code that will be made available at run time. When the resulting program is loaded into memory to be executed, the function references are resolved and calls are made to the shared library, which will be loaded into memory if needed.

In this way, the system can arrange for a single copy of a shared library to be used by many applications at once and stored just once on the disk. An additional benefit is that the shared library can be updated independently of the applications that rely on it. Symbolic links from the `/lib/libm.so` file to the actual library revision (`/lib/libm.so.N` where `N` represents a major version number — 6 at the time of writing) are used. When Linux starts an application, it can take into account the version of a library required by the application to prevent major new versions of a library from breaking older applications.

> **The following example outputs are taken from a SUSE 10.3 distribution. Your output may differ slightly if you are not using this distribution.**

For Linux systems, the program (the dynamic loader) that takes care of loading shared libraries and resolving client program function references is called `ld.so` and may be made available as `ld-linux.so.2` or `ld-lsb.so.2` or `ld-lsb.so.3`. The additional locations searched for shared libraries are configured in the file `/etc/ld.so.conf`, which needs to be processed by `ldconfig` if changed (for example, if X11 shared libraries are added when the X Window System is installed).

You can see which shared libraries are required by a program by running the utility `ldd`. For example, if you try running it on your example application, you get the following:

```
$ ldd program
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7db4000)
        /lib/ld-linux.so.2 (0xb7efc000)
```

In this case, you see that the standard C library (`libc`) is shared (`.so`). The program requires major Version 6. Other UNIX systems will make similar arrangements for access to shared libraries. Refer to your system documentation for details.

In many ways, shared libraries are similar to dynamic-link libraries used under Windows. The `.so` libraries correspond to `.DLL` files and are required at run time, and the `.a` libraries are similar to `.LIB` files included in the program executable.

Getting Help

The vast majority of Linux systems are reasonably well documented with respect to the system programming interfaces and standard utilities. This is true because, since the earliest UNIX systems, programmers have been encouraged to supply a manual page with their applications. These manual pages, which are sometimes provided in a printed form, are invariably available electronically.

The `man` command provides access to the online manual pages. The pages vary considerably in quality and detail. Some may simply refer the reader to other, more thorough documentation, whereas others give a complete list of all options and commands that a utility supports. In either case, the manual page is a good place to start.

The GNU software suite and some other free software use an online documentation system called `info`. You can browse full documentation online using a special program, `info`, or via the `info` command of

the emacs editor. The benefit of the info system is that you can navigate the documentation using links and cross-references to jump directly to relevant sections. For the documentation author, the info system has the benefit that its files can be automatically generated from the same source as the printed, typeset documentation.

Try It Out — Manual Pages and info

Let's look for documentation of the GNU C compiler (gcc).

1. First take a look at the manual page.

```
$ man gcc

GCC(1)                        GNU                        GCC(1)



NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below
       for the remainder.  g++ accepts mostly the same options as
       gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, com
       pilation, assembly and linking.  The ``overall options''
       allow you to stop this process at an intermediate stage.
       For example, the -c option says not to run the linker.
       Then the output consists of object files output by the
       assembler.

       Other options are passed on to one stage of processing.
       Some options control the preprocessor and others the com
       piler itself. Yet other options control the assembler and
       linker; most of these are not documented here, since we
       rarely need to use any of them.
...
```

If you want, you can read about the options that the compiler supports. The manual page in this case is quite long, but it forms only a small part of the total documentation for GNU C (and C++).

When reading manual pages, you can use the spacebar to read the next page, Enter (or Return if your keyboard has that key instead) to read the next line, and *q* to quit altogether.

2. To get more information on GNU C, you can try `info`.

```
$ info gcc

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 4.1.3.  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++ Applications.
* 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, 39 lines --Top-------------------------------
Welcome to Info version 4.8. Type ? for help, m for menu item.
```

You're presented with a long menu of options that you can select to move around a complete text version of the documentation. Menu items and a hierarchy of pages allow you to navigate a very large document. On paper, the GNU C documentation runs to many hundreds of pages.

The `info` system also contains its own help page in `info` form pages, of course. If you type Ctrl+H, you'll be presented with some help that includes a tutorial on using `info`. The `info` program is available with many Linux distributions and can be installed on other UNIX systems.

Summary

In this introductory chapter, we've looked at Linux programming and the things Linux holds in common with proprietary UNIX systems. We've noted the wide variety of programming systems available to UNIX developers. We've also presented a simple program and library to demonstrate the basic C tools, comparing them with their Windows equivalents.

2

Shell Programming

Having started this book on programming Linux using C, we now take a detour into writing shell programs. Why? Well, Linux isn't like systems where the command-line interface is an afterthought to the graphical interface. UNIX, Linux's inspiration, originally had no graphical interface at all; everything was done from the command line. Consequently, the command-line system of UNIX underwent a lot of development and became a very powerful feature. This has been carried into Linux, and some of the most powerful things that you can do are most easily done from the shell. Because the shell is so important to Linux, and is so useful for automating simple tasks, shell programming is covered early.

Throughout this chapter, we'll be showing you the syntax, structures, and commands available to you when you're programming the shell, usually making use of interactive (screen-based) examples. These should serve as a useful synopsis of most of the shell's features and their effects. We will also sneak a look at a couple of particularly useful command-line utilities often called from the shell: `grep` and `find`. While looking at `grep`, we also cover the fundamentals of regular expressions, which crop up in Linux utilities and in programming languages such as Perl, Ruby, and PHP. At the end of the chapter, you'll learn how to program a real-life script, which is reprogrammed and extended in C throughout the book. This chapter covers the following:

- What a shell is
- Basic considerations
- The subtleties of syntax: variables, conditions, and program control
- Lists
- Functions
- Commands and command execution
- Here documents
- Debugging
- `grep` and regular expressions
- `find`

Whether you're faced with a complex shell script in your system administration, or you want to prototype your latest big (but beautifully simple) idea, or just want to speed up some repetitive task, this chapter is for you.

Why Program with a Shell?

One reason to use the shell for programming is that you can program the shell quickly and simply. Moreover, a shell is always available even on the most basic Linux installation, so for simple prototyping you can find out if your idea works. The shell is also ideal for any small utilities that perform some relatively simple task for which efficiency is less important than easy configuration, maintenance, and portability. You can use the shell to organize process control, so that commands run in a predetermined sequence dependent on the successful completion of each stage.

Although the shell has superficial similarities to the Windows command prompt, it's much more powerful, capable of running reasonably complex programs in its own right. Not only can you execute commands and call Linux utilities, you can also write them. The shell executes shell programs, often referred to as *scripts*, which are interpreted at runtime. This generally makes debugging easier because you can easily execute single lines, and there's no recompile time. However, this can make the shell unsuitable for time-critical or processor-intensive tasks.

A Bit of Philosophy

Here we come to a bit of UNIX — and of course Linux — philosophy. UNIX is built on and depends on a high level of code reuse. You build a small and simple utility and people use it as one link in a string of others to form a command. One of the pleasures of Linux is the variety of excellent tools available. A simple example is this command:

```
$ ls -al | more
```

This command uses the `ls` and `more` utilities and pipes the output of the file listing to a screen-at-a-time display. Each utility is one more building block. You can often use many small scripts together to create large and complex suites of programs.

For example, if you want to print a reference copy of the `bash manual` pages, then use

```
$ man bash | col -b | lpr
```

Furthermore, because of Linux's automatic file type handling, the users of these utilities usually don't need to know what language the utilities are written in. If the utility needs to run faster, it's quite common to prototype utilities in the shell and reimplement them later in C or C++, Perl, Python, or some other language that executes more swiftly once an idea has proven its worth. Conversely, if the utility works adequately in the shell, you can leave well enough alone.

Whether or not you ever reimplement the script depends on whether it needs optimizing, whether it needs to be portable, whether it should be easy to change, and whether (as usually happens) it outgrows its original purpose.

Numerous examples of shell scripts are already loaded on your Linux system in case you're curious, including package installers, `.xinitrc` *and* `startx`*, and the scripts in* `/etc/rc.d` *to configure the system on boot-up.*

What Is a Shell?

Before jumping in and discussing how to program using a shell, let's review the shell's function and the different shells available for Linux. A *shell* is a program that acts as the interface between you and the Linux system, enabling you to enter commands for the operating system to execute. In that respect, it resembles the Windows command prompt, but as mentioned earlier, Linux shells are much more powerful. For example, input and output can be redirected using < and >, data piped between simultaneously executing programs using `|`, and output from a subprocess grabbed by using `$(...)`. On Linux it's quite feasible to have multiple shells installed, with different users able to pick the one they prefer. Figure 2-1 shows how the shell (two shells actually, both bash and csh) and other programs sit around the Linux kernel.

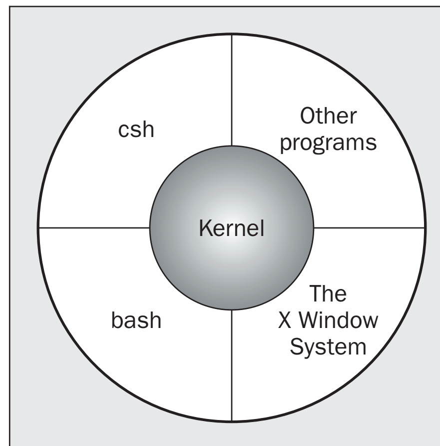


Figure 2-1

Because Linux is so modular, you can slot in one of the many different shells in use, although most of them are derived from the original Bourne shell. On Linux, the standard shell that is always installed as `/bin/sh` is called *bash* (the GNU Bourne-Again SHell), from the GNU suite of tools. Because this is an excellent shell that is always installed on Linux systems, is open source, and is portable to almost all UNIX variants, bash is the shell we will be using. This chapter uses bash version 3 and mostly uses the features common to all POSIX-compatible shells. We assume that the shell has been installed as `/bin/sh` and that it is the default shell for your login. On most Linux distributions, the program `/bin/sh`, the default shell, is actually a link to the program `/bin/bash`.

You can check the version of bash you have with the following command:

```
$ /bin/bash --version
GNU bash, version 3.2.9(1)-release (i686-pc-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
```

> **To change to a different shell — if bash isn't the default on your system, for example — just execute the desired shell's program (e.g., `/bin/bash`) to run the new shell and change the command prompt. If you are using UNIX, and bash isn't installed, you can download it free from the GNU Web site at `www.gnu.org`. The sources are highly portable, and chances are good that it will compile on your version of UNIX straight out of the box.**

When you create Linux users, you can set the shell that they will use, either when the user is created or afterwards by modifying their details. Figure 2-2 shows the selection of the shell for a user using Fedora.

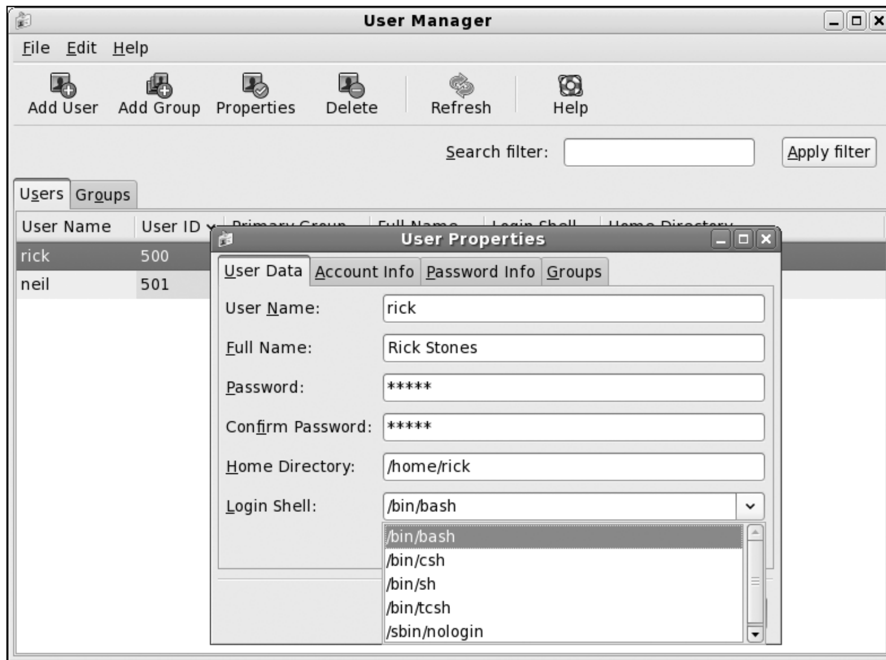


Figure 2-2

Many other shells are available, either free or commercially. The following table offers a brief summary of some of the more common shells available:

Shell Name	A Bit of History
`sh` (Bourne)	The original shell from early versions of UNIX
`csh, tcsh, zsh`	The C shell, and its derivatives, originally created by Bill Joy of Berkeley UNIX fame. The C shell is probably the third most popular type of shell after bash and the Korn shell.
`ksh, pdksh`	The Korn shell and its public domain cousin. Written by David Korn, this is the default shell on many commercial UNIX versions.
`bash`	The Linux staple shell from the GNU project. bash, or Bourne Again SHell, has the advantage that the source code is freely available, and even if it's not currently running on your UNIX system, it has probably been ported to it. bash has many similarities to the Korn shell.

Except for the C shell and a small number of derivatives, all of these are very similar and are closely aligned with the shell specified in the X/Open 4.2 and POSIX 1003.2 specifications. POSIX 1003.2 provides the minimum specification for a shell, but the extended specification in X/Open provides a more friendly and powerful shell. X/Open is usually the more demanding specification, but it also yields a friendlier system.

Pipes and Redirection

Before we get down to the details of shell programs, we need to say a little about how inputs and outputs of Linux programs (not just shell programs) can be redirected.

Redirecting Output

You may already be familiar with some redirection, such as

```
$ ls -l > lsoutput.txt
```

which saves the output of the `ls` command into a file called `lsoutput.txt`.

However, there is much more to redirection than this simple example reveals. You'll learn more about the standard file descriptors in Chapter 3, but for now all you need to know is that file descriptor 0 is the standard input to a program, file descriptor 1 is the standard output, and file descriptor 2 is the standard error output. You can redirect each of these independently. In fact, you can also redirect other file descriptors, but it's unusual to want to redirect any other than the standard ones: 0, 1, and 2.

The preceding example redirects the standard output into a file by using the `>` operator. By default, if the file already exists, then it will be overwritten. If you want to change the default behavior, you can use the command `set -o noclobber` (or `set -C`), which sets the `noclobber` option to prevent a file from being overwritten using redirection. You can cancel this option using `set +o noclobber`. You'll see more options for the `set` command later in the chapter.

To append to the file, use the `>>` operator. For example,

```
$ ps >> lsoutput.txt
```

will append the output of the `ps` command to the end of the specified file.

To redirect the standard error output, preface the `>` operator with the number of the file descriptor you wish to redirect. Because the standard error is on file descriptor 2, use the `2>` operator. This is often useful to discard error information and prevent it from appearing on the screen.

Suppose you want to use the `kill` command to kill a process from a script. There is always a slight risk that the process will die before the `kill` command is executed. If this happens, `kill` will write an error message to the standard error output, which, by default, will appear on the screen. By redirecting both the standard output and the error, you can prevent the `kill` command from writing any text to the screen.

The command

```
$ kill -HUP 1234 >killout.txt 2>killerr.txt
```

will put the output and error information into separate files.

If you prefer to capture both sets of output into a single file, you can use the `>&` operator to combine the two outputs. Therefore,

```
$ kill -1 1234 >killouterr.txt 2>&1
```

will put both the output and error outputs into the same file. Notice the order of the operators. This reads as "redirect standard output to the file `killouterr.txt`, and then direct standard error to the same place as the standard output." If you get the order wrong, the redirect won't work as you expect.

Because you can discover the result of the `kill` command using the return code (discussed in more detail later in this chapter), you don't often want to save either standard output or standard error. You can use the Linux universal "bit bucket" of `/dev/null` to efficiently discard the entire output, like this:

```
$ kill -1 1234 >/dev/null 2>&1
```

Redirecting Input

Rather like redirecting output, you can also redirect input. For example,

```
$ more < killout.txt
```

Obviously, this is a rather trivial example under Linux; the Linux `more` command is quite happy to accept filenames as parameters, unlike the Windows command-line equivalent.

Pipes

You can connect processes using the pipe operator (|). In Linux, unlike in MS-DOS, processes connected by pipes can run simultaneously and are automatically rescheduled as data flows between them. As a simple example, you could use the `sort` command to sort the output from `ps`.

If you don't use pipes, you must use several steps, like this:

```
$ ps > psout.txt
$ sort psout.txt > pssort.out
```

A much more elegant solution is to connect the processes with a pipe:

```
$ ps | sort > pssort.out
```

Because you probably want to see the output paginated on the screen, you could connect a third process, `more`, all on the same command line:

```
$ ps | sort | more
```

There's practically no limit to the permissible number of connected processes. Suppose you want to see all the different process names that are running excluding shells. You could use

```
$ ps -xo comm | sort | uniq | grep -v sh | more
```

This takes the output of `ps`, sorts it into alphabetical order, extracts processes using `uniq`, uses `grep -v sh` to remove the process named `sh`, and finally displays it paginated on the screen.

As you can see, this is a much more elegant solution than a string of separate commands, each with its own temporary file. However, be wary of one thing here: If you have a string of commands, the output file is created or written to immediately when the set of commands is created, so never use the same filename twice in a string of commands. If you try to do something like

```
cat mydata.txt | sort | uniq > mydata.txt
```

you will end up with an empty file, because you will overwrite the `mydata.txt` file before you read it.

The Shell as a Programming Language

Now that you've seen some basic shell operations, it's time to move on to some actual shell programs. There are two ways of writing shell programs. You can type a sequence of commands and allow the shell to execute them interactively, or you can store those commands in a file that you can then invoke as a program.

Interactive Programs

Just typing the shell script on the command line is a quick and easy way of trying out small code fragments, and is very useful while you are learning or just testing things out.

Suppose you have a large number of C files and wish to examine the files that contain the string `POSIX`. Rather than search using the `grep` command for the string in the files and then list the files individually, you could perform the whole operation in an interactive script like this:

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

Note how the normal `$` shell prompt changes to a `>` when the shell is expecting further input. You can type away, letting the shell decide when you're finished, and the script will execute immediately.

In this example, the `grep` command prints the files it finds containing `POSIX` and then `more` displays the contents of the file to the screen. Finally, the shell prompt returns. Note also that you called the shell variable that deals with each of the files to self-document the script. You could equally well have used `i`, but `file` is more meaningful for humans to read.

The shell also performs wildcard expansion (often referred to as *globbing*). You are almost certainly aware of the use of `'*'` as a wildcard to match a string of characters. What you may not know is that you can request single-character wildcards using `?`, while `[set]` allows any of a number of single characters to be checked. `[^set]` negates the set — that is, it includes anything but the set you've specified. Brace expansion using `{}` (available on some shells, including `bash`) allows you to group arbitrary strings together in a set that the shell will expand. For example,

```
$ ls my_{finger,toe}s
```

will list the files `my_fingers` and `my_toes`. This command uses the shell to check every file in the current directory. We will come back to these rules for matching patterns near the end of the chapter when we look in more detail at `grep` and the power of regular expressions.

Experienced Linux users would probably perform this simple operation in a much more efficient way, perhaps with a command such as

```
$ more `grep -l POSIX *`
```

or the synonymous construction

```
$ more $(grep -l POSIX *)
```

In addition,

```
$ grep -l POSIX * | more
```

will output the name of the file whose contents contained the string `POSIX`. In this script, you see the shell making use of other commands, such as `grep` and `more`, to do the hard work. The shell simply enables you to glue several existing commands together in new and powerful ways. You will see wildcard expansion used many times in the following scripts, and we'll look at the whole area of expansion in more detail when we look at regular expressions in the section on the `grep` command.

Going through this long rigmarole every time you want to execute a sequence of commands is a bore. You need to store the commands in a file, conventionally referred to as a *shell script*, so you can execute them whenever you like.

Creating a Script

Using any text editor, you need to create a file containing the commands; create a file called `first` that looks like this:

```
#!/bin/sh

# first
# This file looks through all the files in the current
# directory for the string POSIX, and then prints the names of
# those files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    echo $file
  fi
done

exit 0
```

Comments start with a `#` and continue to the end of a line. Conventionally, though, `#` is kept in the first column. Having made such a sweeping statement, we next note that the first line, `#!/bin/sh`, is a special form

of comment; the `#!` characters tell the system that the argument that follows on the line is the program to be used to execute this file. In this case, `/bin/sh` is the default shell program.

Note the absolute path specified in the comment. It is conventional to keep this shorter than 32 characters for backward compatibility, because some older UNIX versions can only use this limited number of characters when using `#!`, although Linux generally does not have this limitation.

Since the script is essentially treated as standard input to the shell, it can contain any Linux commands referenced by your `PATH` environment variable.

The `exit` command ensures that the script returns a sensible exit code (more on this later in the chapter). This is rarely checked when programs are run interactively, but if you want to invoke this script from another script and check whether it succeeded, returning an appropriate exit code is very important. Even if you never intend to allow your script to be invoked from another, you should still exit with a reasonable code. Have faith in the usefulness of your script: Assume it may need to be reused as part of another script someday.

A zero denotes success in shell programming. Since the script as it stands can't detect any failures, it always returns success. We'll come back to the reasons for using a zero exit code for success later in the chapter, when we look at the `exit` command in more detail.

Notice that this script does not use any filename extension or suffix; Linux, and UNIX in general, rarely makes use of the filename extension to determine the type of a file. You could have used `.sh` or added a different extension, but the shell doesn't care. Most preinstalled scripts will not have any filename extension, and the best way to check if they are scripts or not is to use the `file` command — for example, `file first` or `file /bin/bash`. Use whatever convention is applicable where you work, or suits you.

Making a Script Executable

Now that you have your script file, you can run it in two ways. The simpler way is to invoke the shell with the name of the script file as a parameter:

```
$ /bin/sh first
```

This should work, but it would be much better if you could simply invoke the script by typing its name, giving it the respectability of other Linux commands. Do this by changing the file mode to make the file executable for all users using the `chmod` command:

```
$ chmod +x first
```

> **Of course, this isn't the only way to use `chmod` to make a file executable. Use `man chmod` to find out more about octal arguments and other options.**

You can then execute it using the command

```
$ first
```

You may get an error saying the command wasn't found. This is almost certainly because the shell environment variable `PATH` isn't set to look in the current directory for commands to execute. To change this, either type `PATH=$PATH:.` on the command line or edit your `.bash_profile` file to add this command to the end of the file; then log out and log back in again. Alternatively, type `./first` in the directory containing the script, to give the shell the full relative path to the file.

Specifying the path prepended with `./` does have one other advantage: It ensures that you don't accidentally execute another command on the system with the same name as your script file.

> **You shouldn't change the `PATH` variable like this for the superuser, conventionally the user name `root`. It's a security loophole, because the system administrator logged in as root can be tricked into invoking a fake version of a standard command. One of the authors admits to doing this once — just to prove a point to the system administrator about security, of course! It's only a slight risk on ordinary accounts to include the current directory in the path, so if you are particularly concerned, just get into the habit of prepending `./` to all commands that are in the local directory.**

Once you're confident that your script is executing properly, you can move it to a more appropriate location than the current directory. If the command is just for your own use, you could create a `bin` directory in your home directory and add that to your path. If you want the script to be executable by others, you could use `/usr/local/bin` or another system directory as a convenient location for adding new programs. If you don't have root permissions on your system, you could ask the system administrator to copy your file for you, although you may have to convince them of its worth first. To prevent other users from changing the script, perhaps accidentally, you should remove write access from it. The sequence of commands for the administrator to set ownership and permissions would be something like this:

```
# cp first /usr/local/bin
# chown root /usr/local/bin/first
# chgrp root /usr/local/bin/first
# chmod 755 /usr/local/bin/first
```

Notice that rather than alter a specific part of the permission flags, you use the absolute form of the `chmod` here because you know exactly what permissions you require.

If you prefer, you can use the rather longer, but perhaps more obvious, form of the `chmod` command:

```
# chmod u=rwx,go=rx /usr/local/bin/first
```

Check the manual page of `chmod` for more details.

> **In Linux you can delete a file if you have write permission on the directory that contains it. To be safe, ensure that only the superuser can write to directories containing files that you want to keep safe. This makes sense because a directory is just another file, and having write permission to a directory file allows users to add and remove names.**

Shell Syntax

Now that you've seen an example of a simple shell program, it's time to look in greater depth at the programming power of the shell. The shell is quite an easy programming language to learn, not least because it's easy to test small program fragments interactively before combining them into bigger scripts. You can use the bash shell to write quite large, structured programs. The next few sections cover the following:

- Variables: strings, numbers, environments, and parameters
- Conditions: shell Booleans
- Program control: `if`, `elif`, `for`, `while`, `until`, `case`
- Lists
- Functions
- Commands built into the shell
- Getting the result of a command
- Here documents

Variables

You don't usually declare variables in the shell before using them. Instead, you create them by simply using them (for example, when you assign an initial value to them). By default, all variables are considered and stored as strings, even when they are assigned numeric values. The shell and some utilities will convert numeric strings to their values in order to operate on them as required. Linux is a case-sensitive system, so the shell considers the variable `foo` to be different from `Foo`, and both to be different from `FOO`.

Within the shell you can access the contents of a variable by preceding its name with a `$`. Whenever you extract the contents of a variable, you must give the variable a preceding `$`. When you assign a value to a variable, just use the name of the variable, which is created dynamically if necessary. An easy way to check the contents of a variable is to `echo` it to the terminal, preceding its name with a `$`.

On the command line, you can see this in action when you set and check various values of the variable `salutation`:

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

> **Note how a string must be delimited by quote marks if it contains spaces. In addition, there can't be any spaces on either side of the equals sign.**

You can assign user input to a variable by using the `read` command. This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text. The read normally completes when the user presses Enter. When reading a variable from the terminal, you don't usually need the quote marks:

```
$ read salutation
Wie geht's?
$ echo $salutation
Wie geht's?
```

Quoting

Before moving on, you should be clear about one feature of the shell: the use of quotes.

Normally, parameters in scripts are separated by whitespace characters (e.g., a space, a tab, or a newline character). If you want a parameter to contain one or more whitespace characters, you must quote the parameter.

The behavior of variables such as `$foo` inside quotes depends on the type of quotes you use. If you enclose a `$` variable expression in double quotes, then it's replaced with its value when the line is executed. If you enclose it in single quotes, then no substitution takes place. You can also remove the special meaning of the `$` symbol by prefacing it with a `\`.

Usually, strings are enclosed in double quotes, which protects variables from being separated by white space but allows `$` expansion to take place.

Try It Out Playing with Variables

This example shows the effect of quotes on the output of a variable:

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

This behaves as follows:

```
$ ./variable
Hi there
Hi there
$myvar
$myvar
```

```
Enter some text
Hello World
$myvar now equals Hello World
```

How It Works

The variable `myvar` is created and assigned the string `Hi there`. The contents of the variable are displayed with the `echo` command, showing how prefacing the variable with a `$` character expands the contents of the variable. You see that using double quotes doesn't affect the substitution of the variable, while single quotes and the backslash do. You also use the `read` command to get a string from the user.

Environment Variables

When a shell script starts, some variables are initialized from values in the environment. These are normally in all uppercase form to distinguish them from user-defined (shell) variables in scripts, which are conventionally lowercase. The variables created depend on your personal configuration. Many are listed in the manual pages, but the principal ones are listed in the following table:

Environment Variable	Description
`$HOME`	The home directory of the current user
`$PATH`	A colon-separated list of directories to search for commands
`$PS1`	A command prompt, frequently `$`, but in bash you can use some more complex values; for example, the string `[\u@\h \W]$` is a popular default that tells you the user, machine name, and current directory, as well as providing a `$` prompt.
`$PS2`	A secondary prompt, used when prompting for additional input; usually `>`.
`$IFS`	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmpfile_$$`

> **If you want to check out how the program works in a different environment by running the `env <command>`, try looking at the `env` manual pages. Later in the chapter you'll see how to set environment variables in subshells using the `export` command.**

Parameter Variables

If your script is invoked with parameters, some additional variables are created. If no parameters are passed, the environment variable `$#` still exists but has a value of 0.

The parameter variables are listed in the following table:

Parameter Variable	Description
`$1, $2, …`	The parameters given to the script
`$*`	A list of all the parameters, in a single variable, separated by the first character in the environment variable `IFS`. If `IFS` is modified, then the way `$*` separates the command line into parameters will change.
`$@`	A subtle variation on `$*`; it doesn't use the `IFS` environment variable, so parameters are not run together even if `IFS` is empty.

It's easy to see the difference between `$@` and `$*` by trying them out:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

As you can see, within double quotes, `$@` expands the positional parameters as separate fields, regardless of the `IFS` value. In general, if you want access to the parameters, `$@` is the sensible choice.

In addition to printing the contents of variables using the `echo` command, you can also read them by using the `read` command.

Try It Out Manipulating Parameter and Environment Variables

The following script demonstrates some simple variable manipulation. Once you've typed the script and saved it as `try_var`, don't forget to make it executable with `chmod +x try_var`.

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
```

```
echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

If you run this script, you get the following output:

```
$ ./try_var foo bar baz
Hello
The program ./try_var is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

How It Works

This script creates the variable `salutation`, displays its contents, and then shows how various parameter variables and the environment variable `$HOME` already exist and have appropriate values.

We'll return to parameter substitution in more detail later in the chapter.

Conditions

Fundamental to all programming languages is the ability to test conditions and perform different actions based on those decisions. Before we talk about that, though, let's look at the conditional constructs that you can use in shell scripts and then examine the control structures that use them.

A shell script can test the exit code of any command that can be invoked from the command line, including the scripts that you have written yourself. That's why it's important to always include an `exit` command with a value at the end of any scripts that you write.

The test or [Command

In practice, most scripts make extensive use of the `[` or `test` command, the shell's Boolean check. On some systems, the `[` and `test` commands are synonymous, except that when the `[` command is used, a trailing `]` is also used for readability. Having a `[` command might seem a little odd, but within the code it does make the syntax of commands look simple, neat, and more like other programming languages.

These commands call an external program in some older UNIX shells, but they tend to be built in to more modern ones. We'll come back to this when we look at commands in a later section.

Because the `test` command is infrequently used outside shell scripts, many Linux users who have never written shell scripts try to write simple programs and call them *test*. If such a program doesn't work, it's probably conflicting with the shell's `test` command. To find out whether your system has an external command of a given name, try typing something like `which test`, to check which `test` command is being executed, or use `./test` to ensure that you execute the script in the current directory. When in doubt, just get into the habit of executing your scripts by preceding the script name with `./` when invoking them.

We'll introduce the `test` command using one of the simplest conditions: checking to see whether a file exists. The command for this is `test -f <filename>`, so within a script you can write

```
if test -f fred.c
then
...
fi
```

You can also write it like this:

```
if [ -f fred.c ]
then
...
fi
```

The `test` command's exit code (whether the condition is satisfied) determines whether the conditional code is run.

Note that you must put spaces between the `[` braces and the condition being checked. You can remember this by remembering that `[` is just the same as writing `test`, and you would always leave a space after the `test` command.

If you prefer putting `then` on the same line as `if`, you must add a semicolon to separate the test from the `then`:

```
if [ -f fred.c ]; then
...
fi
```

The condition types that you can use with the `test` command fall into three types: *string comparison, arithmetic comparison,* and *file conditionals*. The following table describes these condition types:

String Comparison	Result
`string1 = string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string`	True if the string is not null
`-z string`	True if the string is null (an empty string)
Arithmetic Comparison	**Result**
`expression1 -eq expression2`	True if the expressions are equal
`expression1 -ne expression2`	True if the expressions are not equal
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`
`expression1 -lt expression2`	True if `expression1` is less than `expression2`
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`
`! expression`	True if the expression is false, and vice versa
File Conditional	**Result**
`-d file`	True if the file is a directory
`-e file`	True if the file exists. Note that historically the `-e` option has not been portable, so `-f` is usually used.
`-f file`	True if the file is a regular file
`-g file`	True if `set-group-id` is set on `file`
`-r file`	True if the file is readable
`-s file`	True if the file has nonzero size
`-u file`	True if `set-user-id` is set on `file`
`-w file`	True if the file is writable
`-x file`	True if the file is executable

> **You may be wondering what the `set-group-id` and `set-user-id` (also known as `set-gid` and `set-uid`) bits are. The `set-uid` bit gives a program the permissions of its owner, rather than its user, while the `set-gid` bit gives a program the permissions of its group. The bits are set with `chmod`, using the `s` and `g` options. The `set-gid` and `set-uid` flags have no effect on files containing shell scripts, only on executable binary files.**

We're getting ahead of ourselves slightly, but following is an example of how you would test the state of the file `/bin/bash`, just so you can see what these look like in use:

```
#!/bin/sh

if [ -f /bin/bash ]
then
  echo "file /bin/bash exists"
fi

if [ -d /bin/bash ]
then
  echo "/bin/bash is a directory"
else
  echo "/bin/bash is NOT a directory"
fi
```

Before the test can be true, all the file conditional tests require that the file also exists. This list contains just the more commonly used options to the `test` command, so for a complete list refer to the manual entry. If you're using bash, where `test` is built in, use the `help test` command to get more details. We'll use some of these options later in the chapter.

Now that you know about conditions, you can look at the control structures that use them.

Control Structures

The shell has a set of control structures, which are very similar to other programming languages.

> **In the following sections, the statements are the series of commands to perform when, while, or until the condition is fulfilled.**

if

The `if` statement is very simple: It tests the result of a command and then conditionally executes a group of statements:

```
if condition
then
  statements
```

```
else
  statements
fi
```

A common use for `if` is to ask a question and then make a decision based on the answer:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

This would give the following output:

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

This script uses the `[` command to test the contents of the variable `timeofday`. The result is evaluated by the `if` command, which then allows different lines of code to be executed.

> **Notice that you use extra white space to indent the statements inside the `if`. This is just a convenience for the human reader; the shell ignores the additional white space.**

elif

Unfortunately, there are several problems with this very simple script. For one thing, it will take any answer except `yes` as meaning `no`. You can prevent this by using the `elif` construct, which allows you to add a second condition to be checked when the `else` portion of the `if` is executed.

Try It Out Doing Checks with an elif

You can modify the previous script so that it reports an error message if the user types in anything other than `yes` or `no`. Do this by replacing the `else` with `elif` and then adding another condition:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]
then
  echo "Good morning"
```

```
elif [ $timeofday = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

How It Works

This is quite similar to the previous example, but now the `elif` command tests the variable again if the first `if` condition is not true. If neither of the tests is successful, an error message is printed and the script exits with the value 1, which the caller can use in a calling program to check whether the script was successful.

A Problem with Variables

This fixes the most obvious defect, but a more subtle problem is lurking. Try this new script, but just press Enter (or Return on some keyboards), rather than answering the question. You'll get this error message:

```
[: =: unary operator expected
```

What went wrong? The problem is in the first `if` clause. When the variable `timeofday` was tested, it consisted of a blank string. Therefore, the `if` clause looks like

```
if [ = "yes" ]
```

which isn't a valid condition. To avoid this, you must use quotes around the variable:

```
if [ "$timeofday" = "yes" ]
```

An empty variable then gives the valid test:

```
if [ "" = "yes" ]
```

The new script is as follows:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ "$timeofday" = "yes" ]
then
  echo "Good morning"
elif [ "$timeofday" = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
```

```
  exit 1
fi

exit 0
```

This is safe should a user just press Enter in answer to the question.

> **If you want the `echo` command to delete the trailing new line, the most portable option is to use the `printf` command (see the `printf` section later in this chapter), rather than the `echo` command. Some shells use `echo -e`, but that's not supported on all systems. bash allows `echo -n` to suppress the new line, so if you are confident your script needs to work only on bash, we suggest using that syntax.**

```
echo -n "Is it morning? Please answer yes or no: "
```

Note that you need to leave an extra space before the closing quotes so that there is a gap before the user-typed response, which looks neater.

for

Use the `for` construct to loop through a range of values, which can be any set of strings. They could be simply listed in the program or, more commonly, the result of a shell expansion of filenames.

The syntax is simple:

```
for variable in values
do
  statements
done
```

Try It Out Using a for Loop with Fixed Strings

The values are normally strings, so you can write the following:

```
#!/bin/sh

for foo in bar fud 43
do
  echo $foo
done
exit 0
```

That results in the following output:

```
bar
fud
43
```

> **What would happen if you changed the first line from `for foo in bar fud 43` to `for foo in "bar fud 43"`? Remember that adding the quotes tells the shell to consider everything between them as a single string. This is one way of getting spaces to be stored in a variable.**

How It Works

This example creates the variable `foo` and assigns it a different value each time around the `for` loop. Since the shell considers all variables to contain strings by default, it's just as valid to use the string `43` as the string `fud`.

Try It Out Using a for Loop with Wildcard Expansion

As mentioned earlier, it's common to use the `for` loop with a shell expansion for filenames. This means using a wildcard for the string value and letting the shell fill out all the values at run time.

You've already seen this in the original example, `first`. The script used shell expansion, the `*` expanding to the names of all the files in the current directory. Each of these in turn is used as the variable `$file` inside the `for` loop.

Let's quickly look at another wildcard expansion. Imagine that you want to print all the script files starting with the letter "f" in the current directory, and you know that all your scripts end in `.sh`. You could do it like this:

```
#!/bin/sh

for file in $(ls f*.sh); do
  lpr $file
done
exit 0
```

How It Works

This illustrates the use of the `$(command)` syntax, which is covered in more detail later (in the section on command execution). Basically, the parameter list for the `for` command is provided by the output of the command enclosed in the `$()` sequence.

The shell expands `f*.sh` to give the names of all the files matching this pattern.

> **Remember that all expansion of variables in shell scripts is done when the script is executed, never when it's written, so syntax errors in variable declarations are found only at execution time, as shown earlier when we were quoting empty variables.**

---

while

Because all shell values are considered strings by default, the `for` loop is good for looping through a series of strings, but is not so useful when you don't know in advance how many times you want the loop to be executed.

When you need to repeat a sequence of commands, but don't know in advance how many times they should execute, you will normally use a `while` loop, which has the following syntax:

```
while condition do
  statements
done
```

For example, here is a rather poor password-checking program:

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done
exit 0
```

An example of the output from this script is as follows:

```
Enter password
password
Sorry, try again
secret
$
```

Clearly, this isn't a very secure way of asking for a password, but it does serve to illustrate the `while` statement. The statements between `do` and `done` are continuously executed until the condition is no longer true. In this case, you're checking whether the value of `trythis` is equal to `secret`. The loop will continue until `$trythis` equals `secret`. You then continue executing the script at the statement immediately following the `done`.

until

The `until` statement has the following syntax:

```
until condition
do
  statements
done
```

This is very similar to the `while` loop, but with the condition test reversed. In other words, the loop continues until the condition becomes true, not while the condition is true.

In general, if a loop should always execute at least once, use a `while` loop; if it may not need to execute at all, use an `until` loop.

As an example of an `until` loop, you can set up an alarm that is initiated when another user, whose login name you pass on the command line, logs on:

```
#!/bin/bash

until who | grep "$1" > /dev/null
do
   sleep 60
done

# now ring the bell and announce the expected user.

echo -e '\a'
echo "**** $1 has just logged in ****"

exit 0
```

If the user is already logged on, the loop doesn't need to execute at all, so using `until` is a more natural choice than `while`.

case

The `case` construct is a little more complex than those you have encountered so far. Its syntax is as follows:

```
case variable in
  pattern [ | pattern] ...) statements;;
  pattern [ | pattern] ...) statements;;
  ...
esac
```

This may look a little intimidating, but the `case` construct enables you to match the contents of a variable against patterns in quite a sophisticated way and then allows execution of different statements, depending on which pattern was matched. It is much simpler than the alternative way of checking several conditions, which would be to use multiple `if`, `elif`, and `else` statements.

Notice that each pattern line is terminated with double semicolons (`;;`). You can put multiple statements between each pattern and the next, so a double semicolon is needed to mark where one statement ends and the next pattern begins.

The capability to match multiple patterns and then execute multiple related statements makes the `case` construct a good way of dealing with user input. The best way to see how `case` works is with an example. We'll develop it over three Try It Out examples, improving the pattern matching each time.

Be careful with the `case` construct if you are using wildcards such as '*' in the pattern. The problem is that the first matching pattern will be taken, even if a later pattern matches more exactly.

Try It Out Case I: User Input

You can write a new version of the input-testing script and, using the `case` construct, make it a little more selective and forgiving of unexpected input:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes)  echo "Good Morning";;
    no )  echo "Good Afternoon";;
    y  )  echo "Good Morning";;
    n  )  echo "Good Afternoon";;
    *  )  echo "Sorry, answer not recognized";;
esac

exit 0
```

How It Works

When the `case` statement is executing, it takes the contents of `timeofday` and compares it to each string in turn. As soon as a string matches the input, the `case` command executes the code following the `)` and finishes.

The `case` command performs normal expansion on the strings that it's using for comparison. You can therefore specify part of a string followed by the * wildcard. Using a single `*` will match all possible strings, so always put one after the other matching strings to ensure that the `case` statement ends with some default action if no other strings are matched. This is possible because the `case` statement compares against each string in turn. It doesn't look for a best match, just the first match. The default condition often turns out to be the impossible condition, so using `*` can help in debugging scripts.

Try It Out Case II: Putting Patterns Together

The preceding `case` construction is clearly more elegant than the multiple `if` statement version, but by putting the patterns together, you can make a much cleaner version:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )      echo "Good Morning";;
    n* | N* )                  echo "Good Afternoon";;
    * )                        echo "Sorry, answer not recognized";;
esac

exit 0
```

How It Works

This script uses multiple strings in each entry of the `case` so that `case` tests several different strings for each possible statement. This makes the script both shorter and, with practice, easier to read. This code also shows how the `*` wildcard can be used, although this may match unintended patterns. For example, if the user enters `never`, then this will be matched by `n*` and `Good Afternoon` will be displayed, which isn't the intended behavior. Note also that the `*` wildcard expression doesn't work within quotes.

Try It Out Case III: Executing Multiple Statements

Finally, to make the script reusable, you need to have a different exit value when the default pattern is used because the input was not understood:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )
           echo "Good Morning"
           echo "Up bright and early this morning"
           ;;
    [nN]*)
           echo "Good Afternoon"
           ;;
    *)
           echo "Sorry, answer not recognized"
           echo "Please answer yes or no"
           exit 1
           ;;
esac

exit 0
```

How It Works

To show a different way of pattern matching, this code changes the way in which the `no` case is matched. You also see how multiple statements can be executed for each pattern in the `case` statement. You must be careful to put the most explicit matches first and the most general match last. This is important because `case` executes the first match it finds, not the best match. If you put the `*)` first, it would always be matched, regardless of what was input.

Note that the `;;` *before* `esac` *is optional. Unlike C programming, where leaving out a break is poor programming practice, leaving out the final* `;;` *is no problem if the last case is the default because no other cases will be considered.*

To make the `case` matching more powerful, you could use something like this:

```
[yY] | [Yy][Ee][Ss] )
```

This restricts the permitted letters while allowing a variety of answers, and offers more control than the `*` wildcard.

Lists

Sometimes you want to connect commands in a series. For instance, you may want several different conditions to be met before you execute a statement:

```
if [ -f this_file ]; then
    if [ -f that_file ]; then
        if [ -f the_other_file ]; then
            echo "All files present, and correct"
        fi
    fi
fi
```

Or you might want at least one of a series of conditions to be true:

```
if [ -f this_file ]; then
   foo="True"
elif [ -f that_file ]; then
   foo="True"
elif [ -f the_other_file ]; then
   foo="True"
else
   foo="False"
fi
if [ "$foo" = "True" ]; then
   echo "One of the files exists"
fi
```

Although these can be implemented using multiple `if` statements, you can see that the results are awkward. The shell has a special pair of constructs for dealing with lists of commands: the AND list and the OR list. These are often used together, but we'll review their syntax separately.

The AND List

The AND list construct enables you to execute a series of commands, executing the next command only if all the previous commands have succeeded. The syntax is

```
statement1 && statement2 && statement3 && ...
```

Starting at the left, each statement is executed; if it returns `true`, the next statement to the right is executed. This continues until a statement returns `false`, after which no more statements in the list are executed. The `&&` tests the condition of the preceding command.

Each statement is executed independently, enabling you to mix many different commands in a single list, as the following script shows. The AND list as a whole succeeds if all commands are executed successfully, but it fails otherwise.

Try It Out AND Lists

In the following script, you `touch file_one` (to check whether it exists and create it if it doesn't) and then remove `file_two`. Then the AND list tests for the existence of each of the files and echoes some text in between.

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

Try the script and you'll get the following result:

```
hello
in else
```

How It Works

The `touch` and `rm` commands ensure that the files in the current directory are in a known state. The `&&` list then executes the `[ -f file_one ]` statement, which succeeds because you just made sure that the file existed. Because the previous statement succeeded, the `echo` command is executed. This also succeeds (`echo` always returns `true`). The third test, `[ -f file_two ]`, is then executed. It fails because the file doesn't exist. Because the last command failed, the final `echo` statement isn't executed. The result of the `&&` list is `false` because one of the commands in the list failed, so the `if` statement executes its `else` condition.

The OR List

The OR list construct enables us to execute a series of commands until one succeeds, and then not execute any more. The syntax is as follows:

```
statement1 || statement2 || statement3 || ...
```

Starting at the left, each statement is executed. If it returns `false`, then the next statement to the right is executed. This continues until a statement returns `true`, at which point no more statements are executed.

The `||` list is very similar to the `&&` list, except that the rule for executing the next statement is that the previous statement must fail.

Try It Out OR Lists

Copy the previous example and change the shaded lines in the following listing:

```
#!/bin/sh

rm -f file_one

if [ -f file_one ] || echo "hello" || echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

This results in the following output:

```
hello
in if
```

How It Works

The first two lines simply set up the files for the rest of the script. The first command, `[ -f file_one ]`, fails because the file doesn't exist. The `echo` statement is then executed. Surprise, surprise — this returns `true`, and no more commands in the `||` list are executed. The `if` succeeds because one of the commands in the `||` list (the `echo`) was `true`.

The result of both of these constructs is the result of the last statement to be executed.

These list-type constructs execute in a similar way to those in C when multiple conditions are being tested. Only the minimum number of statements is executed to determine the result. Statements that can't affect the result are not executed. This is commonly referred to as *short circuit evaluation*.

Combining these two constructs is a logician's heaven. Try out the following:

```
[ -f file_one ] && command for true || command for false
```

This will execute the first command if the test succeeds and the second command otherwise. It's always best to experiment with these more unusual lists, and in general you should use braces to force the order of evaluation.

Statement Blocks

If you want to use multiple statements in a place where only one is allowed, such as in an AND or OR list, you can do so by enclosing them in braces `{}` to make a statement block. For example, in the application presented later in this chapter, you'll see the following code:

```
get_confirm && {
   grep -v "$cdcatnum" $tracks_file > $temp_file
```

```
    cat $temp_file > $tracks_file
    echo
    add_record_tracks
}
```

Functions

You can define functions in the shell; and if you write shell scripts of any size, you'll want to use them to structure your code.

> **As an alternative, you could break a large script into lots of smaller scripts, each of which performs a small task. This has some drawbacks: Executing a second script from within a script is much slower than executing a function. It's more difficult to pass back results, and there can be a very large number of small scripts. You should consider the smallest part of your script that sensibly stands alone and use that as your measure of when to break a large script into a collection of smaller ones.**

To define a shell function, simply write its name followed by empty parentheses and enclose the statements in braces:

```
function_name () {
  statements
}
```

Try It Out A Simple Function

Let's start with a really simple function:

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

Running the script will output the following:

```
script starting
Function foo is executing
script ending
```

How It Works

This script starts executing at the top, so nothing is different there, but when it finds the `foo() {` construct, it knows that a function called `foo` is being defined. It stores the fact that `foo` refers to a function and continues executing after the matching `}`. When the single line `foo` is executed, the shell knows to execute the previously defined function. When this function completes, execution resumes at the line after the call to `foo`.

You must always define a function before you can invoke it, a little like the Pascal style of function definition before invocation, except that there are no forward declarations in the shell. This isn't a problem, because all scripts start executing at the top, so simply putting all the functions before the first call of any function will always cause all functions to be defined before they can be invoked.

When a function is invoked, the positional parameters to the script, `$*`, `$@`, `$#`, `$1`, `$2`, and so on, are replaced by the parameters to the function. That's how you read the parameters passed to the function. When the function finishes, they are restored to their previous values.

> **Some older shells may not restore the value of positional parameters after functions execute. It's wise not to rely on this behavior if you want your scripts to be portable.**

You can make functions return numeric values using the `return` command. The usual way to make functions return strings is for the function to store the string in a variable, which can then be used after the function finishes. Alternatively, you can `echo` a string and catch the result, like this:

```
foo () { echo JAY;}

...

result="$(foo)"
```

Note that you can declare local variables within shell functions by using the `local` keyword. The variable is then only in scope within the function. Otherwise, the function can access the other shell variables that are essentially global in scope. If a local variable has the same name as a global variable, it overlays that variable, but only within the function. For example, you can make the following changes to the preceding script to see this in action:

```
#!/bin/sh

sample_text="global variable"

foo() {

    local sample_text="local variable"
    echo "Function foo is executing"
    echo $sample_text
}

echo "script starting"
echo $sample_text
```

```
foo

echo "script ended"
echo $sample_text

exit 0
```

In the absence of a `return` command specifying a return value, a function returns the exit status of the last command executed.

Try It Out Returning a Value

The next script, `my_name`, shows how parameters to a function are passed and how functions can return a `true` or `false` result. You call this script with a parameter of the name you want to use in the question.

1. After the shell header, define the function `yes_or_no`:

```
#!/bin/sh

yes_or_no() {
  echo "Is your name $* ?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no"
    esac
  done
}
```

2. Then the main part of the program begins:

```
echo "Original parameters are $*"

if yes_or_no "$1"
then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi
exit 0
```

Typical output from this script might be as follows:

```
$ ./my_name Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
```

```
Enter yes or no: yes
Hi Rick, nice name
$
```

How It Works

As the script executes, the function `yes_or_no` is defined but not yet executed. In the `if` statement, the script executes the function `yes_or_no`, passing the rest of the line as parameters to the function after substituting the `$1` with the first parameter to the original script, `Rick`. The function uses these parameters, which are now stored in the positional parameters `$1`, `$2`, and so on, and returns a value to the caller. Depending on the return value, the `if` construct executes the appropriate statement.

As you've seen, the shell has a rich set of control structures and conditional statements. You need to learn some of the commands that are built into the shell; then you'll be ready to tackle a real programming problem with no compiler in sight!

Commands

You can execute two types of commands from inside a shell script. There are "normal" commands that you could also execute from the command prompt (called *external commands*), and there are "built-in" commands (called *internal commands*), as mentioned earlier. Built-in commands are implemented internally to the shell and can't be invoked as external programs. However, most internal commands are also provided as standalone programs — this requirement is part of the POSIX specification. It generally doesn't matter if the command is internal or external, except that internal commands execute more efficiently.

Here we'll cover only the main commands, both internal and external, that we use when we're programming scripts. As a Linux user, you probably know many other commands that are valid at the command prompt. Always remember that you can use any of these in a script in addition to the built-in commands presented here.

break

Use `break` for escaping from an enclosing `for`, `while`, or `until` loop before the controlling condition has been met. You can give `break` an additional numeric parameter, which is the number of loops to break out of, but this can make scripts very hard to read, so we don't suggest you use it. By default, `break` escapes a single level.

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
    if [ -d "$file" ]; then
        break;
```

```
    fi
done

echo first directory starting fred was $file

rm -rf fred*
exit 0
```

The : Command

The colon command is a null command. It's occasionally useful to simplify the logic of conditions, being an alias for `true`. Since it's built-in, `:` runs faster than `true`, though its output is also much less readable.

You may see it used as a condition for `while` loops; `while :` implements an infinite loop in place of the more common `while true`.

The `:` construct is also useful in the conditional setting of variables. For example,

```
: ${var:=value}
```

Without the `:`, the shell would try to evaluate `$var` as a command.

> **In some, mostly older, shell scripts, you may see the colon used at the start of a line to introduce a comment, but modern scripts should always use # to start a comment line because this executes more efficiently.**

```
#!/bin/sh

rm -f fred
if [ -f fred ]; then
    :
else
    echo file fred did not exist
fi

exit 0
```

continue

Rather like the C statement of the same name, this command makes the enclosing `for`, `while`, or `until` loop continue at the next iteration, with the loop variable taking the next value in the list:

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
```