

SECOND EDITION

bash

Bourne

zsh

Korn

SAMS
Teach Yourself

Shell Programming

in **24**
Hours

Sriranga Veeraraghavan

Sriranga Veeraraghavan



SAMS
Teach Yourself

Shell Programming

in **24** Hours

SECOND EDITION

SAMS

800 East 96th St., Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Shell Programming in 24 Hours, Second Edition

Copyright © 2002 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32358-3

Library of Congress Catalog Card Number: 2001096631

Printed in the United States of America

First Printing: April 2002

06 05 13 12 11 10 9 8 7

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

ACQUISITIONS EDITOR

Katie Purdum

DEVELOPMENT EDITOR

Steve Rowe

TECHNICAL EDITOR

Michael Watson

MANAGING EDITOR

Charlotte Clapp

PROJECT EDITOR

Natalie Harris

COPY EDITORS

Kezia Endsley
Rhonda Tinch-Mize

INDEXER

Kelly Castell

PROOFREADERS

Linda Seifert
Karen Whitehouse

INTERIOR DESIGN

Gary Adair

COVER DESIGN

Aren Howell

PAGE LAYOUT

Stacey Richwine-DeRome

Contents at a Glance

Introduction	1
PART I Introduction to UNIX and Shell Tools	7
Hour 1 Shell Basics	9
2 Script Basics	21
3 Working with Files	37
4 Working with Directories	53
5 Input and Output	71
6 Manipulating File Attributes	89
7 Processes	105
PART II Shell Programming	119
Hour 8 Variables	121
9 Substitution	135
10 Quoting	147
11 Flow Control	159
12 Loops	181
13 Parameters	197
14 Functions	213
15 Text Filters	231
16 Filtering Text with Regular Expressions	249
17 Filtering Text with awk	267
18 Other Tools	293
PART III Advanced Topics	311
Hour 19 Signals	313
20 Debugging	325
21 Problem Solving with Functions	341
22 Problem Solving with Shell Scripts	359
23 Scripting for Portability	389
24 Shell Programming FAQs	403

PART IV	Appendixes	417
Appendix A	Command Quick Reference	419
B	Glossary	433
C	Answers to Questions	441
D	Shell Function Library	461
	Index	465

Contents

Introduction	1
PART I Introduction to UNIX and Shell Tools	7
Hour 1 Shell Basics	9
What Is a Command?	10
Simple Commands.....	11
Complex Commands	11
Compound Commands	12
What Is the Shell?.....	13
The Shell Prompt.....	14
Different Types of Shells.....	14
Summary	18
Questions.....	19
Terms.....	19
Hour 2 Script Basics	21
The UNIX System	22
Logging In	23
Shell Modes and Initialization	24
Initialization Procedures	24
Initialization File Contents	26
Interactive and Non-Interactive Shells	28
Getting Help	31
man	31
Online Resources.....	34
Summary	35
Questions.....	35
Terms.....	35
Hour 3 Working with Files	37
Listing Files	38
Hidden Files.....	39
Option Grouping	40
File Contents	41
cat	41
wc	43
Manipulating Files	46
Copying Files (cp).....	46
Renaming Files (mv)	48
Removing Files (rm)	49

Summary	50
Questions.....	51
Terms.....	51
Hour 4 Working with Directories	53
The Directory Tree	54
Filenames.....	54
Pathnames	55
Switching Directories	57
Home Directories.....	57
Changing Directories.....	58
Listing Files and Directories.....	60
Listing Directories.....	60
Listing Files.....	61
Manipulating Directories	62
Creating Directories.....	62
Copying Files and Directories.....	63
Moving Files and Directories	64
Removing Directories	66
Summary	68
Questions.....	68
Terms.....	69
Hour 5 Input and Output	71
Output	71
Output to the Terminal	72
Output Redirection	77
Input	79
Input Redirection.....	79
Reading User Input	81
Pipelines.....	81
File Descriptors.....	82
Associating Files with a File Descriptor	82
General Input/Output Redirection	83
Summary	87
Questions.....	87
Terms.....	87
Hour 6 Manipulating File Attributes	89
File Types	89
Determining a File's Type	90
Regular Files	90
Links	91
Device Files.....	94
Named Pipes	95

Owners, Groups, and Permissions	95
Viewing Permissions	96
Changing File and Directory Permissions.....	98
Changing Owners and Groups	101
Summary	103
Questions.....	103
Terms.....	104
Hour 7 Processes	105
Starting a Process	105
Foreground Processes	106
Background Processes	106
Listing and Terminating Processes	111
jobs	112
ps Command	112
Killing a Process (kill Command).....	114
Parent and Child Processes.....	114
Subshells	115
Process Permissions.....	116
Overlaying the Current Process (exec Command)	116
Summary	117
Questions.....	117
Terms.....	117
PART II Shell Programming	119
Hour 8 Variables	121
Working with Variables.....	121
Scalar Variables	122
Array Variables	124
Read-Only Variables	128
Unsetting Variables	129
Environment and Shell Variables	129
Exporting Environment Variables	130
Shell Variables	131
Summary	132
Questions.....	132
Terms.....	133
Hour 9 Substitution	135
Filename Substitution (Globbing)	136
The * Meta-Character	136
The ? Meta-Character	138
Matching Sets of Characters	139

Variable Substitution.....	141
Default Value Substitution.....	141
Default Value Assignment	142
Null Value Error.....	142
Substitute When Set	143
Command and Arithmetic Substitution	143
Command Substitution	143
Arithmetic Substitution	144
Summary	146
Questions.....	146
Terms.....	146
Hour 10 Quoting	147
Quoting with Backslashes.....	148
Meta-Characters and Escape Sequences	149
Using Single Quotes	149
Using Double Quotes	150
Quoting Rules and Situations	151
Quoting Ignores Word Boundaries	152
Combining Quoting in Commands	152
Embedding Spaces in a Single Argument.....	152
Quoting Newlines to Continue on the Next Line	153
Quoting to Access Filenames Containing Special Characters	154
Quoting Regular Expression Wildcards	155
Quoting the Backslash to Enable echo Escape Sequences	155
Quoting Wildcards for <code>cpio</code> and <code>find</code>	156
Summary	157
Questions.....	158
Terms.....	158
Hour 11 Flow Control	159
The <code>if</code> Statement	160
An <code>if</code> Statement Example.....	160
Using <code>test</code>	163
The <code>case</code> Statement.....	175
A <code>case</code> Statement Example	175
Using Patterns	177
Summary	178
Questions.....	178
Terms.....	179
Hour 12 Loops	181
The <code>while</code> Loop	181
Nesting <code>while</code> Loops	183
Validating User Input with <code>while</code>	184

Input Redirection and <code>while</code>	185
The <code>until</code> Loop	187
The <code>for</code> and <code>select</code> Loops	188
The <code>for</code> Loop	188
The <code>select</code> Loop	190
Loop Control	192
Infinite Loops and the <code>break</code> Command	192
The <code>continue</code> Command	194
Summary	195
Questions	195
Terms	196
Hour 13 Parameters	197
Special Variables	198
Using <code>\$0</code>	198
Options and Arguments	200
Dealing with Arguments	201
Using <code>basename</code>	201
Common Argument Handling Problems	203
Option Parsing in Shell Scripts	205
Using <code>getopts</code>	206
Summary	210
Questions	210
Terms	211
Hour 14 Functions	213
Using Functions	213
Executing Functions	214
Aliases Versus Functions	217
Unsetting Functions	218
Understanding Scope, Recursion, Return Codes, and Data Sharing	218
Scope	218
Recursion	221
Return Codes	223
Data Sharing	223
Moving Around the File System	223
Summary	228
Questions	228
Terms	229
Hour 15 Text Filters	231
The <code>head</code> and <code>tail</code> Commands	231
The <code>head</code> Command	232
The <code>tail</code> Command	233

Using grep	234
Looking for Words.....	235
Reading From STDIN	236
Line Numbers	237
Listing Filenames Only	238
Counting Words	238
The tr Command	239
The sort Command.....	241
The uniq Command.....	241
Sorting Numbers	242
Using Character Classes with tr.....	244
Summary	245
Questions.....	246
Terms.....	247
Hour 16 Filtering Text with Regular Expressions	249
The Basics of awk and sed	250
Invocation Syntax	250
Basic Operation	250
Regular Expressions	251
Using sed	257
Printing Lines	258
Deleting Lines	259
Performing Substitutions	260
Using Multiple sed Commands.....	262
Using sed in a Pipeline	263
Summary	264
Questions.....	264
Terms.....	265
Hour 17 Filtering Text with awk	267
What Is awk?	267
Basic Syntax	268
Field Editing	269
Taking Pattern-Specific Actions	270
Comparison Operators.....	271
Using STDIN as Input.....	274
Using awk Features	275
Variables	276
Flow Control	283
Summary	288
Questions.....	289
Terms.....	291

Hour 18 Other Tools	293
The Built-In Commands	293
The eval Command.....	294
The : Command	294
The type Command.....	296
The sleep Command	297
The find Command	298
find: Starting Directory	299
find: -name Option	300
find: -type Option	300
find: -mtime, -atime, -ctime	301
find: -size Option	302
find: Combining Options.....	302
find: Negating Options	303
find: -print Action.....	303
find: -exec Action.....	303
xargs	304
The expr Command	306
expr and Regular Expressions.....	307
The bc Command	307
Summary	308
Questions.....	309
Terms.....	309
PART III Advanced Topics	311
Hour 19 Signals	313
How Are Signals Represented?	314
Getting a List of Signals	314
Default Actions	315
Delivering Signals	315
Dealing with Signals.....	316
The trap Command.....	317
Cleaning Up Temporary Files	317
Ignoring Signals.....	319
Setting Up a Timer	320
Summary	324
Questions.....	324
Terms.....	324
Hour 20 Debugging	325
Enabling Debugging	326
Using the set command	327

Using Syntax Checking	328
Why Syntax Checking Is Important	329
Using Verbose Mode	331
Shell Tracing	332
Finding Syntax Bugs Using Shell Tracing	333
Finding Logical Bugs Using Shell Tracing	335
Using Debugging Hooks	337
Summary	339
Questions	339
Terms	340
Hour 21 Problem Solving with Functions	341
Library Basics	341
What Is a Library?	342
Using a Library	342
Creating a Library	343
Naming the Library	343
Naming the Functions	344
Displaying Error and Warning Messages	344
Asking Questions	345
Checking Disk Space	351
Obtaining a Process ID by its Process Name	354
Getting a User's Numeric User ID	355
Summary	356
Questions	356
Terms	357
Hour 22 Problem Solving with Shell Scripts	359
Startup Scripts	360
System Startup	360
Developing an Init Script	364
Maintaining an Address Book	373
Showing People	375
Adding a Person	377
Deleting a Person	380
Summary	385
Questions	385
Terms	387
Hour 23 Scripting for Portability	389
Determining UNIX Versions	390
BSD	390
System V	390
Linux	391
Using uname to Determine the UNIX Version	392
Determining the UNIX Version Using a Function	394

Techniques for Increasing Portability	396
Conditional Execution	396
Abstraction	397
Summary	400
Question	401
Terms	401
Hour 24 Shell Programming FAQs	403
Shell and Command Questions	404
Variable and Argument Questions	409
File and Directory Questions	412
Summary	416
PART IV Appendixes	417
APPENDIX A Command Quick Reference	419
Reserved Words and Built-in	
Shell Commands	420
Conditional Expressions	423
File Tests	423
String Tests	424
Integer Comparisons	424
Compound Expressions	424
Arithmetic Expressions (ksh, bash, and zsh Only)	424
Integer Expression Operators	425
Parameters and Variables	426
User-Defined Variables	426
Special Variables	427
Shell Variables	428
Input/Output	428
Input and Output Redirection	429
Here Document	429
Pattern Matching and Regular Expressions	430
Filename Expansion and Pattern Matching	430
Limited Regular Expression Wildcards	430
Extended Regular Expression Wildcards	430
APPENDIX B Glossary	433
APPENDIX C Answers to Questions	441
APPENDIX D Shell Function Library	461
Index	465

About the Author

SRIRANGA VEERARAGHAVAN is a material scientist by training and a software engineer by trade. He has several years of software development experience in C, Java, Perl, and Bourne Shell and has contributed to several books, including *Solaris 8: Complete Reference*, *UNIX Unleashed* and *Special Edition Using UNIX*. Sriranga graduated from the University of California at Berkeley in 1997 and is presently pursuing further studies. He is currently employed in the Server Appliance group at Sun Microsystems, Inc. Before joining Sun, Sriranga was employed at Cisco Systems, Inc. Among other interests, Sriranga enjoys mountain biking, classical music, and playing Marathon with his brother Srivathsa. Sriranga can be reached via e-mail at ranga@soda.berkeley.edu.

Dedication

For my grandmother, who taught me to love the English language.

For my mother, who taught me to love programming languages.

Acknowledgments

Writing a book on shell programming is a daunting task, due to the myriad UNIX versions and shell versions that are available. Thanks to the hard work of my development editor Steve Rowe, my technical editor Michael Watson, and my copy editor Kezia Endsley, I was able to make sure the book covered the material completely and correctly. Their suggestions and comments have helped enormously.

In addition to the technical side of the book, the task of coordinating and managing the publishing process is a difficult one. The assistance of my acquisitions editor, Kathryn Purdum, in handling all of the editorial issues and patiently working with me to keep this book on schedule was invaluable.

Working on a book takes a lot of time and makes it difficult to concentrate on work and family activities. Thanks to the support of my manager, Larry Coryell, my parents, my brother Srivathsa, and my uncle and aunt Srinvasa and Suma, I was able to balance work, family, and authoring.

Thanks to everyone else on the excellent team at Sams who worked on this book. Without their support, this book would not exist.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: opensource@sampublishing.com

Mail: Mark Taber
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Introduction

In recent years, the UNIX operating system has seen a huge boost in its popularity, especially with the emergence of Linux. For programmers and users of UNIX, this comes as no surprise: UNIX was designed to provide an environment that's powerful yet easy to use.

One of the main strengths of UNIX is that it comes with a large collection of standard programs. These programs perform a wide variety of tasks from listing your files to reading e-mail. Unlike other operating systems, one of the key features of UNIX is that these programs can be combined to perform complicated tasks and solve your problems.

One of the most powerful standard programs available in UNIX is the shell. The *shell* is a program that provides a consistent and easy-to-use environment for executing programs in UNIX. If you have ever used a UNIX system, you have interacted with the shell.

The main responsibility of the shell is to read the commands you type and then ask the UNIX kernel to perform these commands. In addition to this, the shell provides several sophisticated programming constructs that enable you to make decisions, repeatedly execute commands, create functions, and store values in variables.

This book concentrates on the standard UNIX shell called the Bourne shell. When Dennis Ritchie and Ken Thompson were developing much of UNIX in the early 1970s, they used a very simple shell. The first real shell, written by Stephen Bourne, appeared in the mid 1970s. The original Bourne shell has changed slightly over the years; some features were added and others were removed, but its syntax and its resulting power have remained the same.

The most attractive feature of the shell is that it enables you to create scripts. *Scripts* are files that contain a list of commands you want to run. Because every script is contained in a file and every file has a name, scripts enable you to combine existing programs to create completely new programs that solve your problems. This book teaches you how to create, execute, modify, and debug shell scripts quickly and easily. After you get used to writing scripts, you will find yourself solving more and more problems with them.

How This Book Is Organized

This book assumes that you have some familiarity with UNIX and know how to log in, create, and edit files, as well as how to work with files and directories to a limited extent. If you haven't used UNIX in a while or you aren't familiar with one of these topics, don't worry; the first part of this book reviews this material thoroughly.

This book is divided into three parts:

- Part I is an introduction to UNIX, the shell, and some common tools.
- Part II covers programming using the shell.
- Part III covers advanced topics in shell programming.

Part I consists of Chapters 1 through 7. The following material is covered in the individual chapters:

- Chapter 1, “Shell Basics,” discusses several important concepts related to the shell and describes the different versions of the shell.
- Chapter 2, “Script Basics,” describes the process of creating and running a shell script. It also covers the login process and the different modes in which the shell executes.
- Chapters 3, “Working with Files,” and 4, “Working with Directories,” provide an overview of the commands used when working with files and directories. These chapters show you how to list the contents of a directory, view the contents of a file, and manipulate files and directories.
- Chapter 5, “Input and Output” covers the `echo`, `printf`, and `read` commands along with the `<` and `>` input redirection operators. This chapter also covers using file descriptors.
- Chapter 6, “Manipulating File Attributes,” introduces the concept of file attributes. It covers the different types of files along with how to modify a file’s permissions.
- Chapter 7, “Processes,” shows you how to start and stop a process. It also explains the term *process ID* and how you can view them.

By this point, you should have a good foundation in the UNIX basics. This will enable you to start writing shell scripts that solve real problems using the concepts covered in Part II. Part II is the heart of this book, consisting of Chapters 8 through 18. It teaches you about all the tools available when programming in the shell. The following material is covered in these chapters:

- Chapter 8, “Variables,” explains the use of variables in shell programming, shows you how to create and delete variables, and explains the concept of environment variables.
- Chapters 9, “Substitution,” and 10, “Quoting,” cover the topics of substitution and quoting. Chapter 9 shows you the four main types of substitution: filename, variable, command, and arithmetic substitution. Chapter 10 shows you the behavior of the different types of quoting and its affect on substitution.

- Chapters 11, “Flow Control,” and 12, “Loops,” provide complete coverage of flow control and looping. The flow control constructs `if` and `case` are covered along with the loop constructs `for` and `while`.
- Chapter 13, “Parameters,” shows you how to write scripts that use command-line arguments. The special variables and the `getopts` command are covered in detail.
- Chapter 14, “Functions,” discusses shell functions. Functions provide a mapping between a name and a set of commands. Learning to use functions in a shell script is a powerful technique that helps you solve complicated problems.
- Chapters 15, “Text Filters,” 16, “Filtering Text with Regular Expressions,” and 17, “Filtering Text with `awk`,” cover text filtering. These chapters show you how to use a variety of UNIX commands including `grep`, `tr`, `sed`, and `awk`.
- Chapter 18, “Other Tools,” provides an introduction to some tools that are used in shell programming. Some of the commands that are discussed include `type`, `find`, `bc`, and `expr`.

At this point, you will know enough about the shell and the external tools available in UNIX that you can solve most problems. The last part of the book, Part III, is designed to help you solve the most difficult problems encountered in shell programming. Part III spans Chapters 19 through 24 and covers the following material:

- Chapter 19, “Signals,” explains the concept of signals and shows you how to deliver a signal and how to deal with a signal using the `trap` command.
- Chapter 20, “Debugging,” discusses the shell’s built-in debugging tools. It shows you how to use syntax checking and shell tracing to track down bugs and fix them.
- Chapters 21, “Problem Solving with Functions,” and 22, “Problem Solving with Shell Scripts,” cover problem solving. Chapter 21 covers problems that can be solved using functions. Chapter 22 introduces some real-world problems and shows you how to solve them using a shell script.
- Chapter 23, “Scripting for Portability,” covers the topic of portability. In this chapter, you will rewrite several scripts from previous chapters to be portable to different versions of UNIX.
- Chapter 24, “Shell Programming FAQs,” is a question-and-answer chapter. Several common programming questions are presented along with detailed answers and examples.

Each chapter in this book includes complete syntax descriptions for the various commands along with several examples to illustrate the use of commands. The examples are designed to show you how to apply the commands to solve real problems. At the end of

each chapter are a few questions that you can use to check your progress. Some of the questions are short answers, whereas others require you to write scripts.

After Chapter 24, four appendixes are available for your reference:

- Appendix A, “Command Quick Reference,” provides a complete command reference.
- Appendix B, “Glossary,” contains the terms used in this book.
- Appendix C, “Answers to Questions,” contains the answers to all the questions in the book.
- Appendix D, “Shell Function Library,” contains a listing of the shell function library discussed in Chapter 21, “Problem Solving with Functions.”

About the Examples

As you work through the chapters, try typing in the examples to get a better feeling for how the computer responds and how each command works. After you get an example working, try experimenting with the example by changing commands. Don’t be afraid to experiment. Experiments (both successes and failures) teach you important things about UNIX and the shell.

Many of the examples and the answers to the questions are available for downloading from the following URL:

<http://www.csua.berkeley.edu/~ranga/downloads/tysp2.tar.Z>

After you have downloaded this file, change to the directory where the file was saved and execute the following commands:

```
$ uncompress tysp2.tar.Z
$ tar -xvf tysp2.tar
```

This creates a directory named `tysp2` that contains the examples from this book.

There is no warranty of any kind on the examples in this book. Much effort has been placed into making the examples as portable as possible. To this end the examples have been tested on the following versions of UNIX:

- Sun Solaris versions 2.5.1 to 8
- Hewlett-Packard HP-UX versions 10.10 to 11.0
- OpenBSD versions 2.6 to 2.9
- Apple MacOS X 10.0 to 10.1.2
- Red Hat Linux versions 4.2, 5.1, 5.2, 6.0, and 6.2
- FreeBSD versions 2.2.6 and 4.0 to 4.3

It is possible that some of the examples might not work on other versions of UNIX. If you encounter a problem or have a suggestion about improvements to the examples or the content of the book, please feel free to contact me at the following e-mail address:

ranga@soda.berkeley.edu

I appreciate any suggestions and feedback you have regarding this book.

Conventions Used in This Book

Features in this book include the following:



Notes give you comments and asides about the topic at hand, as well as full explanations of certain concepts.



Tips provide great shortcuts and hints on how to program in shell more effectively.



Cautions warn you against making your life miserable and avoiding the pitfalls in programming.

NEW TERM

New terms appear in *italic*. Each of the new terms covered in a chapter is listed at the end of that chapter in the “Terms” section.

At the end of each chapter, you’ll find the handy Summary and Quiz sections (with answers found in Appendix C).

In addition, you’ll find various typographic conventions throughout this book:

- Commands, variables, directories, and files appear in text in a special monospaced font.
- Commands and such that you type appear in **boldface type**.
- Placeholders in syntax descriptions appear in a *monospaced italic* typeface. This indicates that you will replace the placeholder with the actual filename, parameter, or other element that it represents.

This page intentionally left blank



PART I

Introduction to UNIX and Shell Tools

Hour

- 1 Shell Basics
- 2 Script Basics
- 3 Working with Files
- 4 Working with Directories
- 5 Input and Output
- 6 Manipulating File Attributes
- 7 Processes

This page intentionally left blank

HOUR 1



Shell Basics

My father is an avid woodworker. He has a tool chest that holds all his woodworking tools, from screwdrivers and chisels to power sanders and power drills. Over the years, he has used his tools to build everything from a toy bridge to a shed. By applying the same tools, he has been able to build all the elements required in his projects.

In many ways, shell programming is similar to woodworking. A woodworking project requires a design for the project and its elements along with the right tools. In shell programming, the project design is provided by the programmer and the tools are *utilities* or *commands* provided by UNIX. There are simple commands such as `ls` and `cd`, and there are also commands such as `awk` and `sed`, which are the power tools in UNIX.

The simple commands are easy to learn. You probably already know how to use many of them. The power tools take longer to learn, but after mastering them almost any problem can be tackled. This book covers both the simple tools and the power tools, with the main focus on the most powerful tool in UNIX, the shell. In this chapter you will learn about

- Simple, complex, and compound commands
- Command separators
- Different types of shells

What Is a Command?

A *command* is a file containing a set of instructions that UNIX can run or *execute*. In operating systems such as Mac OS or Windows, commands are executed by clicking their icons. In UNIX, a command is executed by typing in its name and pressing Enter or Return. For example, in order to execute the date command, use the following:

```
$ date [ENTER]
Wed Dec  9 08:49:13 PST 1998
$
```

The purpose of the date command is to display the current day, date, time, and year. Notice that after the command finishes executing, the character \$ is displayed. This character is the *prompt*. When a prompt is present, the name of a command can be given for execution. The shell reads the command name and tries to execute it. While the command executes, the prompt is not displayed. When the command finishes executing, the prompt is displayed again.



The \$ character is a prompt for you to enter a command. It is not part of the command itself.

For example, to execute the date command, only the word date is typed at the prompt. Don't type \$ date. Some systems might display an error message if you type \$ date instead of date.

Here is another example of executing the who command:

```
$ who
vathsa  tty1    Dec  6 19:36
ranga   tty0     Dec  9 09:23
$
```

The who command displays a list of all the people, or users, who are currently using the UNIX machine. The first column of the output lists the usernames of the people who are logged in. On this system, there are two users, vathsa and ranga. The second column lists the terminals they are logged in to, and the final column lists the time they logged in. The output varies from system to system. On some versions of UNIX or Linux, there might be additional columns in the output. Try it on your system to see who is logged in.

For those readers who are not familiar with the process of logging in to a UNIX system, the details are discussed in Chapter 2, “Script Basics.”

Simple Commands

The commands `who` and `date` are examples of simple commands. A *simple command* is one that can be executed by just specifying the command name at the prompt. The syntax for executing a simple command is

```
$ cmd
```

Here `cmd` is the name of the command to be executed.

Simple commands in UNIX can be small commands such as `who` and `date`, or they can be large commands such as a Web browser or spreadsheet program. Most commands in UNIX can be executed as simple commands.

Complex Commands

A *complex command* consists of a command name followed by a list of arguments. *Arguments* are modifiers specified after the command name and are used to alter the behavior of the command. The syntax for a complex command is

```
$ cmd arg1 arg2 arg3 ... argN
```

Here `cmd` is the name of the command you want to execute, and `arg1` through `argN` are the arguments you want to give `cmd`. As an example, you can use the `who` command to determine information about yourself by executing it as follows:

```
$ who am i
ranga pts/0 Dec 9 08:49
$
```

In this mode, `who` omits information about the other users and just prints information about you. This is an example of a complex command. Here, the `cmd` is `who` and the arguments, `arg1` and `arg2`, are `am` and `i`. These arguments change the behavior of the `who` command. Most commands accept arguments that modify their behavior.



Although you can specify any arguments you want to a command, most commands only understand a handful of arguments. Some commands ignore arguments they do not understand, whereas others display error messages. The `man` command, discussed in Chapter 2, can help determine the arguments a command understands.

When `who` was executed as a simple command, it displayed information about all the users who were logged in. This is referred to as the *default behavior* for the `who` command. The default behavior of a command is the output produced by the command when it is executed as a simple command.

Compound Commands

It is possible to combine simple and complex commands into *compound commands*. A compound command consists of a list of simple and complex commands, with each command separated by a semicolon, `;`. The syntax for a complex command is

```
$ cmd1 ; cmd2 ; cmd3 ; ... ; cmdN ;
```

Here, *cmd1* through *cmdN* are either simple or complex commands. The order of execution is *cmd1*, followed by *cmd2*, followed by *cmd3*, and so on. When *cmdN* finishes executing, the prompt is returned.

An example of a complex command is

```
$ date ; who am i ;
Wed Dec  9 10:10:10 PST 1998
ranga      pts/0          Dec  9 08:49
$
```

Here the compound command consists of the simple command `date` and the complex command `who am i`. The `date` command is executed first, followed by the `who am i` command. The behavior of the previous complex command is the same as if each of the commands were executed as follows:

```
$ date
Wed Dec  9 10:25:34 PST 1998
$ who am i
ranga      pts/0          Dec  9 08:49
$
```

The difference between executing commands in this fashion and using a compound command is that in a compound command, the prompt is returned only after all the commands that compose the complex command have been executed.

Command Separators

The semicolon character (`;`) is treated as a *command separator*. Command separators indicate where one command ends and another begins. If a command separator is not used to separate each of the individual commands in a complex command, the system will not be able to distinguish between the ending of one command and the beginning of the next command.

For example, if the previous example is executed without the first semicolon, such as shown here,

```
$ date who am i
```

the system will produce an error message similar to the following:

```
date: bad conversion
```

In this case, `date` thinks that it is being executed as a complex command with the arguments `who`, `am`, and `i`. The `date` command is confused by these arguments and displays an error message. When using complex commands, remember to use the semicolon character.

You can also terminate individual simple and complex commands using the semicolon character. Both of the following commands produce the same output:

```
$ date  
$ date ;
```

In the first case, the simple command `date` executes, and the prompt returns. In the second case, the shell thinks that a complex command is executing. It begins by executing the first command in the complex command (in this case, `date`). When this command finishes, the shell tries to execute the next command. In this case, no other commands are left to execute, so the prompt returns.



You will frequently see the semicolon used to terminate simple and complex commands in scripts. Because the semicolon is required to terminate commands in other languages, such as C, Perl, and Java, many script programmers use it the same way in scripts. There is no overhead in using the semicolon for this purpose.

What Is the Shell?

The *shell* provides you with an interface to the UNIX system. It reads input from you and executes the programs you specified. While the programs are executing, it displays their output. For this reason, the shell is often referred to as the UNIX system's *command interpreter*. For users familiar with Windows, the UNIX shell is similar to the DOS shell, `COMMAND.COM`.

The real power of the shell lies in the fact that it is much more than a command interpreter. It is also a powerful programming language, complete with conditional statements, loops, and functions.

If you are familiar with these types of statements from other programming languages, you can learn shell programming quickly. If you haven't seen these before, don't fret. By working through the examples and exercises in this book, you will learn how to effectively use all these statements.

The Shell Prompt

The *prompt*, \$, discussed earlier in this chapter, is printed by the shell. When the prompt is displayed, you can type in a command. The shell waits for you to press Enter or Return before reading your input. The command to execute is determined by examining the first *word* of your input. A word is a set of characters separated by a space or tab. The shell treats input as follows:

```
$ word1 word2 word3 ... wordN
```

The first word, *word1*, is always assumed to be the name of the command to execute. If there is only one word, as in the following example, the shell simply executes the command:

```
$ date
```

If there are multiple words as follows,

```
$ who am i
```

the extra words are passed as arguments to the command specified by *word1*.

Different Types of Shells

The prompt on your system might be different from the simple \$ used in this book. The actual prompt that is displayed depends on the type of shell you are using. In UNIX, there are two major types of shells:

- Bourne (includes sh, ksh, bash, and zsh)
- C (includes csh and tcsh)

If you are using most Bourne-type shells, the last character of the default prompt is the dollar sign character, \$. If you are using a C-type shell or zsh, the last character of the default prompt is the percent character, %.

This book covers Bourne-type shells. Unless explicitly noted, the examples and exercise answers in this book will work with any Bourne-type shell. The C-type shells have several problems that make them unsuitable for shell programming, thus they are not covered in this book. For more information on this topic, refer to the following article:

<http://www.faqs.org/faqs/unix-faq/shell/csh-why-not/>



In UNIX, there are two types of accounts, regular accounts and the root account. Normal users are given *regular* accounts. The *root* account is an account with special privileges that the administrator of a UNIX system

(called the *sysadmin*) uses to perform maintenance and upgrades.

When the root account is used, both Bourne-type and C-type shells display the # character as the last character of the prompt.

Use extreme caution when executing commands as the root user because the commands affect the whole system. None of the examples in this book require that you have access to the root account to execute them.

Bourne Shell

The original UNIX shell was written at AT&T Bell Labs in New Jersey during the mid-1970s by Steve Bourne. Because the Bourne shell was the first shell to appear on UNIX systems, it is often referred to as “the shell.” Historically, it was installed as `/bin/sh`.

In addition to being a command interpreter, the Bourne shell is a powerful language with a programming syntax similar to that of the ALGOL language. Steve Bourne had written a ALGOL-68 compiler when he was at Cambridge University in England and liked the syntax of that language so much that he modeled the syntax of the shell after it.

Some of the features of the Bourne shell are

- Process control (see Chapter 7, “Processes”)
- Variables (see Chapter 8, “Variables”)
- Regular expressions (see Chapter 9, “Substitution”)
- Flow control (see Chapter 11, “Flow Control,” and Chapter 12, “Loops”)
- Powerful input and output controls (see Chapter 5, “Input and Output”)
- Functions (see Chapter 14, “Functions”)

One of the main complaints against the Bourne shell is that, although it is excellent for programming, it is hard to use interactively. Some of the major drawbacks are

- Lack of filename completion
- Lack of command history or command editing
- Difficulty in executing multiple background processes

C Shell

The C shell was written at the University of California at Berkeley in the early 1980s by Bill Joy. C shell was designed to make the shell easier to use interactively. It first appeared in BSD UNIX and was later incorporated into AT&T’s version of UNIX. C shell is usually installed as `/bin/csh`.

The C shell updated the shell's syntax from the older ALGOL-like syntax to a more modern C-like syntax. At the time, most people felt that this change would simplify shell programming for Berkeley's UNIX programmers, who were well versed in C and its syntax. As it turned out, C shell could not be used for much more than the most trivial scripts because of the following flaws:

- Weak input and output controls
- Lack of functions
- Confusing syntax

Although the C shell did not catch on for scripts, it has become extremely popular for interactive use. Some of the key improvements responsible for this popularity are:

- *Command History*. Previously executed commands can be recalled for re-execution. The command can also be edited before it is re-executed.
- *Aliases*. C shell allows for the creation of short mnemonic names that can be entered in lieu of the full command names. Aliases are a simplified form of the Bourne shell functions.
- *File Name Completion*. The C shell can automatically complete a filename after a few characters of the file's name have been entered.
- *Job Controls*. The C shell allows for the execution of multiple background processes and allows for their control via the `jobs` command.

The TENEX/TOPS C shell, `tcsh`, is a newer version of the C shell that features several usability enhancements. For example, it can scroll through the command history using the up and down arrow keys and it allows for the editing of commands using right and left arrow keys. For more information on `tcsh`, refer to the following URL:

<http://www.dubois.ws/software/csh-tcsh-book/>

The Korn Shell

For many years, the only shells to choose from were the Bourne shell and the C shell. This meant that most users had to learn two shells, the Bourne shell for programming and the C shell for interactive use. To rectify this situation, David Korn of AT&T Bell Labs wrote the Korn Shell, `ksh`. It incorporates all the C shell's interactive features while preserving the Bourne shell's ALGOL-like syntax. The Korn Shell is usually installed as `/bin/ksh` or `/usr/bin/ksh`.

Some of the additional features that the Korn Shell adds to the Bourne shell are

- Command history and history substitution
- Command aliases and functions

- Filename completion
- Arrays (see Chapter 8)
- Built-in integer arithmetic (see Chapter 9)

In general `ksh` is fully compatible with `sh`. Some minor differences exist that can affect the execution of a script. Where appropriate, such differences are noted in this book.

There are several variants of `ksh`. The official version is pre-installed on most commercial versions of UNIX, such as Solaris and HP-UX. For other systems, it is available in binary form from

<http://www.kornshell.com>

Most non-commercial versions of UNIX, such as Linux and BSD, use the public domain version of the Korn Shell, `pksh`. Eric Gisin created `pksh` using Charles Forsyth's public domain V7 shell along with parts of the BRL shell. Currently, `pksh` is maintained by Michael Rendell. It is available in both source and binary forms from

<http://web.cs.mun.ca/~michael/pdksh/>

For the shell programmer, there is no difference between the official and the public domain versions of `ksh`—scripts that run in one version will run in the other. For users, the official version provides a few nice features such as command line completion with the Tab key rather than the Esc key.

Another variant of `ksh` is the POSIX shell. The Institute of Electrical and Electronics Engineers (IEEE) created the POSIX standards in order to help programmers write portable programs that are compatible with a wide range of systems. One particular standard, the 1003.2/ISO 9945.2 Shell and Tools specification, specifies the syntax and behavior of a portable shell, which is essentially the syntax and behavior of `ksh`. Most commercial UNIX vendors are slowly adapting the POSIX standards. HP is currently shipping the POSIX shell as the default shell, `/bin/sh`, on all of its new HP-UX systems.

Bourne Again Shell

The Bourne Again Shell, `bash`, was written by Brian Fox of the Free Software Foundation as a replacement for the Bourne shell. At present `bash` is maintained by Chet Ramey. It incorporates most of the features of `cs`, `tcsh`, and `ksh` while retaining compatibility with the original Bourne shell and compliance with the POSIX standard.

Most Linux distributions, such as Red Hat, Debian, and Slackware, ship with `bash` installed as `/bin/bash` and `/bin/sh`. Because of licensing restrictions, the original Bourne shell cannot be easily distributed with Linux. Since `bash` is compatible with the Bourne shell, most Linux distributions have chosen to use a copy of `bash` in place of a genuine Bourne shell.

For non-Linux systems, bash is available in both source and binary forms from

<http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html>

Some features that bash includes, in addition to those of the Korn Shell, are

- Name completion for variable names, usernames, hostnames, commands, and file-names
- Spelling correction for pathnames in the cd command
- Arrays of unlimited size
- Integer arithmetic in any base between 2 and 64

The Z Shell

The Z shell, zsh, was written by Paul Falstad while he was a student at Princeton University. It is extremely customizable and is mostly compatible with ksh.

On Mac OS X systems, zsh is installed as `/bin/zsh` and `/bin/sh`. Because of licensing issues, Apple has chosen not to distribute the original Bourne shell with Mac OS X. Apple distributes zsh as its Bourne shell replacement.

For non-Mac OS X systems, zsh is available from

<http://zsh.sunsite.dk/>

In addition to the features of ksh and bash, some additional features of zsh are

- Highly configurable command-line editing
- Fully programmable filename, username, hostname, and history completion
- Highly customizable keyboard mappings

Summary

This chapter covered shell basics, including the execution of simple commands, complex commands, and compound commands. The concept of a shell and several different shells, including ksh, bash, and zsh, were described. The next chapter, “Script Basics,” explores the function of the shell in greater detail, starting with interactive and non-interactive uses of the shell.

Questions

1. Classify each of the following as simple, complex, or compound commands:

```
$ ls
$ date ; uptime
$ ls -l
$ echo "hello world"
```

If you haven't seen some of these commands before, try them out on your system. As you progress through the book, each will be formally introduced.

2. What is the effect of putting a semicolon at the end of a single simple or complex command?

For example, will the output of the following commands be different?

```
$ who am i
$ who am i ;
```

3. What are the two major types of shells? Give an example of a shell that falls into each type.

Terms

Arguments Arguments are command modifiers that change the behavior of a command.

Command Separators A command separator indicates where one command ends and another begins. The most common command separator is the semicolon character (;).

Commands A command is a program that can be executed. To execute a command, type its name and press Enter or Return.

Complex Commands A complex command is a command that consists of a command name and a list of arguments.

Compound Commands A compound command consists of a list of simple and complex commands separated by the semicolon character (;).

Default Behavior The default behavior of a command is the output generated by a command when it is run as a simple command.

Prompt The prompt is displayed by the shell. When the prompt is present, the shell can be given a command to execute. In this book, the \$ character is used to indicate the prompt.

Shell The shell is an interface to the UNIX system. It reads input and executes programs based on that input. When a program has finished executing, it displays that program's output. The shell is sometimes called a command interpreter.

Simple Commands A simple command is a command that can be executed by giving just its name at the prompt.

Words Words are sets of characters separated by spaces and tabs.

HOUR 2



Script Basics

Chapter 1, “Shell Basics,” introduced the concept of a shell and commands, and described how the shell reads input and executes the specified commands. This chapter expands on those basic concepts to explain in greater detail what the shell is and how it works, including the login and logout process as it relates to the shell.

This chapter also explains how to group commands that are normally executed interactively into a file, thus creating a program or script. Scripts are the power behind the shell because they allow commands to be grouped together to create new commands.

Specifically, the topics covered in this chapter are

- The UNIX System
- Shell Initialization
- Getting Help

The UNIX System

The UNIX system consists of two main components:

- Utilities
- Kernel

Utilities are programs that can be executed. The programs `who` and `date` from the previous chapter are examples of utilities.

Commands are slightly different from utilities. The term *utility* refers to the name of a program, whereas the term *command* refers to the program and any arguments are specified to that program in order to change its behavior. For simple commands, the term *command* is sometimes used in place of the term *utility*.

The kernel is the heart of the UNIX system. It provides utilities with a means of accessing the computer's hardware. It also handles scheduling and executing commands.

When a computer is powered off, both the kernel and the utilities are stored on the hard drives. When the computer boots, the kernel is loaded from disk into memory and remains in memory until the computer is turned off. Utilities, on the other hand, are stored in files on disk and loaded into memory only when they are requested for execution. For example, when the following command is executed,

```
$ who
```

the kernel loads the `who` command from a file on disk, places it in memory, and starts executing it. When the program finishes executing, it remains in the machine's memory for a short period of time before it is removed. This enables frequently used commands to execute faster. Consider what happens when the `date` command is executed three times in quick succession:

```
$ date
Sun Dec 27 09:42:37 PST 1998
$ date
Sun Dec 27 09:42:38 PST 1998
$ date
Sun Dec 27 09:42:39 PST 1998
```

The first time the `date` command might need to be loaded from the computer's hard disk, but the second and third time the `date` command usually remains in the computer's memory, allowing it to execute faster. Try it on your system and see if you notice a slight delay the first time and no delay the second and third times.

Commands and Files

In UNIX most commands are stored in separate files on disk. For example, the `who` and `date` commands are stored in two separate files named `who` and `date` on the disk; they are not part of the shell. This allows for new commands to be added and bugs in existing commands to be fixed without modifying the shell.

If you are unfamiliar with the concept of a file, don't panic! Files are covered in the next chapter.

Logging In

Just like `date`, the shell is a program that is stored on disk. The main difference is that the shell is loaded into memory when you log in and stays in memory until you log out.

When you first connect to a UNIX system, a login prompt will be presented. Usually it looks similar to

```
login:
```

Here you need to enter a *username*, which is your identity on a UNIX system. After entering a username, another prompt will be presented:

```
login: ranga  
Password:
```

Here you need to enter the password corresponding to the username you entered. Your username, password, and associated files are called your *user account*. The system administrator is responsible for creating your user account and providing you with the username and password associated with it.

After reading both the username and password, the system looks through the user database, normally located in the file `/etc/passwd`, for an entry matching the information that was provided. If a match is found, the shell associated with that entry is executed; otherwise, an error is displayed.



For those of you who are not familiar with UNIX file and directory names, such as `/etc/passwd`, these topics are covered in Chapter 3, "Working with Files," and Chapter 4, "Working with Directories."

Files and directories are discussed very briefly in this chapter. A general idea about files and directories from other operating systems is sufficient to understand the examples.

The following is a sample entry from `/etc/passwd` on my system:

```
ranga:x:500:100:Sriranga Veeraraghavan:/home/ranga:/bin/bash
```

The entry is composed of several fields, each separated from the other fields by a colon, `:`. Later chapters will explain the information stored in each field. For now, only the last two fields are important. The last field stores the shell associated with the account. The second from the last field stores the *home directory* for the account. The home directory is where you first start out after logging in. In some documentation, you will see home directories denoted by a tilde, `~`, or a tilde followed by a slash, `~/`. In the previous example, the shell is `/bin/bash` and the home directory is `/home/ranga`. Your shell and home directory will most likely be different.

In most cases, the system administrator will assign you the default shell for a particular version of UNIX. In some cases, there are two defaults and the system administrator can choose between them based on his personal preferences.

The default shells for some common versions of UNIX are as follows:

- Solaris uses Bourne shell or C Shell.
- HP-UX uses POSIX shell.
- BSD uses Korn Shell or C Shell.
- Mac OS X uses Z Shell or C Shell.
- Linux uses the Bourne Again Shell.

For the sake of brevity, we assume that you have been assigned Bourne shell, Korn Shell (ksh), Bourne Again Shell (bash), or Z Shell (zsh) as your shell.

Shell Modes and Initialization

In this section, we will first discuss the startup procedure for the various Bourne-type shells, and then we will examine the different modes of execution for a shell.

Initialization Procedures

After you log in, a shell is executed on your behalf. When this shell starts executing, it is *uninitialized*. In this state, several parameters required for its proper operation are not defined. The shell undergoes a process called *initialization* that defines these parameters. The steps and files involved in initialization are different in each shell, so we will examine the process used by each of the Bourne-type shells individually. In general each of the shells uses default or system-wide configuration files located in the `/etc` directory along with a set of personal configuration files located in your home directory.

Bourne Shell

Bourne shell initialization has four steps and involves the initialization files (also called init files) `/etc/profile` and `.profile`. The process is as follows:

1. The shell checks to see whether the file `/etc/profile` exists.
2. If it exists, the shell reads it; otherwise, the shell skips it.
3. The shell checks to see whether the file `.profile` exists in your home directory.
4. If it exists, the shell reads it; otherwise, the shell skips it.

After these steps have been performed, the prompt is displayed. The default prompt for Bourne shell is `$` (a dollar sign followed by a space).

Korn Shell

Korn Shell (`ksh`) closely resembles Bourne shell initialization. It has six steps and involves the init files `/etc/profile`, `.profile`, and `.kshrc`:

1. `ksh` checks to see whether the file `/etc/profile` exists.
2. If it exists, `ksh` reads it; otherwise, `ksh` skips it.
3. `ksh` checks to see whether the file `.profile` exists in your home directory.
4. If it exists, `ksh` reads it; otherwise, `ksh` skips it.
5. `ksh` checks to see whether the file `.kshrc` exists in your home directory.
6. If it exists, `ksh` reads it; otherwise, `ksh` skips it.

After these steps have been performed, the prompt is displayed. The default prompt for `ksh` is `$` (a dollar sign followed by a space).

Bourne Again Shell

Bourne Again shell (`bash`) initialization is a bit longer than Korn shell and Bourne shell initialization. It has eight steps and involves the init files `/etc/profile`, `.bash_profile`, `.bash_login`, and `.profile`:

1. `bash` checks to see whether the file `/etc/profile` exists.
2. If it exists, `bash` reads it; otherwise, `bash` skips it.
3. `bash` checks to see whether the file `.bash_profile` exists in your home directory.
4. If it exists, `bash` reads it; otherwise, `bash` skips it.
5. `bash` checks to see whether the file `.bash_login` exists in your home directory.
6. If it exists, `bash` reads it; otherwise, `bash` skips it.
7. `bash` checks to see whether the file `.profile` exists in your home directory.
8. If it exists, `bash` reads it; otherwise, `bash` skips it.

After these steps have been performed, a prompt is displayed. The default prompt for bash is `bash$` (the string `bash$` followed by a space).

Z Shell

Z shell (`zsh`) initialization is quite long and does not resemble the initialization process of the other shells. It has 16 steps and involves the init files `/etc/zshenv`, `.zshenv`, `/etc/zprofile`, `.zprofile`, `/etc/zlogin`, and `.zlogin`:

1. `zsh` checks to see whether the file `/etc/zshenv` exists.
2. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
3. `zsh` checks to see whether the file `.zshenv` exists in your home directory.
4. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
5. `zsh` checks to see whether the file `/etc/zprofile` exists.
6. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
7. `zsh` checks to see whether the file `.zprofile` exists in your home directory.
8. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
9. `zsh` checks to see whether the file `/etc/zshrc` exists.
10. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
11. `zsh` checks to see whether the file `.zshrc` exists in your home directory.
12. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
13. `zsh` checks to see whether the file `/etc/zlogin` exists.
14. If it exists, `zsh` reads it; otherwise, `zsh` skips it.
15. `zsh` checks to see whether the file `.zlogin` exists in your home directory.
16. If it exists, `zsh` reads it; otherwise, `zsh` skips it.

After these steps have been performed, a prompt is displayed. The default prompt for `zsh` is `host%`. Here `host` is the hostname of your system. For example, on a system named `mars`, the default `zsh` prompt would be `mars%`.

Initialization File Contents

Usually a shell's init files are quite short. The purpose of these files is to provide a complete working environment with as little overhead as possible. In this section, we will look at the basic settings required for Bourne shell. If you are using a different shell, you can put these settings into an init file used by that shell.

The init file `.profile` contains all of your shell initialization settings. You can add as much customization information as you want to this file. The minimum set of information that you need to configure includes

- A list of directories in which to locate commands
- A list of directories in which to locate manual pages for commands

Setting PATH

When you type the command,

```
$ date
```

the shell has to locate the command `date` before it can be executed. The `PATH` variable specifies the directories in which the shell should look for commands. The most basic setting is as follows:

```
PATH=/bin:/usr/bin
```

Each of the individual entries separated by the colon character, `:`, should be directories. Directories are discussed in Chapter 4.

If you request the shell to execute a command and it cannot find it in any of the directories given in the `PATH` variable, a message similar to the following appears:

```
$ hello
hello: not found
```

Setting MANPATH

In UNIX, online help has been available since the beginning. The next section, “Getting Help,” discusses how to access the online help using the `man` command. In order for this command to work properly, you have to tell the shell where the help pages are located. This information is specified using the `MANPATH`. A common setting is

```
MANPATH=/usr/man:/usr/share/man
```

Similar to the path, each of the individual entries separated by the colon character, `:`, are directories.

When you use the `man` command to request online help, it searches every directory given in the `MANPATH` for an online help page corresponding to the topic you requested. For example, the command

```
$ man who
```

looks for the online help page corresponding to the `who` command. If this page is found, it is displayed.

Interactive and Non-Interactive Shells

Shell can run in two different modes: interactive and non-interactive. In *interactive mode*, the shell expects to read input from you and execute the commands that you specify. This mode is called interactive because it interacts with the user. In *non-interactive mode*, the shell does not interact with the user; instead it reads commands stored in a file and executes them. When it reaches the end of the file, it exits.

Most people are familiar with interactive mode: log in, execute some commands in the shell, and log out.

Starting an Interactive Shell

To start a shell in interactive mode, you can type in its name at the prompt. For example, the following command starts bash in interactive mode:

```
$ /bin/bash
bash$
```

The first prompt, \$, that is displayed by the shell started on your behalf when you logged in; the second prompt, bash\$, is displayed by the bash you started.

At this point, we have two interactive shells: The first one is waiting for the other to finish. At first glance, this does not sound extremely useful, but there are cases in which it can be quite helpful. For example, if you need to make changes to the shell's settings, the easiest way to test your changes is to start another shell, perform and verify the changes, and then exit back to the original, unaltered shell.

To exit from the second shell, you can use the `exit` command:

```
bash$ exit
$
```

This returns you to the original shell. If you type `exit` here, the system will log you out. The `exit` command works in all Bourne-type shells.

Starting a Non-Interactive Shell

You can start a shell in non-interactive mode as follows:

```
$ /bin/sh filename
```

Here *filename* is the name of a file that contains commands to execute. As an example, consider the compound command:

```
$ date ; who
```

Let's put these commands into a file called `logins`. First open a file called `logins` in an editor and type in this command and save the file. Now you can execute the commands in this file using the command:

```
$ /bin/sh logins
```

This executes the compound command and displays its output. This is the first example of a shell script or shell program. Basically, a *shell script* is a file that contains a list of commands. When the shell executes the commands contained in the file, it does so without interacting with the user. For this reason, when the shell is used to execute a shell script, it is said to execute in non-interactive mode.

Making a Shell Script Executable

One of the most important tasks in writing shell scripts is making the shell script executable and making sure that the correct shell is invoked on the script.

In a previous example, you created the `logins` script that executes the following compound command:

```
date ; who ;
```

If you wanted to run the script by just typing its name, you need to do two things:

- Mark the file as executable.
- Make sure that the right shell is used to execute the script.

To make this script executable, you need to execute a command of the form:

```
chmod a+x filename
```

The `chmod` command, when used in this form, marks the file specified by *filename* as executable. For a complete discussion of `chmod` and its function, see Chapter 6, “Manipulating File Attributes.” As an example, the following command marks the file `logins` executable:

```
$ chmod a+x $logins
```

To ensure that the correct shell is used to execute a script, you must add a *magic* line, of the following form, as the first line of the script:

```
#!shell
```

Here *shell* is the name of the shell that should be used to execute the script. In most cases, you will want to use `/bin/sh` as *shell*, but if you want to use `ksh` for your scripts, you can specify `/bin/ksh` instead. Without a magic line, the current shell is always used

to evaluate a script, regardless of which shell the script was written for. If you omit the magic line from your scripts, `csh` and `tcsh` users might not be able to run them correctly.

The Magic of `#!/bin/sh`

The `#!/bin/sh` must be the first line of a shell script in order for `sh` to be used to run the script. If this appears on any other line, it is treated as a comment and ignored by all shells.

After this addition, the `logins` script contains two lines:

```
#!/bin/sh
date ; who ;
```

Now it is possible to execute the script by just typing in its name:

```
$ logins
Tue Sep 18 18:44:12 PDT 2001
ranga console Sep 12 10:22
```

Comments

The magic first line for the shell script, `#!/bin/sh`, introduces the concept of comments. A *comment* is a statement embedded in a shell script that is not intended for execution by the shell. In shell scripts, comments start with the `#` character. Everything between the `#` and end of the line are considered part of the comment and are ignored by the shell.

Adding comments to a script is quite simple: Open the script using an editor and add lines that start with the `#` character. For example, to add the following line to the `logins` shell script,

```
# print out the date and who's logged on
```

you can open the file `logins` with an editor and insert this line as the second line in the file. Now the script has three lines:

```
#!/bin/sh
# print out the date and who's logged on
date ; who ;
```

There is no change in the output of the script because comments are ignored. Comments do not slow down a script because the shell just skips them.

You can also add comments to lines that contain commands by adding the `#` character after the commands. For example, you can add a comment to the line `date ; who ;` as follows:

```
date ; who ; # execute the date and who commands
```

When you are writing a shell script, make sure to use comments to explain what the script is doing. If someone else has to look at your shell script, it will help him to understand how your script functions. Comments can also help you figure out what your script is doing, months or years after you wrote it.

Getting Help

As you read through this book, you will want to get more information about the commands and features that are discussed. Much of this information is available by using the online help features of UNIX. Some other resources include Web sites that cover shell programming and Usenet newsgroups.

man

Every version of UNIX comes with an extensive collection of online help pages called *man pages* (short for *manual pages*). The man pages are the authoritative source about your UNIX system. They contain complete information about both the kernel and all the utilities.

You can access man pages by using the `man` command:

```
man cmd
```

Here, *cmd* is the name of a command that you want more information about. As an example,

```
$ man uptime
```

displays the following man page on a Solaris machine:

```
User Commands                               uptime(1)
```

NAME

```
uptime - show how long the system has been up
```

SYNOPSIS

```
uptime
```

DESCRIPTION

```
The uptime command prints the current time, the length of time the system has been up, and the average number of jobs in the run queue over the last 1, 5 and 15 minutes. It is, essentially, the first line of a w(1) command.
```

EXAMPLE

```
Below is an example of the output uptime provides:  
example% uptime
```



```
10:47am up 27 day(s), 50 mins, 1 user,
↳load average: 0.18, 0.26, 0.20
```

SEE ALSO

w(1), who(1), whodo(1M), attributes(5)

NOTES

who -b gives the time the system was last booted.

Man Page Sections

As you can see from the output in the previous example, a man page is divided into several sections that are described in Table 2.1. Almost every man page will include these sections. The content and style of the material in the sections differs from system to system.

TABLE 2.1 Sections in a Man Page

<i>Section</i>	<i>Description</i>
NAME	This section gives the name of the command along with a short description of it.
SYNOPSIS	This section describes all the different modes in which the command can be run. If a command accepts arguments, they are shown in this section.
DESCRIPTION	This section includes a verbose description of the command. If a command accepts arguments, each argument will be fully explained in this section.
EXAMPLE	This section contains an example demonstrating how to execute the command. It might also contain some sample output. Not all man pages contain this section.
SEE ALSO	This section lists other commands that are related to the command.
NOTES	This section usually lists some additional information about the command. Sometimes it lists the known bugs.

Most man pages include all the sections given in Table 2.1 and might include one or two optional sections described in Table 2.2.

TABLE 2.2 Optional Sections Found in Man Pages

<i>Section</i>	<i>Description</i>
AVAILABILITY	This section describes the versions of UNIX that include support for a given command. Sometimes it lists the optional software packages you need to purchase from the vendor to gain extra functionality from a command.
KNOWN BUGS	This section usually lists one or more known problems with the command. If you encounter a problem that is not included in this section, it should be reported to the vendor or author.

TABLE 2.2 continued

<i>Section</i>	<i>Description</i>
FILES	This section lists the files that are required for the command to function correctly. It might also list the files that can be used to configure a command.
AUTHORS or CONTACTS	These sections list the commands' author or authors and provide contact information such as e-mail or postal addresses.
STANDARDS COMPLIANCE	If the behavior of a command is specified by a standards organization such as ISO (International Standards Organization), IEEE (Institute of Electrical and Electronic Engineers), or ANSI (American National Standards Institute), this section lists the relevant standard or standards.

Try using the `man` command to get more information on some of the commands discussed in this chapter.

If the `man` command cannot find a man page corresponding to the command you requested, it issues an error message. For example, the command

```
$ man apple
```

produces an error message similar to the following on my system:

```
No manual entry for apple
```

The exact error message depends on your version of UNIX.

UNIX System Manuals

The term manual page comes from the original versions of UNIX, when the online pages were available as large bound manuals. In all, there were eight different manuals covering the main topics of the UNIX system. These manuals are described in Table 2.3.

TABLE 2.3 The UNIX System Manuals

<i>Manual Section</i>	<i>Description</i>
1	Covers commands.
2	Covers UNIX <i>system calls</i> . System calls are used inside a program, such as <code>date</code> , to ask the kernel for a service.
3	Covers libraries. Libraries are used to store non-kernel-related functions used by C programmers.
4	Covers file formats. For example, the format file <code>/etc/passwd</code> is documented in this section.

TABLE 2.3 continued

<i>Manual Section</i>	<i>Description</i>
5	A secondary section that covers file formats.
6	Includes instructions for playing games on UNIX. (UNIX wasn't always a serious academic and business operating system; it started out as a gaming platform!)
7	Covers device drivers.
8	Covers system maintenance.

In the printed version, you had to know the section where you needed to look for a particular manual page. The big advantage of `man` over the printed manual is that `man` looks in all the sections of the manual for the information you requested, making it much easier to get help.

Online Resources

In addition to `man`, there are several Usenet newsgroups, Web sites, and e-mail lists that are good sources of information about the different shells and shell programming.

The main Usenet newsgroup for shell programming questions and information is `comp.unix.shell`. Before posting a question, you should read the *frequently asked questions (FAQ)* for the newsgroup. The FAQ is located at

<http://www.faqs.org/faqs/unix-faq/shell/intro/>

Often you will find that your question, or something very similar to it, has an answer in this or one of the other FAQ's mentioned later.

For questions regarding the Bourne Again shell (`bash`), you can subscribe to the `bash` e-mail list: `bug-bash@gnu.org`. A subscription form is located at

<http://mail.gnu.org/mailman/listinfo/bug-bash>

If you prefer reading news to e-mail, the e-mail list is available as the newsgroup `gnu.bash.bug`. Before posting to the newsgroup, you should read the `bash` FAQ located at

<http://www.faqs.org/faqs/unix-faq/shell/bash/>

For questions regarding the Z Shell (`zsh`), you can subscribe to the `zsh-users` mailing list: `zsh-users@sunsite.dk`. Subscription instructions can be found at

<http://zsh.sunsite.dk/Arc/mlist.html>

Before posting to the mailing list, you should read the zsh FAQ located at

<http://www.faqs.org/faqs/unix-faq/shell/zsh/>

The following Web sites are also excellent references for shell programming:

<http://www-h.eng.cam.ac.uk/help/tp1/unix/scripts/scripts.html>

<http://www.shelldorado.com/>

The first site contains a tutorial on shell programming written by Tim Love at Cambridge University. The second site is an archive of shell scripts and shell programming information maintained by Heiner Steven of Sun Microsystems.

Summary

This chapter covered what the shell is and how it operates in greater detail. The init process for the various shells was described along with a brief description of the basic information required in the init file `.profile`. The different modes of operation for the shell, interactive and non-interactive, were also covered. Shell programming relies on the non-interactive mode because it enables commands specified in a file to be executed.

We also covered `man` and `man` pages, the online help system in UNIX. Finally, online sources for shell programming information, such as Usenet newsgroups, Web sites, and e-mail lists, were covered.

The next chapter formally introduces the concept of files by showing you how to list files, view the contents of files, and manipulate files.

Questions

1. What are the two files used by the shell to initialize itself?
2. Why do you need to set `PATH` and `MANPATH`?
3. What is the purpose of the following line in a shell script?

```
#!/bin/sh
```
4. What command should you use to access the online help?

Terms

Commands A command is comprised of the name of a program along with zero or more arguments. You might see the term `command` used instead of the term `utility` for simple commands, where only the program name is given.

Comments A comment is a statement that is embedded in a shell script but is not executed by the shell.

Home Directory The home directory is where you first start out after logging in.

Interactive Mode In interactive mode, the shell reads input from the user and executes the specified commands. This mode is called interactive because the shell is interacting with a user.

Kernel The kernel is the heart of the UNIX system. It provides utilities with a means of accessing a machine's hardware. It also handles scheduling and executing commands.

Man Pages Every version of UNIX comes with an extensive collection of online help pages called man pages (short for manual pages). The man pages are the authoritative source about your UNIX system. They contain complete information about both the kernel and all the utilities.

Non-interactive Mode In non-interactive mode, the shell does not interact with the user; instead it reads commands stored in a file and executes them. When the shell reaches the end of the file, it exits.

Shell Initialization After a shell is started, it undergoes a phase called initialization in which important parameters are set up.

Shell Script A shell script is a list of commands stored in a file.

Uninitialized Shell An uninitialized shell is one that has not yet read its init files in order to set up the parameters required for its proper operation.

Utilities Utilities are programs, such as `who` and `date`, that can be executed.

HOUR 3



Working with Files

In UNIX there are two basic types of files: ordinary and special. An *ordinary* file contains data, text, or program instructions. Almost all of the files on a UNIX system are ordinary files. This chapter covers operations on ordinary files.

Special files are mainly used to provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Some special files are similar to aliases or shortcuts and enable you to access a single file using different names. Special files are covered in Chapter 6, “Manipulating File Attributes.”

Both ordinary and special files are stored in directories. *Directories* are similar to folders in the Mac OS or Windows, and they are covered in detail in Chapter 4, “Working with Directories.”

In this chapter, we will examine ordinary files, concentrating on the following topics:

- Listing files
- File contents
- Manipulating files

Listing Files

We'll start by using the `ls` (short for **list**) command to list the contents of the current directory:

```
$ ls
```

The output will be similar to the following:

```
Desktop      Icon          Music         Sites
Documents    Library       Pictures       Temporary Items
Downloads     Movies        Public
```

We can tell that several items are in the current directory, but this output does not tell us whether these items are files or directories. To find out which of the items are files and which are directories, we can specify the `-F` option to `ls`. An option is an argument that starts with the hyphen or dash character, '-'.

The following example illustrates the use of the `-F` option of `ls`:

```
$ ls -F
```

Now the output for the directory is slightly different:

```
Desktop/      Icon          Music/         Sites/
Documents/    Library/      Pictures/      Temporary Items/
Downloads/     Movies/       Public/
```

As you can see, some of the items now have a `/` at the end, indicating each of these items is a directory. The other items, such as `icon`, have no character appended to them. This indicates that they are ordinary files.

When the `-F` option is specified to `ls`, it appends a character indicating the file type of each of the items it lists. The exact character depends on your version of `ls`. For ordinary files, no character is appended. For special files, a character such as `!`, `@`, or `#` is appended to the filename. For more information on the `-F` options, check the UNIX manual page for the `ls` command. You can do this as follows:

```
$ man ls
```

Options Are Case Sensitive

The options that can be specified to a command, such as `ls`, are case sensitive. When specifying an option, you need to make sure that you have specified the correct case for the option. For example, the output from the `-F` option to `ls` is different from the output produced when the `-f` option is specified.

So far, you have seen `ls` list more than one file on a line. Although this is fine for humans reading the output, it is hard to manipulate in a shell script. Shell scripts are geared toward dealing with lines of text, not the individual words on a line. Although external tools, such as the `awk` language covered in Chapter 17, “Filtering Text with `awk`,” can be used to deal with multiple words on a line, it is much easier to manipulate the output when each file is listed on a separate line. You can modify the output of `ls` to this format by using the `-1` option. For example,

```
$ ls -1
```

produces the following listing:

```
Desktop
Documents
Downloads
Icon
Library
Movies
Music
Pictures
Public
Sites
Temporary Items
```

Hidden Files

In the examples you have seen thus far, the output has listed only the visible files and directories. You can also use `ls` to list invisible or hidden files and directories. An *invisible* or *hidden* file is one whose first character is a dot or period (`.`). Many programs, including the shell, use such files to store configuration information. Some common examples of invisible files include

- `.profile`, the Bourne shell (`sh`) initialization script
- `.kshrc`, the Korn Shell (`ksh`) initialization script
- `.cshrc`, the C Shell (`csh`) initialization script
- `.rhosts`, the remote shell configuration file

All files that do not start with the `.` character are considered *visible*.

To list invisible files, specify the `-a` option to `ls`:

```
$ ls -a
```

The directory listing now resembles this:

```
.                .FBCLockFolder  Icon             Public
..               .ssh            Library          Sites
.CFUserTextEncoding Desktop          Movies           Temporary Items
```



```
.DS_Store      Documents      Music
.FBCIndex      Downloads      Pictures
```

As you can see, this directory contains several invisible files.

Notice that in this output, the file type information is missing. To get the file type information, specify the `-F` and the `-a` options as follows:

```
$ ls -a -F
```

The output changes to the following:

```
./              .ssh/          Movies/
../            Desktop/       Music/
.CFUserTextEncoding Documents/     Pictures/
.DS_Store      Downloads/     Public/
.FBCIndex      Icon?          Sites/
.FBCLockFolder/ Library/       Temporary Items/
```

With the file type information, you see that there are two invisible directories (`.` and `..`). These directories are special entries present in all directories. The first one, `..`, represents the current directory, whereas the second one, `.`, represents the parent directory. These concepts are discussed in greater detail in Chapter 4.

Option Grouping

In the previous example, you specified the options to `ls` separately. You could have grouped the options together, as follows:

```
$ ls -aF
$ ls -Fa
```

Both of these commands are equivalent to the following command:

```
$ ls -a -F
```

The order of the options does not matter to `ls`. As an example of option grouping, consider the following equivalent commands:

```
ls -1 -a -F
ls -1aF
ls -a1F
ls -Fa1
```

All permutations of the options `-1`, `-a`, and `-F` produce the same output:

```
./
../
.CFUserTextEncoding
.DS_Store
```

```
.FBCIndex
.FBCLockFolder/
.ssh/
Desktop/
Documents/
Downloads/
Icon?
Library/
Movies/
Music/
Pictures/
Public/
Sites/
Temporary Items/
```

File Contents

In the last section we looked at listing files and directories with the `ls` command. In this section we will look at the `cat` and `wc` commands. The `cat` command lets you view the contents of a file. The `wc` command gives you information about the number of words and lines in a file.

cat

To view the contents of a file, we can use the `cat` (short for **concatenate**) command as follows:

```
cat [opts] file1 ... fileN
```

Here `opts` are one or more of the options understood by `cat`, and `file1...fileN` are the names of the files whose contents should be printed. The options, `opts`, are optional and can be omitted. Two commonly used options are discussed later in this section.

The following example illustrates the use of `cat`:

```
$ cat fruits
```

This command prints the contents of a file called `fruits`:

Fruit	Price/lbs	Quantity
Banana	\$0.89	100
Peach	\$0.79	65
Kiwi	\$1.50	22
Pineapple	\$1.29	35
Apple	\$0.99	78

If more than one file is specified, the output includes the contents of both files concatenated together. For example, the following command outputs the contents of the files `fruits` and `users`:

```
$ cat fruits users
Fruit      Price/lbs    Quantity
Banana     $0.89        100
Peach      $0.79        65
Kiwi       $1.50        22
Pineapple  $1.29        35
Apple      $0.99        78

ranga
vathsa
amma
```

Numbering Lines

The `-n` option of `cat` will number each line of output. It can be used as follows:

```
$ cat -n fruits
```

This produces the output

```
 1 Fruit      Price/lbs    Quantity
 2 Banana     $0.89        100
 3 Peach      $0.79        65
 4 Kiwi       $1.50        22
 5 Pineapple  $1.29        35
 6 Apple      $0.99        78
 7
```

From this output, you can see that the last line in this file is blank. We can ask `cat` to skip numbering blank lines using the `-b` option as follows:

```
$ cat -b fruits
```

Now the output resembles the following:

```
 1 Fruit      Price/lbs    Quantity
 2 Banana     $0.89        100
 3 Peach      $0.79        65
 4 Kiwi       $1.50        22
 5 Pineapple  $1.29        35
 6 Apple      $0.99        78

```

The blank line is still presented in the output, but it is not numbered. If the blank line occurs in the middle of a file, it is printed but not numbered:

```
$ cat -b hosts
 1 127.0.0.1    localhost    loopback

 2 128.32.43.52 soda.berkeley.edu  soda
```

If multiple files are specified, the contents of the files are concatenated in the output, but line numbering is restarted at 1 for each file. As an illustration, the following command,

```
$ cat -b fruits users
```

produces the output

```

1 Fruit           Price/lbs      Quantity
2 Banana          $0.89          100
3 Peach           $0.79          65
4 Kiwi            $1.50          22
5 Pineapple       $1.29          35
6 Apple           $0.99          78

1 ranga
2 vathsa
3 amma
```

WC

Now let's look at getting some information about the contents of a file. Using the `wc` command (short for **w**ord **c**ount), we can get a count of the total number of lines, words, and characters contained in a file. The basic syntax of this command is:

```
wc [opts] files
```

Here `opts` are one or more of the options given in Table 3.1, and `files` are the files you want examined. The options, `opts`, are optional and can be omitted.

TABLE 3.1 `wc` Options

<i>Option</i>	<i>Description</i>
-l	Count of the number of lines.
-w	Count of the number of words.
-m	Count of the number of characters. This option is available on Mac OS X, OpenBSD, Solaris, and HP-UX. This option is not available on FreeBSD and Linux systems.
-c	Count of the number of characters. This option is the Linux and FreeBSD equivalents of the <code>-m</code> option.

When no options are specified, the default behavior of `wc` is to print out a summary of the number of lines, words, and characters contained in a file. For example, the command

```
$ wc fruits
```

produces the following output:

```
      8      18     219 fruits
```

The first number, in this case 8, is the number of lines in the file. The second number, in this case 18, is the number of words in the file. The third number, in this case 219, is the number of characters in the file. At the end of the line, the filename is listed. When multiple files are specified, the filename helps to identify the information associated with a particular file.

If more than one file is specified, `wc` gives the counts for each file along with a total. For example, the command

```
$ wc fruits users
```

produces output similar to the following:

```
      8      18     219 fruits
      3       3      18 users
     11      21     237 total
```

The output on your system might be slightly different.

Counting Lines

To count the number of lines, the `-l` (as in lines) option can be used. For example, the command

```
$ wc -l fruits
```

produces the output

```
      8 fruits
```

The first number, in this case 8, is the number of lines in the file. The name of the file is listed at the end of the line.

When multiple files are specified, the number of lines in each file is listed along with the total number of lines in all of the specified files. As an example, the command

```
$ wc -l fruits users
```

produces the output

```
      8 fruits
      3 users
     11 total
```

Counting Words

To count the number of words in a file, the `-w` (as in words) option can be used. For example, the command

```
$ wc -w fruits
```

produces the output

```
18 hosts
```

The first number, in this case 18, is the number of words in the file. The name of the file is listed at the end of the line.

When multiple files are specified, the number of words in each file is listed along with the total number of words in all of the specified files. As an example, the command

```
$ wc -w fruits users
```

produces the output

```
18 fruits
 3 users
21 total
```

Counting Characters

To count the number of characters, we need to use either the `-m` or the `-c` option. The `-m` option is available on Mac OS X, OpenBSD, Solaris, and HP-UX. On FreeBSD and Linux systems, the `-c` option should be used instead.

For example, on Solaris the command

```
$ wc -m fruits
```

produces the output

```
219 fruits
```

The same output is produced on Linux and FreeBSD systems using the command

```
$ wc -c fruits
```

The first number, in this case 219, is the number of characters in the file. The name of the file is listed at the end of the line.

When multiple files are specified, the number of characters in each file is listed along with the total number of characters in all the specified files. As an example, the command

```
$ wc -m fruits users
```

produces the output

```
219 hosts
 18 users
237 total
```

Combining Options

The options to `wc` can be grouped together and specified in any order. For example, to obtain a count of the number of lines and words in the file `fruits`, we can use any of the following commands:

```
$ wc -w -l fruits
$ wc -l -w fruits
$ wc -wl fruits
$ wc -lw fruits
```

The output from each of these commands is identical:

```
8      18 fruits
```

The output lists the number of words in the files, followed by the number of lines in the file. The filename is specified at the end of the line. When multiple files are specified, the information for each file is listed along with the appropriate total values.

Manipulating Files

In the preceding sections, you looked at listing files and viewing their content. In this section, you will look at copying, renaming, and removing files using the `cp`, `mv`, and `rm` commands.

Copying Files (`cp`)

The `cp` command (short for copy) is used to make a copy of a file. The basic syntax of the command is

```
cp src dest
```

Here `src` is the name of the file to be copied (the source) and `dest` is the name of the copy (the destination). For example, the following command creates a copy of the file `fruits` in a file named `fruits.sav`:

```
$ cp fruits fruits.sav
```

If `dest` is the name of a directory, a copy with the same name as `src` is created in `dest`. For example, the command

```
$ cp fruits Documents/
```

creates a copy of the file `fruits` in the directory `Documents`.

It is also possible to specify multiple source files to `cp`, provided that the destination, `dest`, is a directory. The syntax for copying multiple files is

```
$ cp src1 ... srcN dest
```

Here `src1 ... srcN` are the source files and `dest` is the destination directory. As an example, the following command

```
$ cp fruits users Documents/
```

creates a copy the files `fruits` and `users` in the directory `Documents`.

Interactive Mode

The default behavior of `cp` is to automatically overwrite the destination file if it exists. This behavior can lead to problems. The `-i` option (short for interactive) can be used to prevent such problems. In interactive mode, `cp` prompts for confirmation before overwriting any files.

Assuming that the file `fruits.sav` exists, the following command

```
$ cp -i fruits fruits.sav
```

results in a prompt similar to the following:

```
overwrite fruits.sav? (y/n)
```

If `y` (yes) is chosen, the file `fruits.sav` is overwritten; otherwise the file is untouched. The actual prompt varies among the different versions of UNIX.

Common Errors

When an error is encountered, `cp` generates a message. Some common error conditions follow:

- The source, `src`, is a directory.
- The source, `src`, does not exist.
- The destination, `dest`, is not a directory when multiple sources, `src1 ... srcN`, are specified.
- A non-existent destination, `dest`, is specified along with multiple sources, `src1 ... srcN`.
- One of the sources in `src1 ... srcN` is not a file.

The first error type is illustrated by the following command:

```
$ cp Downloads/ fruits
```

Because `src` (`Downloads` in this case) is a directory, an error message similar to the following is generated:


```
cp: Downloads: is a directory
```

In this example, *dest* was the file *fruits*; the same error would have been generated if *dest* was a directory.

The second error type is illustrated by the following command:

```
$ cp fritus fruits.sav
cp: cannot access fritus: No such file or directory
```

Here the filename *fruits* has been misspelled *fritus*, resulting in an error. In this example *dest* was the file *fruits.sav*; the same error would have been generated if *dest* was a directory.

The third error type is illustrated by the following command:

```
$ cp fruits users fruits.sav
usage: cp [-R [-H | -L | -P]] [-f | -i] [-p] src target
        cp [-R [-H | -L | -P]] [-f | -i] [-p] src1 ... srcN directory
```

Because *dest*, in this case *fruits.sav*, is not a directory, a usage statement that highlights the proper syntax for a *cp* command is presented. The output might be different on your system because some versions of *cp* do not display the usage information.

If the file *fruits.sav* does not exist, the error message is

```
cp: fruits.sav: No such file or directory
```

This illustrates the fourth error type.

The fifth error type is illustrated by the following command:

```
$ cp fruits Downloads/ users Documents/
cp: Downloads is a directory (not copied).
```

Although *cp* reports an error for the directory *Downloads*, the other files are correctly copied to the directory *Documents*.

Renaming Files (*mv*)

The *mv* command (short for move) can be used to change the name of a file. The basic syntax is

```
mv src dest
```

Here *src* is the original name of the file and *dest* is the new name of the file. For example, the command

```
$ mv fruits fruits.sav
```

changes the name of the file `fruits` to `fruits.sav`. There is no output from `mv` if the name change is successful.

If `src` does not exist, an error will be generated. For example,

```
$ mv cp fritus fruits.sav
mv: fritus: cannot access: No such file or directory
```

Similar to `cp`, `mv` does not report an error if `dest` already exists. The old file is automatically overwritten. This problem can be avoided by specifying the `-i` option (short for interactive). In interactive mode, `mv` prompts for confirmation before overwriting any files.

Assuming that the file `fruits.sav` already exists, the command

```
$ mv -i fruits fruits.sav
```

results in a confirmation prompt similar to the following:

```
overwrite fruits.sav?
```

If `y` (yes) is chosen, the file `fruits.sav` is overwritten; otherwise the file is untouched. The actual prompt varies among the different versions of UNIX.

Removing Files (`rm`)

The `rm` command (short for remove) can be used to remove or delete files. Its syntax is

```
rm file1 ... fileN
```

Here `file1 ... fileN` is a list of one or more files to remove. For example, the command

```
$ rm fruits users
```

removes the files `fruits` and `users`.

Because there is no way to recover files that have been removed using `rm`, you should make sure that you specify only those files you really want removed. One way to ensure this is by specifying the `-i` option (short for interactive). In interactive mode, `rm` prompts before removing every file. For example, the command

```
$ rm -i fruits users
```

produces confirmation prompts similar to the following:

```
fruits: ? (n/y) y
users: ? (n/y) n
```

In this case, you answered `y` (yes) to removing `fruits` and `n` (no) to removing `users`. Thus, the file `fruits` was removed, but the file `users` was untouched.

Common Errors

The two most common errors when using `rm` are

- One of the specified files does not exist.
- One of the specified files is a directory.

The first error type is illustrated by the following command:

```
$ rm users fritus hosts
rm: fritus non-existent
```

Because the file `fruits` is misspelled as `fritus`, it cannot be removed. The other two files are removed correctly.

The second error type is illustrated by the following command:

```
$ rm fruits users Documents/
rm: Documents directory
```

The `rm` command is unable to remove directories and presents an error message stating this fact. It removes the two other files correctly.

Summary

In this chapter, the following topics were discussed:

- Listing files using `ls`
- Viewing the content of a file using `cat`
- Counting the words, lines, and characters in a file using `wc`
- Copying files using `cp`
- Renaming files using `mv`
- Removing files using `rm`

Knowing how to perform each of these basic tasks is essential to becoming a good shell programmer. In the chapters ahead, you will use these basics to create scripts for solving real-world problems.

Questions

1. What are invisible files? How can they be listed with `ls`?
2. Is there any difference in the output of the following commands?
 - a. `$ ls -a1`
 - b. `$ ls -1 -a`
 - c. `$ ls -1a`
3. Which options should be specified to `wc` to count just the number of lines and characters in a file?
4. Given that `hw1`, `hw2`, `ch1`, and `ch2` are files and `book` and `homework` are directories, which of the following commands generates an error message?
 - a. `$ cp hw1 ch2 homework`
 - b. `$ cp hw1 homework hw2 book`
 - c. `$ rm hw1 homework ch1`
 - d. `$ rm hw2 ch2`

Terms

Directories Directories are used to hold ordinary and special files. Directories are similar to folders in Mac OS or Windows.

Invisible Files An invisible file is one whose first character is a dot or period (.). Many programs (including the shell) use such files to store configuration information. Invisible files are also referred to as hidden files.

Option An option is an argument that starts with the hyphen or dash character, '-'.

Ordinary File An ordinary file is a file that contains data, text, or program instructions. Almost all the files on a UNIX system are ordinary files.

Special Files Special files are mainly used to provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Some special files are similar to aliases or shortcuts and enable you to access a single file using different names.

This page intentionally left blank

HOUR 4



Working with Directories

UNIX uses a hierarchical structure for organizing files and directories, which is referred to as a *directory tree*. The tree has a single root node, slash (/); all other directories are contained below it.

Every directory, including /, can store both files and other directories. Every file is stored in a directory, and every directory, except /, is stored within a directory.

This is slightly different from the multi-root hierarchical structure used by Windows and Mac OS. In those operating systems, all devices (floppy disk drives, CD-ROMs, hard drives, and so on) are located at the same top-most level. The UNIX model is slightly different, but after you've adjusted to it, it is extremely convenient.

This chapter introduces the directory tree and shows you how to manipulate its building blocks: directories. Specifically, the topics we will cover are

- The Directory tree
- Switching directories
- Listing files and directories
- Manipulating directories

The Directory Tree

To understand the origin and advantages of the directory tree, let's consider a project that requires organization: writing a book. When you start out, it is easiest to put all the documents related to the book in one location. As you work on the book, you might find it hard to locate the material related to a particular chapter.

If you are writing the book with pen and paper, the easiest solution to this problem is to take all the pages related to the first chapter and put them into a folder labeled Chapter 1. As you write more chapters, you can put the material related to these chapters into separate folders. If you stick to this method, you will have many separate folders by the time you finish the book. You might put all the folders into a box and label that box with the name of the book. (Then you can stack the boxes in your closet.)

By grouping the material for the different chapters into folders and grouping the folders into boxes, the multitude of pages required to write a book becomes organized and easily accessible. When you want to see Chapter 5 from a particular book, you can grab that box from your closet and look at the folder pertaining to Chapter 5.

You can use this same method for projects on a computer. When you start out, all the files for the book might be in your home directory, but as you write more chapters, you can create directories to store the material relating to a particular chapter. Finally, you can group all of those directories into a directory named after the book.

As you can probably see, this arrangement creates an upside-down *tree* with a root at the top and directories *branching* off from the root. The files stored in the directories can be thought of as *leaves*.

This brings up the notion of *parent* directories and *child* or *subdirectories*. For example, consider two directories A and B, where directory A contains directory B. In this case, A is called the parent of B, and B is called a child of A.

The only limitation on the depth of the directory tree is that the *absolute path* to a file cannot have more than 1,024 characters. Absolute paths are covered later in the chapter.

Filenames

Every file and directory has a name associated with it. This name is referred to as that file or directory's *filename*. Every file and directory is also associated with the name of its parent. When a filename is combined with the parent directory's name, the result is called a *pathname*. Two examples of pathnames are

```
/home/ranga/docs/book/ch5.doc  
/usr/local/bin/
```

As you can see, a pathname consists of several words separated by slashes, /. The individual words in a pathname are the names of files or directories. Taken together, the words and the slashes make up the pathname. The last word in a pathname is the actual name of the file or directory being referenced. The rest of the words are the names of its parent directories. In the first pathname of the previous example, the filename is `ch5.doc`.

Strictly speaking, a filename can be up to 255 characters long and can contain any ASCII character except /. In general, the characters used in pathnames are the alphanumeric characters (a to z, A to Z, and 0 to 9) along with periods (.), hyphens (-), and underscores (_). Other characters, such as space, tab, and the shell's special characters (!, #, \$, %, &, *, (,), |, \, ", ', ?, {, }, [,], `, <, >, ; and :), are usually avoided because many programs cannot deal with them properly. For example, consider a file with the following name:

```
A Farewell To Arms
```

Most programs will treat this as four separate files named `A`, `Farewell`, `To`, and `Arms`. A workaround for this problem is covered in Chapter 10, "Quoting."

One thing to keep in mind about filenames is that two files in the same directory cannot have the same name. Both of the following pathnames refer to the same file:

```
/home/ranga/docs/ch5.doc  
/home/ranga/docs/ch5.doc
```

whereas the following pathnames refer to the different files:

```
/home/ranga/docs/ch5.doc  
/home/ranga/docs/books/ch5.doc
```

Filenames are also case sensitive: You can have two files in the same directory whose names differ only by case. For example, the following pathnames refer to different files:

```
/home/ranga/docs/ch5.doc  
/home/ranga/docs/CH5.doc
```

Pathnames

In order to access a file or directory, its pathname must be specified. As you have seen, a pathname consists of two parts: the name of the directory and the names of its parents. UNIX offers two ways to specify the names of the parent directory, leading to two types of pathnames:

- Absolute
- Relative

An Analogy for Pathnames

The following statements illustrate the difference between absolute and relative pathnames:

"I live in San Jose."

"I live in San Jose, California, USA."

The first statement gives only the city in which I live. It does not give any more information, thus it is a relative location. It could be located in any state or country containing a city called San Jose. The second statement fully qualifies the location, thus it is an absolute location.

Absolute Pathnames

An *absolute* pathname represents the location of a file or directory starting from the root directory and listing all the directories between the root and the file or directory of interest. Because absolute pathnames list the path from the root directory, they always start with the slash (/) character. Regardless of what the current directory is, an absolute path points to an exact location of a file or directory. The following is an example of an absolute pathname:

```
/home/ranga/work/bugs.txt
```

This absolute path tells you that the file `bugs.txt` is located in the directory `work`, which is located in the directory `ranga`, which in turn is located in the directory `home`. The slash at the beginning of the path tells you that the directory `home` is located in the root directory.

Relative Pathnames

A *relative* pathname enables you to access files and directories by specifying a path to that file or directory within your current directory. When your current directory changes, the relative pathname to a file can also change.

To find out what the current directory is, use the `pwd` command (short for **print working directory**), which prints the name of the directory in which you are currently located. For example,

```
$ pwd
/home/ranga/pub
```

tells us that the current directory is `/home/ranga/pub`.

When you're specifying a relative pathname, the slash character is not present at the beginning of the pathname. The relative pathname is a list of the directories located between your current directory and the file or directory you are representing.

If you are pointing to a directory in your pathname that is below your current one, you can access it by specifying its name. For example, the directory name `docs/` refers to the directory `docs` located in the current directory.

In order to access the current directory's parent directory or other directories at a higher level in the tree than the current level, use the special name of two dots (`..`). The UNIX filesystem uses two dots (`..`) to represent the directory above you in the tree, and a single dot (`.`) to represent your current directory.

Let's look at an example that illustrates how relative pathnames are used. Assuming that the current directory is

```
/home/ranga/work
```

the relative pathname

```
../docs/ch5.doc
```

represents the file

```
/home/ranga/docs/ch5.doc
```

whereas

```
./docs/ch5.doc
```

represents the file

```
/home/ranga/work/docs/ch5.doc
```

You can also refer to this file using the following relative path:

```
docs/ch5.doc
```

You do not have to append `/` to the beginning of pathnames referring to files or directories located within the current directory or its subdirectories.

Switching Directories

Now that we have covered the basics of the directory tree, let's look at moving around the tree using the `cd` (short for *change directory*) command.

Home Directories

First print the working directory:

```
$ pwd  
/home/ranga
```

The preceding example should indicate to you that I am in my home directory. Your home directory is the initial directory where you start when you log in to a UNIX machine. The easiest way to determine the location of your home directory is to do the following:

```
$ cd
$ pwd
/home/ranga
```

When you issue the `cd` command without arguments, it changes the current directory to your home directory. After the `cd` command completes, the `pwd` command prints the working directory, which happens to be your home directory.

Changing Directories

You can use the `cd` command to do more than change to a home directory; it can be used to change to any directory by specifying a valid absolute or relative path. The syntax is as follows:

```
cd dir
```

Here *dir* is the name of the directory that you want to change to. For example, the command

```
$ cd /usr/local/bin
```

changes to the directory `/usr/local/bin`. Regardless of the directory we were in before, this command always places us in the directory `/usr/local/bin`. That is the advantage of using an absolute path. Let's look at another example. Say that the current directory is

```
$ pwd
/home/ranga
```

From this directory, we can `cd` to the directory `/usr/local/bin` using the following relative path:

```
$ cd ../../usr/local/bin
```

Changing the current directory means that all your relative path specifications must be relative to the new directory rather than the old one. For example, consider the following sequence of commands:

```
$ pwd
/home/ranga/docs
$ cat names
ranga
vathsa
amma
$ cd /usr/local
```

```
$ cat names
cat: cannot open names
```

When the first `cat` command was issued, the working directory was `/home/ranga/docs`. The file, `names`, was located in this directory, thus the `cat` command found it and displayed its contents.

After the `cd` command, the working directory became `/usr/local`. Because there was no file called `names` in that directory, `cat` produced an error message stating that it could not open the file. To access the file `names` from the new directory, you need to specify either the absolute path to the file or a relative path from the current directory.

Common Errors

The most common errors with `cd` are

- Specifying more than one argument
- Trying to `cd` to a file
- Trying to `cd` to a directory that does not exist

An example of specifying more than one argument is seen here:

```
$ cd /home /tmp /var
$ pwd
/home
```

As you can see, `cd` uses only its first argument. The other arguments are ignored. Sometimes, in shell programming, this becomes an issue. When you issue a `cd` command in a shell script, you need make sure that you end up in the correct directory.

Let's now take a look at trying to `cd` to a file. An example of this is as follows:

```
$ pwd
/home/ranga
$ cd docs/ch5.doc
cd: docs/ch5.doc: Not a directory
$ pwd
/home/ranga
```

Here, we tried to change to a location that was not a directory, and `cd` reported an error. If this error occurs, the working directory does not change. The output from `pwd` illustrates this.

Finally, let's try to `cd` to a directory that does not exist:

```
$ pwd
/home/ranga
$ cd final_exam_answers
cd: final_exam_answers: No such file or directory
```

```
$ pwd
/home/ranga
```

Here, we tried to change into the directory `final_exam_answers`, a directory that did not exist, thus `cd` reported an error. The final `pwd` command shows that the working directory did not change.

Listing Files and Directories

In Chapter 3, “Working with Files,” you looked at using the `ls` command to list the files in the current directory. Now let’s look at using the `ls` command to list the files in any directory.

Listing Directories

To list the files in a directory, you can use the following syntax:

```
ls dir
```

Here, *dir* is the absolute or relative pathname of the directory whose contents you want listed.

For example, both of the following commands will list the contents of the directory `/usr/local` (assuming that the working directory is `/home/ranga`):

```
$ ls /usr/local
$ ls ../../usr/local
```

On my system, the listing resembles

```
X11          bin          gimp          jikes         sbin
ace          doc          include       lib           share
atalk       etc          info          man           turboj-1.1.0
```

The listing on your system might look quite different.

You can use any of the options you covered in Chapter 3 to change the output. For example, the command

```
$ ls -aF /usr/local
```

produces the output

```
./          atalk/      gimp/       lib/         turboj-1.1.0/
../         bin/        include/    man/
X11/       doc/        info/       sbin/
ace/       etc/        jikes/     share/
```

You can also specify more than one directory as an argument. For example,

```
$ ls /home /usr/local
```

produces the following output on my system:

```
/home:
amma  ftp      httpd   ranga   vathsa

/usr/local:
X11          bin          gimp        jikes      sbin
ace          doc          include     lib        share
atalk       etc          info        man        turboj-1.1.0
```

A blank line separates the contents of each directory.

Listing Files

You can mix files and directories as arguments to `ls`:

```
$ ls .profile docs/ /usr/local /bin/sh
```

This produces a listing of the specified files and the contents of the directories. If you don't want the contents of a directory listed, you need to specify the `-d` option to `ls`.

This forces `ls` to display only the name of the directory, not its contents:

```
$ ls -d /home/ranga
/home/ranga
```

The `-d` option can be combined with any of the other `ls` options. An example of this is

```
$ ls -aFd /usr/local /home/ranga /bin/sh
/bin/sh*      /home/ranga/ /usr/local/
```

Common Errors

If the file or directory you specify does not exist, `ls` reports an error. For example,

```
$ ls tomorrows_stock_prices.txt
tomorrows_stock_prices.txt: No such file or directory
```

If you specify several arguments instead of one, `ls` will report errors only for those files or directories that do not exist. It correctly lists the others. For example,

```
$ ls tomorrows_stock_prices.txt /usr/local .profile
```

produces an error message

```
tomorrows_stock_prices.txt: No such file or directory
/usr/local:
X11          bin          gimp        jikes      sbin
ace          doc          include     lib        share
atalk       etc          info        man        turboj-1.1.0

.profile
```

Manipulating Directories

The most common ways of manipulating a directory are

- Creating a directory
- Copying a directory
- Moving a directory
- Removing a directory

Creating Directories

You can create directories with the `mkdir` command (short for make directory). Its syntax is

```
mkdir dir
```

Here, *dir* is the absolute or relative pathname of the directory you want to create. For example, the command

```
$ mkdir hw1
```

creates the directory `hw1` in the current directory. Here is another example:

```
$ mkdir /tmp/test-dir
```

This command creates the directory `test-dir` in the `/tmp` directory. The `mkdir` command produces no output if it successfully creates the requested directory.

If more than one directory is specified, `mkdir` will try to create each of the directories. For example,

```
$ mkdir docs pub
```

creates the directories `docs` and `pub` under the current directory.

Creating Parent Directories

Sometimes when you want to create a directory, one or more of its parent directories might not exist. If this is the case, `mkdir` issues an error message. For example,

```
$ mkdir /tmp/ch4/tst
mkdir: Failed to make directory "/tmp/ch4/tst"; No such file or directory
```

In order to create the parent directories, you can specify the `-p` (p as in parent) option to `mkdir`. For example,

```
$ mkdir -p /tmp/ch4/tst
```

will create all the required parent directories. In order to create this directory, `mkdir` will use the following procedure:

1. `mkdir` checks whether the directory `/tmp` exists. If it does not exist, `mkdir` creates it.
2. `mkdir` checks whether the directory `/tmp/ch04` exists. If it does not exist, `mkdir` creates it.
3. `mkdir` checks whether the directory `/tmp/ch04/test1` exists. If it does not, `mkdir` creates it.

Common Errors

The most common error in using `mkdir` is trying to create a directory that already exists. If the directory `/tmp/ch04` already exists, the command

```
$ mkdir /tmp/ch04
```

generates an error message similar to the following:

```
mkdir: cannot make directory '/tmp/ch04': File exists
```

An error also occurs if you try to create a directory with the same name as a file. For example, the following commands

```
$ ls -F docs/names.txt
names
$ mkdir docs/names
```

result in the error message

```
mkdir: cannot make directory 'docs/names': File exists
```

If you specify more than one argument to `mkdir`, it creates as many of these directories as it can. An error message is generated for each directory that could not be created.

Copying Files and Directories

In Chapter 3, you looked at using the `cp` command to copy files. Now let's look at using it to copy directories.

To copy a directory, you need to specify the `-r` option to `cp`. The syntax is as follows:

```
cp -r src dest
```

Here, *src* is the pathname of the directory you want to copy, and *dest* is the pathname where you want the copy to be placed. When the `-r` option is specified, all files and directories located under *src* are copied to *dest*. For example,

```
$ cp -r docs/book /mnt/zip
```


copies the directory `book` located in the `docs` directory to the directory `/mnt/zip`. If the directory `book` does not exist under `/mnt/book`, it will be created.

Copying Multiple Directories

You can copy multiple directories in much the same way as copying multiple files. If `cp` encounters more than one source, all the source directories are copied to the destination directory. The destination directory is assumed to be the last argument. For example, the command

```
$ cp -r docs/book docs/school work/src /mnt/zip
```

copies the directories `school` and `book`, located in the directory `docs`, to `/mnt/zip`. It also copies the directory `src`, located in the directory `work`, to `/mnt/zip`. After the copies finish, `/mnt/zip` resembles the following:

```
$ ls -aF /mnt/zip
./ ../ book/ school/ src/
```

You can also mix files and directories in the argument list. For example,

```
$ cp -r .profile docs/book .kshrc doc/names work/src /mnt/jaz
```

copies all the requested files and directories to the directory `/mnt/jaz`.

If the argument list consists of only files, the `-r` option has no effect.

Common Errors

The most common problem related to copying directories is using a destination that is not a directory. An example of this is

```
$ cp -r docs /mnt/zip/backup
cp: cannot create directory '/mnt/zip/backup': File exists
$ ls -F /mnt/zip/backup
/mnt/zip/backup
```

As you can see, the `cp` operation fails because a file called `/mnt/zip/backup` already exists.

Moving Files and Directories

In the previous chapter we looked at using `mv` to rename files, but its real purpose is to move files and directories between different locations in the directory tree. The basic syntax is:

```
mv src dest
```

Here, *src* is the name of the file or directory you want to move, and *dest* is the directory where you want the file or directory to end up. For example,

```
$ mv /home/ranga/names /tmp
```

moves the file *names* located in the directory */home/ranga* to the directory */tmp*.

Moving a directory is exactly the same:

```
$ mv docs/ work/
```

moves the directory *docs* into the directory *work*. To move the directory *docs* back to the current directory, you can use the command:

```
$ mv work/docs .
```

One nice feature of *mv* is that you can move and rename a file or directory all in one command. For example,

```
$ mv docs/names /tmp/names.txt
```

moves the file *names* in the directory *docs* to the directory */tmp* and renames it *names.txt*.

Moving Multiple Items

Just as you can with *cp*, you can specify more than one file or directory as the source. For example,

```
$ mv work/ docs/ .profile pub/
```

moves the directories *work* and *docs* along with the file *.profile* into the directory *pub*.

When you are moving multiple items, you cannot rename them. If you want to rename an item and move it, you must use a separate *mv* command for each item.

Common Errors

Two common errors that can occur when using *mv* are

- Moving multiple files and directories to a directory that does not exist
- Moving files and directories to a file

These cases produce the same error message, so look at one example that illustrates what happens:

```
$ mv .profile docs pub /mnt/jaz/backup
mv: when moving multiple files, last argument must be a directory
$ ls -aF /mnt/jaz
./          ../          archive/    lost+found/ old/
```

As you can see, no directory named `backup` exists in the `/mnt/jaz` directory, so `mv` reports an error. The same error is reported if `backup` happens to be a file.

Removing Directories

Two commands can be used to remove directories:

- `rmdir`
- `rm -r`

The first command, `rmdir` (short for remove directory), can only be used to remove empty directories. It is considered “safe” because in the worst case, you just lose an empty directory that can be easily recreated with `mkdir`.

The second command, `rm -r`, removes a directory and all of its contents. It is considered “unsafe” because it is possible to accidentally delete an entire system.



When using `rm` to remove either files or directories, make sure that you remove only those files that you don't want.

There is no way to restore files deleted with `rm`, so mistakes can be very hard to recover from.

rmdir

The syntax for `rmdir` is

```
rmdir dir1 ... dirN
```

Here, `dir1 ... dirN` are the directories you want removed. At least one directory must be specified. For example, the command

```
$ rmdir ch01 ch02 ch03
```

removes the directories `ch01`, `ch02`, and `ch03` if they are empty. The `rmdir` command produces no output if it is successful.

Common Errors

Common errors that might occur when using `rmdir` include

- Trying to remove a directory that is not empty
- Trying to remove files with `rmdir`

For the first case, you need to know how to determine whether a directory is empty. You can do this by using the `-A` option of the `ls` command. An empty directory produces no output. If there is some output, the directory you specified is not empty.

For example, if the directory `bar` is empty, the following command

```
$ ls -A bar
```

returns nothing. This directory can be removed with `rmdir`.

Now say that the directory `docs` is not empty. The following command

```
$ rmdir docs
```

produces an error message

```
rmdir: docs: Directory not empty
```

To illustrate the second type of error, assume that `names` is a file. The following command

```
$ rmdir names
```

produces an error message

```
rmdir: names: Not a directory
```

rm -r

You can specify the `-r` option to `rm` to remove a directory and its contents. The syntax is as follows:

```
rm -r dir1 ... dirN
```

Here `dir1 ... dirN` are the names of the directories you want removed. For example, the command

```
$ rm -r ch01/
```

removes the directory `ch01` and its contents. This command produces no output.

You can specify a combination of files and directories as follows:

```
$ rm -r ch01/ test1.txt ch01-old.txt ch02/
```

In order to make `rm` safer, you can combine the `-r` and `-i` options.

Common Errors

The most common error that can occur when using `rm` is trying to remove a file or directory that does not exist. In this case, `rm` reports an error. For example, if the directory `midterm_answers` does not exist, trying to remove it will fail as follows:

```
$ rm -r midterm_answers
rm: midterm_answers: No such file or directory
```

Summary

In this chapter, we have looked at working with directories. Specifically, the following topics were covered:

- Working with filenames and pathnames
- Switching directories
- Listing files and directories
- Creating directories
- Copying and moving directories
- Removing directories

We reviewed each of these topics because it is important to know how to perform these functions when writing shell scripts. As you go further into this book, you will begin to see that directory manipulations occur quite frequently in shell scripts.

Questions

1. Which of the following are absolute pathnames? Which are relative?
 - a. `/usr/local/bin`
 - b. `../../home/ranga`
 - c. `docs/book/ch01`
 - d. `/`
2. What is the output of the `pwd` command after the following sequence of `cd` commands have been issued?

```
$ cd /usr/local
$ cd bin
$ cd ../../tmp
$ cd
```
3. What command should be used to copy the directory `/usr/local` to `/opt/pgms`?
4. What command(s) should be used to move the directory `/usr/local` to `/opt/pgms`?
5. Given the following listing for the directory `backup`, can the `rmdir` command be used to remove this directory? If not, please give a command that can be used.

```
$ ls -a backup
./    ../    sysbak-980322  sysbak-980112
```

Terms

Absolute Pathname The absolute pathname represents the location of a file or directory starting from / and listing all the directories between / and the file or directory of interest. The pathname `/etc/hosts` is an absolute pathname.

Directory Tree The hierarchical structure used in UNIX for organizing files and directories.

Filename The name of a file. The name of the file `/etc/hosts` is `hosts`.

Parent Directory The directory that contains a given directory. If directory B is contained within directory A, directory A is considered the parent directory of B.

Pathname The filename of a file combined with the filenames of its parent directories. The pathname of the file `hosts` located in the directory `/etc` is `/etc/hosts`.

Relative Pathname The relative pathname represents the location of a file or directory relative to the current directory. The pathname `../etc/hosts` is a relative pathname.

Root The root directory, `/`, is the top-most directory in the UNIX directory tree.

Subdirectory A directory that is contained within another directory. If directory A contains directory B, directory B is considered a subdirectory of A.

This page intentionally left blank

HOUR 5



Input and Output

Until now, you have been looking at commands that output messages. In this chapter, you will look at the different types of output available to shell scripts. You will also discover the mechanisms used to obtain *input* from users. Specifically, the areas that you will cover are

- Output to the screen
- Output to a file
- Input from a file
- Input from users

Output

As you have seen in previous chapters, most commands produce output. For example, the command

```
$ date
```

produces the current date in the terminal window:

```
Thu Nov 12 16:32:35 PST 2001
```


When a command produces output that is written to the terminal, you say that the program has printed its output to the *Standard Output*, or `STDOUT`. When you run the `date` command, it prints the date to `STDOUT`. You have also seen commands produce error messages, such as:

```
$ ln -s ch01.doc ch01-01.doc
ln: cannot create ch01-1.doc: File Exists
```

Error messages are not written to `STDOUT`, but instead they are written to a special type of output called *Standard Error* or `STDERR`, which is reserved for error messages. Most commands use `STDERR` for error messages and `STDOUT` for informational messages. You will look at `STDERR` later in this chapter. In this section, you will look at how shell scripts can use `STDOUT` to output messages to each of the following:

- The terminal (`STDOUT`)
- A file
- The terminal and a file

Output to the Terminal

Two common commands that can be used to output messages to `STDOUT` are `echo` and `printf`. The `echo` command is mostly used for printing strings that require simple formatting. The `printf` command is the shell version of the C language function `printf`. It provides a high degree of flexibility in formatting output.

echo

The syntax for `echo` is as follows

```
echo str
```

Here *str* is the message you want printed. For example, the command

```
$ echo Hi
```

produces the following output:

```
Hi
```

You can also embed spaces in the output as follows:

```
$ echo Safeway has fresh fruit
Safeway has fresh fruit
```

In addition to spaces, you can embed punctuation marks and formatting escape sequences in the *str*.

Embedding Punctuation Marks

Punctuation marks are used when you need to ask the user a question, complete a sentence, or issue a warning. For example, the following command might be used as the prompt in an install script:

```
echo Do you want to install?
```

Usually, significant error messages are terminated with the exclamation point. For example, the following command

```
echo ERROR: Could not find required libraries! Exiting.
```

might be found in a script that configures a program for execution. You can also use any combination of the punctuation marks. For example, the following command uses the comma (,), question mark (?), and exclamation point(!) punctuation marks:

```
$ echo Eliza, where the devil are my slippers?!?  
Eliza, where the devil are my slippers?!?
```

Formatting with Escape Sequences

The output in the previous examples consisted of single lines with words separated by spaces. Frequently, output needs to be formatted into columns or multiple lines. By using *escape sequences*, you can format the output of `echo`. An escape sequence is a special sequence of characters that represents another character. When the shell encounters an escape sequence, it substitutes the escape sequence with a different character. The `echo` command understands several formatting escape sequences, the most common of which are given in Table 5.1.

TABLE 5.1 Escape Sequences for the `echo` Command

<i>Escape Sequence</i>	<i>Description</i>
<code>\n</code>	Prints a newline character
<code>\t</code>	Prints a tab character
<code>\c</code>	Prints a string without a default trailing newline



The escape sequences for the `echo` command, given in Table 5.1, do not work with all shells. These escape sequences work in the Bourne Shell (`/bin/sh` or `/sbin/sh` on Solaris) and Korn Shell (`ksh`). They do not work with `bash` or `zsh`. The `printf` command, covered later in this chapter, can be used as a work-around for this limitation in `bash` and `zsh`.

The `\n` escape sequence is normally used when you need to generate more than one line of output. For example, the command

```
$ FRUIT_BASKET="apple orange pear"
$ echo "Your fruit basket contains:\n$FRUIT_BASKET"
Your fruit basket contains:
apple orange pear
```

generates a list of fruit preceded by a description of the list. This example illustrates two important aspects of using escape sequences:

- The entire input string, *str*, is quoted.
- The escape sequence appears in the middle of the string, *str*, and is not separated by spaces.

Whenever an escape sequence is used in the input string to `echo`, the string must be quoted to prevent the shell from expanding the escape sequence on the command line. Quoting is explained in detail in Chapter 10, “Quoting”. Furthermore, the input string is a specification of how the output should look; spaces should not be used to separate the escape sequences unless that is how the output needs to be formatted.

It is possible to rewrite any `echo` command that uses the `\n` escape sequence as several `echo` commands. For example, you can generate the same output as in the previous example using two `echo` commands:

```
$ echo "Your fruit basket contains:"
$ echo $FRUIT_BASKET
```

Another commonly used escape sequence is the `\t` sequence, which generates a tab in the output. Usually it is used when you need to make a small table or generate tabular output that is only a few lines long. As an example, the following command generates a small table of two users along with their usernames:

```
$ echo "Name \tUser Name\nSriranga\tranga\nSrivathsa\tvathsa"
Name    User Name
Sriranga    ranga
Srivathsa   vathsa
```

As you can see from the output, the heading `User Name` is not centered over its column. You can fix this by adding another tab:

```
$ echo "Name\t\tUser Name\nSriranga\tranga\nSrivathsa\tvathsa"
Name      User Name
Sriranga    ranga
Srivathsa   vathsa
```

For generating large tables, the `printf` command, covered in the next section, is preferred because it provides a greater degree of control over the size of each column in the table.

The `\c` sequence is frequently used in shell scripts that need to generate user prompts or diagnostic output. As you have seen in the previous example, the default behavior of `echo` is to add a newline at the end of its output. When you are generating a prompt, this is not the most user-friendly behavior. When the `\c` escape sequence is used, `echo` does not output a newline when it finishes printing its input string. The following example illustrates the use of this option:

```
$ echo "Do you want to play a game? (y/n) \c"
```

It produces the following message:

```
Do you want to play a game (y/n)?
```

Some versions of `echo` do not understand the `\c` escape sequence. These versions of `echo` treat `\c` literally and the resulting output will look like the following:

```
$ echo "Please enter your name \c"
echo Please enter your name \c
$
```

In these versions of `echo`, you need to use the `-n` option instead of `\c`. In Chapter 23, “Scripting for Portability,” you will develop a mechanism that handles this difference automatically.

printf

The `printf` command is similar to the `echo` command, in that it enables you to print messages to `STDOUT`. In its most basic form, its usage is identical to `echo`. For example, the following `echo` command:

```
$ echo "Is that a mango?"
```

is identical to the `printf` command:

```
$ printf "Is that a mango?\n"
```

The only major difference is that the string specified to `printf` explicitly requires the `\n` escape sequence at the end of a string, in order for a newline to print. The `echo` command prints the newline automatically.



The `printf` command is located in the directory `/usr/bin` on Linux, Solaris, MacOS X, and HP-UX machines. The `printf` command is a built-in command in `bash`.

The power of `printf` comes from its capability to perform complicated formatting by using format specifications. The basic syntax for this is as follows:

```
printf format arguments
```

Here, *format* is a string that contains one or more of the formatting sequences, and *arguments* are strings that correspond to the formatting sequences specified in *format*. For those who are familiar with the C language `printf` function, the formatting sequences supported by the `printf` command are identical. The formatting sequences have the form:

```
%[ - ]m.nx
```

Here % starts the formatting sequence and *x* identifies the formatting sequences type. Table 5.2 gives possible values of *x*.

TABLE 5.2 Formatting Sequence Types

<i>Letter</i>	<i>Description</i>
s	String
c	Character
d	Decimal (integer) number
x	Hexadecimal number
o	Octal number
e	Exponential floating-point number
f	Fixed floating-point number
g	Compact floating-point number

Depending on the value of *x*, the integers *m* and *n* are interpreted differently. Usually *m* is the minimum length of a field, and *n* is the maximum length of a field. If you specify a real number format, *n* is treated as the precision that should be used. The hyphen (-) left justifies a field. By default, all fields are right justified.

The following commands illustrate the use of `printf`:

```
printf "%16s\t%16s\n" "Name" "User Name"
printf "%16s\t%16s\n" "Sriranga" "ranga"
printf "%16s\t%16s\n" "Srivathsa" "vathsa"
```

The format `%16s\t%16s\n` specifies that the output string should be separated in two columns, each 16 characters long and separated by a space. The output of these commands will be similar to the following:

Name	User Name
Sriranga	ranga
Srivathsa	vathsa

As you can see, the headings and the columns are not aligned properly. You can fix this by adding a `-` to the format specification:

```
printf "%-16s\t%-16s\n" "Name" "User Name"
printf "%-16s\t%-16s\n" "Sriranga" "ranga"
printf "%-16s\t%-16s\n" "Srivathsa" "vathsa"
```

The output of these commands will be similar to the following:

Name	User Name
Sriranga	ranga
Srivathsa	vathsa

To format numbers, specify a number formatting sequence, such as `%f`, `%e`, or `%g`, instead of the string formatting sequence, `%s`. One of the questions at the end of this chapter familiarizes you with using number formats.

Output Redirection

In the process of developing a shell script, you often need to capture the output of a command and store it in a file. When the output is in a file, it can be easily edited and modified. The process of capturing the output of a command and storing it in a file is called *output redirection* because it redirects the output of a command into a file instead of the screen. To redirect the output of a command or a script to a file, instead of `STDOUT`, use the output redirection operator, `>`, as follows:

```
cmd > file
list > file
```

The first form redirects the output of the command *cmd* to the file specified by *file*, whereas the second redirects the output of list *list* to the file specified *file*. If *file* exists, its contents are overwritten; if *file* does not exist, it is created. For example, the command

```
date > now
```

redirects the output of the `date` command into the file `now`. The output does not appear on the terminal, but it is placed into the file instead. If you view the file `now`, you find the output of the `date` command:

```
$ cat now
Sat Nov 14 11:14:01 PST 1998
```

You can also redirect the output of lists as follows:

```
{ date; uptime; who ; } > mylog
```

Here the output of the commands `date`, `uptime`, and `who` is redirected into the file `mylog`.



When you redirect output to a file using the output redirection operator, the shell overwrites the data in that file with the output of the command you specified. For example, the command

```
$ date > now
```

overwrites all the data in the file `now` with the output of the `date` command. For this reason, you should take extra care and make sure the file you specified does not contain important information.

Appending to a File

Overwriting a file simply by redirecting output to it is often undesirable. Fortunately, the shell provides a second form of output redirection with the `>>` operator, which appends output to a file. The basic syntax is as follows:

```
cmd >> file
list >> file
```

In these forms, output is appended to the end of `file`. If the file does not exist it is created. For example, you can prevent the loss of data from the file `mylog` each time a date is added by using the following command:

```
{ date; uptime; who ; } >> mylog
```

If you view the contents of `mylog`, you find that it contains the output of both lists:

```
11:15am up 79 days, 14:48, 5 users, load average: 0.00, 0.00, 0.00
ranga tty1 Aug 26 14:12
ranga tty2 Aug 26 14:13 (:0.0)
ranga tty0 Oct 27 19:42 (:0.0)
amma tty3 Oct 30 08:20 (localhost)
ranga tty4 Nov 14 11:13 (rishi.bosland.u)
Sat Nov 14 11:15:54 PST 1998
 11:16am up 79 days, 14:48, 5 users, load average: 0.00, 0.00, 0.00
ranga tty1 Aug 26 14:12
ranga tty2 Aug 26 14:13 (:0.0)
ranga tty0 Oct 27 19:42 (:0.0)
amma tty3 Oct 30 08:20 (localhost)
ranga tty4 Nov 14 11:13 (rishi.bosland.u)
```

Redirecting Output to a File and the Screen

In certain instances, you need to direct the output of a script to a file and onto the terminal. An example of this is shell scripts that are required to produce a log file of their activities. For interactive scripts, the log file cannot just contain the script's output redirected to a file.

To redirect output to a file and the screen, you can use the `tee` command. The basic syntax is as follows:

```
cmd | tee file
```

Here *cmd* is the name of a command, such as `ls`, and *file* is the name of the file where you want the output written. For example, the command

```
$ date | tee now
```

produces the following output on the terminal:

```
Sat Nov 14 19:50:16 PST 2001
```

The same output is written to the file `now`.

Input

Many UNIX programs are interactive and read input from the user. To use such programs in shell scripts, you need to provide them with input in a non-interactive manner. Also, scripts often need to ask the user for input in order to execute commands correctly.

To provide input to interactive programs or to read input from the user, you need to use input redirection. In this section, you will look at the following methods in detail:

- Input redirection from files
- Reading input from a user
- Redirecting the output of one command to the input of another

Input Redirection

When you need to use an interactive command, such as `mail` in a script, you need to provide the command with input. One method for doing this is to store the input of the command in a file and then tell the command to read input from that file. You can accomplish this using input redirection. The input can be redirected in a manner similar to output redirection. In general, input redirection is

```
cmd < file
```

Here the contents of *file* become the input for *cmd*. As an example, the following command is an excellent use of redirection:

```
Mail ranga@soda.berkeley.edu < Final_Exam_Answers
```

Here the input to the `Mail` command, which becomes the body of the mail message, is the file `Final_Exam_Answers`. In this particular example, a professor might perform this function, and the file might contain the answers to a current final exam.

Here Documents

An additional use of input redirection is in the creation of *here* documents. Say you need to send a list of phone numbers or URLs to the printer. You can enter the information that you want to send to the printer into the here document and then send that here document to the printer. This is much simpler than using a temporary file, which needs to be created and then should be deleted.

The general form for a here document is as follows:

```
cmd << delimiter
document
delimiter
```

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified *delimiter*. All the input lines up to the line containing the *delimiter* are then fed into the standard input of the *cmd*. The *delimiter* tells the shell that the here document has completed. Without it, the shell continues to read input forever. The *delimiter* must be a single word that does not contain spaces or tabs. For example, to print a quick list of URLs, you can use the following here document:

```
lpr << MYURLS
    http://www.csua.berkeley.edu/~ranga/
    http://www.cisco.com/
    http://www.marathon.org/story/
    http://www.gnu.org/
MYURLS
```

To strip the tabs in this example, you can give the << operator a - option.

You can also combine here documents with output redirection as follows:

```
cmd > file << delimiter
document
delimiter
```

If used in this form, the output of *cmd* is redirected to the specified *file*, and the input of *cmd* becomes the here document. For example, you can use the following command to create a file with the short list of URLs given previously:

```
cat > urls << MYURLS
    http://www.csua.berkeley.edu/~ranga/
    http://www.cisco.com/
    http://www.marathon.org/story/
    http://www.gnu.org/
MYURLS
```

Reading User Input

A common task in shell scripts is to prompt users for input and then read their responses. You can use the `read` command to read a user's response and store it in a variable. Variables are explained in detail in Chapter 8, "Variables." The syntax of the `read` command is as follows:

```
read name
```

It reads the entire line of user input until the user presses Enter and assigns the input string to the variable specified by *name*. The following example illustrates the use of `read`:

```
echo "What is your name? \c"  
read NAME
```

The user's response is stored in the variable `NAME`.



On some versions of `echo`, the `\c` in the previous example will appear literally in the output as follows:

```
What is your name? \c
```

If you experience this behavior, you should use the following `echo` command instead of the one given in the previous example:

```
echo -n "What is your name?"
```

Pipelines

Most commands in UNIX that are designed to work with files can also read input from `STDIN`. This enables you to use one program to filter the output of another. Using one program to manipulate the output of another program is one of the most common tasks in shell programming. This section provides a short description of this technique; Chapters 15, 16, and 17 contain much more detailed examples.

You can redirect the output of one command to the input of another command using a *pipeline*, which connects several commands together with *pipes* as follows:

```
cmd1 | cmd2 | ... | cmdN
```

The pipe character, `|`, connects the standard output of *cmd1* to the standard input of *cmd2*, and so on. The commands can be as simple or complex as are required. The following commands illustrates the use of pipelines:

```
tail -f /var/adm/messages | more  
ps -ael | grep "$UID" | more
```

In the first example, the standard output of the `tail` command is piped into the standard input of the `more` command, which enables the output to be viewed one screen at a time. In the second example, the standard output of `ps` is connected to the standard input of `grep`, and the standard output of `grep` is connected to the standard input of `more`, so that the output of `grep` can be viewed one screen at a time. The `tail` and `grep` commands are explained in detail in Chapter 15, “Text Filters.”



One important thing about pipelines is that each command is executed as a separate process, and the exit status of a pipeline is the exit status of the last command.

It is vital to remember this fact when writing scripts that must do error handling.

File Descriptors

When you issue any command, three files are opened and associated with that command. In the shell, each of these files is represented by a small integer called a file descriptor. A *file descriptor* is a mechanism by which you can associate a number with a filename and then use that number to read and write from the file. File descriptors are often referred to as *file handles*.

The three files opened for each command along with their corresponding file descriptors are

- Standard Input (STDIN), 0
- Standard Output (STDOUT), 1
- Standard Error (STDERR), 2

The integer following each of these files is its file descriptor. Usually, these files are associated with the user’s terminal, but they can be redirected into other files. In the previous examples in this chapter, you have used input and output redirection using the default file descriptors. This section introduces the general form of input and output redirection.

Associating Files with a File Descriptor

You can associate any file with file descriptors using the `exec` command. Associating a file with a file description is useful when you need to redirect output or input to a file many times but you don’t want to repeat the filename several times. To open a file for writing, use one of the following forms:

```
exec n>file
exec n>>file
```

Here *n* is an integer, and *file* is the name of the file to be opened for writing. The first form overwrites the specified *file* if it exists. The second form appends to the specified *file*. For example, the following command

```
$ exec 4>fd4.out
```

associates the file `fd4.out` with the file descriptor 4.



Use output redirection of STDOUT with care. If you accidentally redirect STDOUT, your commands may appear to stop working. For example, the following command:

```
$ exec 1>fd1.out
```

redirects STDOUT to the file `fd1.out`. If you execute this command, the output from all your commands will be placed in the file `fd1.out`. You will not see any output on your terminal.

To open a file for reading, you can use the following form:

```
exec n<file
```

Here *n* is an integer, and *file* is the name of the file to be opened for reading.

General Input/Output Redirection

You can perform general output redirection by combining a file descriptor and an output redirection operator. The general forms are

```
cmd n> file  
cmd n>> file
```

Here *cmd* is the name of a command, such as `ls`; *n* is a file descriptor (integer) and *file* is the name of the file. The first form redirects the output of *cmd* to the specified *file*, whereas the second form appends the output of *cmd* to the specified *file*. For example, you can write the standard output redirection in the general form as follows:

```
cmd 1> file  
cmd 1>> file
```

Here the 1 explicitly states that STDOUT is being redirected into the given file.

General input redirection is similar to general output redirection. It is performed as follows:

```
cmd n<file
```

Here *cmd* is the name of a command, such as `ls`; *n* is a file descriptor (integer) and *file* is the name of the file. For example, the standard input redirection can be written in the general form as follows:

```
cmd 0<file
```

Redirecting STDOUT and STDERR to Separate Files

One of the most common uses of file descriptors is to redirect STDOUT and STDERR to separate files. The basic syntax is

```
cmd 1> file1 2> file2
```

Here STDOUT of the command *cmd* is redirected to *file1*, and the STDERR (error messages) is redirected to *file2*. Often the STDOUT file descriptor, 1, is omitted, so a shorter form is

```
cmd > file1 2> file2
```

You can also use the append operator in place of either standard redirect operator:

```
cmd >> file1 2> file2
cmd > file1 2>> file2
cmd >> file1 2>> file2
```

The first form appends STDOUT to *file1* and redirects STDERR to *file2*. The second form redirects STDOUT to *file1* and appends STDERR to *file2*. The third form appends STDOUT to *file1* and appends STDERR to *file2*.

The following example illustrates the first form:

```
ln -s ch05.doc ./docs >> /tmp/ln.log 2> /dev/null
```

Here the STDOUT of `ln` is appended to the file `/tmp/ln.log`, and the STDERR is redirected to the file `/dev/null`, in order to discard it.



The file `/dev/null` is a special file available on all UNIX systems used to discard output. It is sometimes referred to as the *bit bucket*. If you redirect the output of a command into `/dev/null`, it is discarded. For example, the command

```
rm file > /dev/null
```

discards the output of the `rm` command.

If you use `cat` to display the contents of `/dev/null` to a *file*, the file's contents are erased:

```
$ cat /dev/null > file
```

After this command, the *file* still exists, but its size is zero.

Redirecting STDOUT and STDERR to the Same File

You looked at how to use file descriptors to redirect STDOUT and STDERR to different files, but sometimes you need to redirect both to the same file. In general, you can do this as follows

```
cmd > file 2>&1
list > file 2>&1
```

Here STDOUT (file description 1) and STDERR (file descriptor 2) of *cmd* are redirected into the specified *file*. Here is a situation where it is necessary to redirect both the standard output and the standard error:

```
rm -rf /tmp/my_tmp_dir > /dev/null 2>&1 ; mkdir /tmp/my_tmp_dir
```

In this case you are not interested in the error message or the informational message printed by the *rm* command. You only want to remove the directory, thus its output, or any error message it prints, is redirected to */dev/null*.

If you have a command or *list* that should append its standard error and standard output to a file, you can use one of the following forms of output redirection:

```
cmd >> file 2>&1
list >> file 2>&1
```

An example of a command that might require this is

```
rdate -s ntp.nasa.gov >> /var/log/rdate.log 2>&1
```

Here you are using the *rdate* command to synchronize the time of the local machine to an Internet time server and you want to keep a log of all the messages.

Printing a Message to STDOUT

You can also use this form of output redirection to output error messages on STDERR. The basic syntax is

```
echo str 1>&2
printf format args 1>&2
```

You might also see these commands with the STDOUT file descriptor, 1, omitted:

```
echo string >&2
printf format args >&2
```

Redirecting Two File Descriptors

You can redirect the output from one file descriptor to another file descriptor using the general form of output redirection:

```
n>&m
```

Here n and m are file descriptors (integers). When you let $n=1$ and $m=2$, `STDERR` is redirected to `STDOUT`. The general form of output redirection is often combined with `exec` to duplicate an already open output file description:

```
exec n>&m
```

Here n is a new file descriptor and m is an open output file descriptor. For example if the file descriptor 4 is opened as follows:

```
exec 4>out.txt
```

then the command:

```
exec 5>&4
```

causes file descriptor 5 to become a duplicate of file descriptor 4. Given these two `exec` commands, the output of the following command:

```
date 1>&5
```

will end up in the file `out.txt`.

The general form of input redirection is similar to the general form of output redirection:

```
n<&m
```

Here, n and m are file descriptors (integers). The general form of output redirection is often combined with `exec` to duplicate an already open input file description:

```
exec n<&m
```

Here n is a new file descriptor and m is an open input file descriptor. In the following example, file descriptor 6 becomes a duplicate of `STDIN`:

```
exec 6<&0
```

Closing File Descriptors

The following syntax can be used to close an open file descriptor:

```
exec n>-
```

Here n is an open file descriptor. When a file descriptor is closed, trying to read or write from it results in an error. The following example closes the previously opened file descriptor 4:

```
exec 4>-
```

Summary

In this chapter, you learned about the concept of input and output and examined the `echo` and `printf` commands that are used to produce messages from within shell scripts. You also learned about output redirection, and covered the methods of redirecting and appending the output of a command to a file. You also learned about the concept of a file descriptor and saw several aspects of its use, including opening files for reading and writing, closing files, and redirecting the output of two file descriptors to one source.

In subsequent chapters, you will expand on the material covered here, and you will see many more applications of both input and output redirection along with the use of file descriptors.

Questions

1. Which file descriptors are associated with `STDOUT`, `STDERR` and `STDIN`?
2. Use `printf` to convert the numbers 16, 255, and 65535 into hexadecimal and octal.
3. Given the following script:

```
exec 4>out.txt
exec 5>&4
exec 1>&5
date
```

Where does the output from `date` end up?

Terms

Escape sequence A special sequence of characters that represents another character.

File descriptor An integer that is associated with a file. Enables you to read and write from a file using the integer instead of the file's name.

Input redirection In UNIX, the process of sending input to a command from a file.

Output redirection In UNIX, the process of capturing the output of a command and storing it in a file. It redirects the output of a command into a file instead of the screen.

STDERR Standard Error. A special type of output used for error messages. The file descriptor for `STDERR` is 2.

STDIN Standard Input. User input is read from `STDIN`. The file descriptor for `STDIN` is 0.

STDOUT Standard Output. The output of scripts is usually to `STDOUT`. The file descriptor for `STDOUT` is 1.

This page intentionally left blank

HOUR 6



Manipulating File Attributes

In addition to files and directories, UNIX supports several special file types along with a set of attributes for each file and directory. Shell scripts are often called upon to create special files and manipulate file attributes. This chapter discusses the following topics related to special files and file attributes:

- Creating links
- Modifying file permissions
- Modifying file ownership and group membership

File Types

UNIX files can contain important data and executable programs or they can represent devices, directories, or pointers to other files. This section looks at the different types of files available under UNIX.

Determining a File's Type

You can determine a file's type by using the `-l` option of the `ls` command. When this option is specified the output of `ls` contains a file's type and its attributes in addition to its name. For example, the command

```
$ ls -l /home/ranga/.profile
```

produces the following output:

```
-rwxr-xr-x  1 ranga  users      2368 Jul 11 15:57 .profile*
```

As you can see, the very first character in the output is a hyphen (`-`). This indicates that this is a regular file. For special files, the first character is one of the letters given in Table 6.1. The subsequent sections describe each of these special files in detail.

TABLE 6.1 Special Characters for Different File Types

<i>Character</i>	<i>File Type</i>
-	Regular file
l	Symbolic link
c	Character special
b	Block special
p	Named pipe
d	Directory file

To obtain file type information for a directory, you need to specify the `-d` option along with the `-l` option. For example, in order to obtain file type information for the directory `/home/ranga`, you use the following command:

```
$ ls -ld /home/ranga
```

This produces the following output:

```
drwxr-xr-x 27 ranga  users      2048 Jul 23 23:49 /home/ranga/
```

Regular Files

Regular files are the most common type of files on UNIX systems. They can be used to store any kind of data, including binary data that the system can execute. Often determining that a file is a regular one tells you very little about the file. Usually you need to know whether a particular file is a binary program, a shell script, or a C language library. In these instances, the `file` command is very useful. Its syntax is as follows:

```
file filename
```

Here, *filename* is the name of the file you want more information about. For example, the command:

```
$ file /bin/sh
```

will produce output similar to the following:

```
/bin/sh: ELF 32-bit MSB executable SPARC Version 1,  
↳statically linked, stripped
```

Based on this output, you can tell that the file, */bin/sh*, is an executable program for SPARC-based system. The output on your system will most likely be different.

Links

A *link* is a file that points to another file on the system. Links are useful for maintaining multiple copies of a file in several locations on the system without using up storage for the copies. Because a link just points to another file, changing the content of the link alters the content of the original file. Similarly, altering the content of the original file appears to alter the content of the link.

UNIX supports two types of links, *hard links* and *symbolic links*.

Hard Links

A hard link is a special directory entry that points to another file. Hard links have some limitations:

- A hard link cannot point to a directory; it can only point to a file.
- Hard links are indistinguishable from the file that it points to; there is no way to tell if a particular file is a hard link or the original file.

Hard links can be created using the `ln` (short for link) command. Its syntax is as follows:

```
ln src target
```

Here *src* is the pathname that the pathname *target* should point to. For example, if you want the hard link *banana* to point to the file *apple*, you can create the link as follows:

```
$ ln apple banana
```

If there is a problem creating the hard link, `ln` displays an error message; otherwise, it displays no output.

When a hard link is moved from one directory to another, it can continue to point to the original file without any problems. For example, consider the following commands:

```
$ echo I drink lemonade in the summer > lemonade  
$ cat lemonade  
I drink lemonade in the summer
```

```
$ ln lemonade summer_drink
$ cat summer_drink
I drink lemonade in the summer
$ mv summer_drink ..
$ cat ../summer_drink
I drink lemonade in the summer
```

As you can see from the output, the file `summer_drink` continues to point to the original file even after it is moved out of the directory where it was created.

When a hard link is removed, the original file that it points to is not affected; only the link is deleted. If the original file is deleted, the entry for the original file is removed, but its content remains on disk until the link is deleted. For example:

```
$ echo Pomegranates are very juicy > juicy
$ ln juicy pomegranate
$ cat pomegranate
Pomegranates are very juicy
$ rm juicy
$ cat pomegranate juicy
Pomegranates are very juicy
cat: juicy: No such file or directory
```

As you can see the file `pomegranate` continues to point to the content of the file `juicy` even after the file `juicy` has been removed.



If a file has multiple hard links to it, simply removing the file will not be sufficient to free up the disk space; you will have to remove all of the hard links to that file.

Symbolic Links

A symbolic link or *symlink* is a special file that stores a pathname to another file. When a symlink is accessed, the system automatically reads the pathname stored in the symlink and accesses the file corresponding to that pathname, thus a symlink can point to any file on the system. The pathname stored in a symlink can be an absolute or relative pathname. If a relative pathname is stored, it is relative to the directory where the link is located rather than the current working directory.

The `ls -l` output for a symbolic link looks similar to the following:

```
lrwxrwxrwx  1 root  root          9 Oct 23 13:58 /bin/ -> ./usr/bin/
```

In this example, the first character `l` specifies that the file is a symlink. The output also indicates that the file `/bin` is a link to the file `./usr/bin`, which is located in the directory `.`

Symlinks are created by using the `-s` option of `ln`. The syntax is as follows:

```
ln -s src target
```

Here *src* is the pathname that the pathname *target* should point to. For example, if you want to create the symlink `citrus` that points to the file `lime`, you can use the following command:

```
ln -s lime citrus
```

If there is a problem creating the symlink, `ln` displays an error message; otherwise, it displays no output.

If a symlink is created using a relative path, then it may not work properly when moved from the directory in which it was created to another directory. For example:

```
$ echo Persimmons are bitter until they ripen > persimmon
$ ln -s persimmon bitter
$ cat bitter
Persimmons are bitter until they ripen
$ mv bitter ..
$ cat ../bitter
cat: ../bitter: No such file or directory
$ ls -l ../bitter
lrwxrwxrwt 1 root  wheel  9 Jan 13 00:16 ../bitter@ -> persimmon
```

As you can see from the output, the link `bitter` correctly pointed to the file `persimmon` while the two files were located in the same directory. When `bitter` was moved, the link stopped working because a file named `persimmon` did not exist in `bitter`'s new directory. This problem can be avoided by using absolute paths when creating symlinks.

When a symlink is removed the file it points to is not affected. When the file that a symlink points to is removed or moved to a different location the symlink will cease to function properly. For example:

```
$ echo Plums were plentiful this year > plums
$ ln -s plums plentiful
$ cat plentiful
Plums were plentiful this year
$ rm plums
$ cat plentiful
cat: plentiful: No such file or directory
$ ls -l plentiful
lrwxr-xr-x 1 ranga  wheel  5 Jan 13 00:25 plentiful@ -> plums
```

As you can see from the output, the link `plentiful` points to the file `plums` even after that file has been removed. This causes the error message from `cat`.

Common Errors

Two common errors encountered when creating links occur when

- *target* already exists.
- *target* is a directory.

If *target* is a file, `ln` will not create the requested link. For example, if the file `.exrc` exists in the current directory, the following command:

```
$ ln -s /etc/exrc .exrc
```

produces the following error message:

```
ln: cannot create .exrc: File exists
```

If *target* is a directory, `ln` creates the link in that directory with the same filename as *src*. For example, if the directory `pub` exists in the current directory, the following command:

```
$ ln -s /home/ftp/pub/ranga pub
```

creates the link `pub/ranga` rather than complaining that the destination is a directory. Forgetting about this behavior is a common source of problems in shell scripts.

Device Files

In UNIX, devices, such as hard drives, keyboards, and printers, are accessed via *device files*. There are two types of device files: *character special files* and *block special files*.

Device files are normally located in the `/dev` directory.

Character Special Files

Character special files provide a mechanism for communicating with a device one character at a time. Usually character devices represent a “raw” device. The output of `ls -l` on a character special file looks like the following:

```
crw----- 1 ranga users 4, 0 Feb 7 13:47 /dev/tty0
```

The first letter in the output, `c`, indicates that this is a character special file. The two extra numbers before the date are known as the *major* and *minor* device numbers. UNIX uses these two numbers to identify the device driver that is connected to the character special file.

Block Special Files

Block special files provide a mechanism for communicating with devices by transferring large blocks of data rather than single characters. Block special files are typically used to

access hard drives and removable media. The output of `ls -l` on a block special file looks like the following:

```
brw-rw---- 1 root    disk      8,  0 Feb  7 13:47 /dev/sda
```

The first letter in the output, `b`, indicates that this file is a block special file. The major and minor numbers for the file identify the device driver that is connected to the block special file.

Named Pipes

An important feature of UNIX is that you can redirect the output of one program to the input of another program with very little work. For example, the following command:

```
$ who | grep ranga
```

takes the output of the `who` command and makes it the input to the `grep` command. On the command line, temporary anonymous pipes are used, but sometimes a program needs more control over the communication channel. *Named pipes* are files that act just like temporary anonymous pipes on the command line.

Named pipes can be created using the `mkfifo` command. Its syntax is as follows:

```
mkfifo file
```

Here *file* is the filename that you want to give the pipe. For example, the following command creates a named pipe with the filename `mypipe`:

```
$ mkfifo mypipe
```

The `ls -l` output for this named pipe will be similar to the following:

```
prw-r--r-- 1 ranga  wheel  0 Nov 22 17:39 mypipe
```

The first character, `p`, indicates that this file is a named pipe.

Owners, Groups, and Permissions

File permissions and file ownership are important components of UNIX because they provide a secure method for storing files. Every file in UNIX has the following attributes:

- Owner permissions
- Group permissions
- Other (world) permissions

The owner's permissions determine which actions the owner of the file can perform on the file. The group's permissions determine which actions a user, who is a member of the

group that a file belongs to, can perform on the file. The permissions for others indicate which action all other users can perform on the file.

The actions that can be performed on a file are *read*, *write*, and *execute*. If a user has read permissions, that user can view the contents of a file. A user with write permissions can change the contents of a file, whereas a user with execute permissions can execute that file.

Viewing Permissions

The `ls -l` command displays the permissions of a file. For example, the following command:

```
$ ls -l .profile
```

produces the following output:

```
-rwxr-xr-x  1 ranga  users      2368 Jul 11 15:57 .profile
```

From the output, you can tell that this is a regular file. The characters that appear after the first dash (-) indicate the permissions for the file. After the permissions, the owner and the group are listed. For this file, the owner is `ranga` and the group is `users`.

The first three characters indicate the permissions for the owner of the file, the next three characters indicate the permissions for the group of the file, and the last three characters indicate the permissions for all other users. The significance of the individual characters is explained in Table 6.2.

TABLE 6.2 Basic Permissions

<i>Letter</i>	<i>Permission</i>	<i>Definition</i>
r	Read	The user can view the contents of the file.
w	Write	The user can alter the contents of the file.
x	Execute	The user can run the file, which is likely a program. For directories, the execute permission must be set in order for users to access the directory.

The permissions for the file in the previous example indicates that the user has read, write, and execute permissions, whereas members of the group `users` and all other users have only read and execute permissions.

Directory Permissions

The `x` bit on a directory grants access to the directory. The read and write permissions have no effect if the access bit is not set. The read permission on a directory enables users to use the `ls` command to view files and their attributes that are located in the

directory. The write permission on a directory allows users to add and remove files from the directory. A directory that grants a user only execute permission will not enable the user to view the contents of the directory or add or delete any files from the directory, but it will let the user run executable files located in the directory.



To ensure that your files are secure, check both the file permissions and the permissions of the directory where the file is located.

If a file has write permission for owner, group, and other, the file is insecure. If a file is in a directory that has write and execute permissions for owner, group, and other, all files located in the directory are insecure, no matter what the permissions are on the files themselves.

SUID and SGID File Permission

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task. As an example, when you change your password with the `passwd` command, your password is stored in the file `/etc/shadow`. As a regular user, you do not have read or write access to this file for security reasons, but when you change your password, you need to have write permission to this file. This means that the `passwd` program has to give you additional permissions so that you can write to the file `/etc/shadow`.

Additional permissions are given to programs via a mechanism known as the *Set User ID (SUID)* and *Set Group ID (SGID)* bits. When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program. When a program is executed, it normally executes with the group permissions of the user. A file that has the SGID bit set will be executed using those group permissions.

As an example, the `passwd` command used to change your password is owned by the root and has the set SUID bit enabled. When you execute it, you effectively become the root while the command runs.

The SUID and SGID bits appear as the letter `s` if SUID or SGID permission has been enabled on that file. The SUID `s` bit is located in the permission bits where the owners who execute permission would normally reside. For example, the following command:

```
$ ls -l /usr/bin/passwd
```

produces the following output:

```
-r-sr-xr-x  1 root  bin           19031 Feb  7 13:47 /usr/bin/passwd*
```

which shows that the SUID bit is set and that the root owns the command. If a capital letter S appears instead of the lowercase s it indicates that the execute bit is not set.

The *SUID bit* or *sticky bit* imposes extra file removal permissions on a directory. A directory with write permissions enabled for a user enables that user to add and delete any files from this directory. If the sticky bit is enabled on the directory, files can be removed only if you are one of the following users:

- The owner of the sticky directory
- The owner the file being removed
- The super user, root

You should consider enabling the sticky bit for any directories to which non-privileged users can write. Examples of such directories include temporary directories and public file upload sites.

Directories can also take advantage of the SGID bit. If a directory has the SGID bit set, any new files added to the directory automatically inherit that directory's group, instead of the group of the user writing the file.

Changing File and Directory Permissions

The `chmod` command changes the permissions on a file or directory. Its syntax is as follows:

```
chmod expression files
```

Here, *expression* is a statement that indicates how the permissions are to be changed. There are two types of *expressions*: symbolic and octal. The symbolic expression method uses letters to alter the permissions, and the octal expression method uses numbers. The numbers in the octal method are base-8 (octal) numbers ranging from 0 to 7.

Symbolic Method

A symbolic *expression* uses syntax of the form:

```
(who)(action)(permissions)
```

Table 6.3 shows the possible values for *who*, Table 6.4 shows the possible *actions*, and Table 6.5 shows the possible *permissions* settings. Using these three reference tables, you can build *expressions*.

TABLE 6.3 who

<i>Letter</i>	<i>Represents</i>
u	Owner
g	Group
o	Other
a	All

TABLE 6.4 actions

<i>Symbol</i>	<i>Represents</i>
+	Adding permissions to the file
-	Removing permissions from the file
=	Explicitly set the file permissions

TABLE 6.5 permissions

<i>Letter</i>	<i>Represents</i>
r	Read
w	Write
x	Execute
t	Sticky bit
s	SUID or SGID

Now let's look at a few examples of using `chmod`. To give the “world” read access to all files in a directory, you can use one of the following commands:

```
$ chmod a=r *
$ chmod guo=r *
```

If the command is successful, there is no output.

To stop anyone except the owner of the file `.profile` from writing to it, try this:

```
$ chmod go-w .profile
```

To deny access to the files in your home directory, you can try one of the following commands:

```
$ cd ; chmod go= *
$ cd ; chmod go-rwx *
```

When specifying the user's part or the permission's part, the order in which you give the letters is irrelevant. Thus these commands are equivalent:

```
$ chmod guo+rx *
$ chmod uog+xr *
```

If you need to apply more than one set of permission changes to a file or files, you can use a comma-separated list. For example:

```
$ chmod go-w,a+x a.out
```

removes the groups and “world” write permission on `a.out` and adds the execute permission for everyone.

To set the SUID and SGID bits for your home directory, try the following:

```
$ cd ; chmod ug+s .
```

So far, the examples you have examined involve changing the permissions for files in a directory. However, `chmod` also enables you to change the permissions for every file in a directory (including the files in subdirectories) by using the `-R` option.

For example, if the directory `pub` contains the following directories:

```
$ ls pub
./      ../      README  faqs/   src/
```

you can change the permission read permissions of the file `README` along with the files contained in the directories `faqs` and `src` with the following command:

```
$ chmod -R o+r pub
```

Octal Method

By changing permissions with an octal expression, you can only explicitly set file permissions. This method uses a single number to assign the desired permission to each of the three categories of users (owner, group, and other). The values of the individual permissions are the following:

- Read permission has a value of 4
- Write permission has a value of 2
- Execute permission has a value of 1

Adding the value of the permissions that you want to grant will give you a number between 0 and 7. This number will be used to specify the permissions for the owner, group, and finally the other category.

Setting SUID and SGID using the octal method places these bits out in front of the standard permissions. The permissions SUID and SGID take on the values 4 and 2, respectively.

Let's look at some of the examples to get an idea of how to use the octal method of changing permissions. In order to set the "world" read access to all files in a directory, do the following:

```
chmod 0444 *
```

To stop anyone except the owner of the file `.profile` from writing to it, do this:

```
chmod 0600 .profile
```

Common Errors

Many new users find the octal specification of file permissions confusing. The most important point to keep in mind is that the octal method sets or assigns permissions to a file, but it does not add or delete them. This means that the octal mode does not have an equivalent to

```
chmod u+rw .profile
```

The closest possible octal version is

```
chmod 0600 .profile
```

But this removes permissions for everyone except the user. It can also reduce the user's permissions by removing that user's execute permission.

Changing Owners and Groups

The `chown` (short for change owner) changes the ownership of a file, whereas the `chgrp` (short for change group) changes the group membership of a file. The `chgrp` command is not available on some older systems, thus the `chown` command must be used in its place. This section shows how to use both `chown` and `chgrp` to change the group of a file.

Changing Ownership

The `chown` command changes the ownership of a file. The basic syntax is as follows:

```
chown user:group files
```

Here, *user* is the name of a user on the system or the user ID (uid) of a user on the system, *group* is the name of a group on the system or the group ID (GID) of a group on the system, and *files* is a list of files to apply the changes to. If *group* is omitted, only the owner of the file is changed. If *user* is omitted, only the group of the file is changed.

The following example illustrates the use of this command to change the owner of a file:

```
chown ranga: /home/httpd/html/users/ranga
```

This changes the owner of the given directory to the user ranga. The following example illustrates the use of `chown` to change just the group of a file:

```
chown :authors /home/ranga/docs/ch5.doc
```

In this case the group of the given file is changed to the group authors.

The `chown` command will recursively change the ownership of all files when the `-R` option is included. For example, the command

```
chown -R ranga: /home/httpd/html/users/ranga
```

changes the owner of all the files and subdirectories located under the given directory to be the user ranga.

The super user, root, has the unrestricted capability to change the ownership of a file, but some restrictions occur for normal users. Normal users can change only the owner of files they own.



Be careful when using the `chown` command. If you give another user ownership of a file, you cannot regain ownership of that file. Only the new owner of the file or the super user can return the ownership to you.

On some systems, the `chown` command is disabled for normal user use. This generally happens if the system is running disk quotas. Under a disk quota system, users might be allowed to store only 100MB of files, but if they change the ownership of some files, their free available disk space increases, and they still have access to their files.

Changing Group Ownership

The `chgrp` command changes the group membership of a file. The syntax of this command is as follows:

```
chgrp group files
```

Here *group* can be either the name of a group or the GID of a group on the system and *files* is a list of files to apply the changes to. For example, the command:

```
chgrp authors /home/ranga/docs/ch5.doc
```

changes the group of the given file to be the group authors. Just like `chown`, all versions of `chgrp` understand the `-R` option also.



Some older systems do not include the `chgrp` command. If you are using such a system, you can use the `chown` command in place of the `chgrp` command.

To use `chown` in place of `chgrp`, you can invoke it as follows:

```
chown :group files
```

Here *group* is either the name of a group or the GID of a group on the system and *files* is a list of files to apply the changes to.

Summary

In this chapter, you learned several important topics relating to files and file permissions. Specifically, you examined:

- Determining a file's type
- Changing file and directory permissions using symbolic and octal notation
- Enabling SUID and SGID permissions for files and directories
- Changing the owner of a file or directory
- Changing the group of a file or directory

As you will see in subsequent chapters, each of these tasks is important in shell scripts.

Questions

For these three questions, refer to the following `ls -l` output:

```
crw-r----- 1 bin      sys      188 0x001000 Oct 13 00:31 /dev/
└─rdskc0t1d0
-r--r--r-- 1 root     sys      418 Oct 13 16:25 /etc/passwd
drwxrwxrwx 10 bin     bin      1024 Oct 15 20:27 /usr/local/
-r-sr-xr-x 1 root     bin      28672 Nov  6 1997 /usr/sbin/ping
```

1. Identify the file type of each of the files given above.
2. Identify the owner and group of each of the files given above.
3. Describe the permissions for the owner, group, and all “other” users for each of the files given above.

Terms

Regular files The most common type of files on UNIX systems and can be used to store any kind of data, including binary data that the system can execute.

Link A file that points to another file on the system.

Hard link A special directory entry that points to another file.

Symbolic link A special file that stores a pathname to another file. A symbolic link is often referred to as a *symlink*.

Character special files Provide a mechanism for communicating with a device one character at a time.

Block special files Provide a mechanism for communicating devices by transferring large blocks of data.

HOUR 7



Processes

In UNIX every program runs as a *process*. A process is an instance of running a program. If, for example, three people are running the same program simultaneously, there are three processes there, not just one. In this chapter you will learn about processes and jobs and the different modes in which they can be executed. You will also look at the commands used to list and terminate processes. Specifically the topics you will examine are:

- Starting processes
- Listing running processes
- Killing processes
- Manipulating parent and child processes

Starting a Process

Whenever you issue a command in UNIX, it creates, or *starts*, a new process on your behalf. When you tried out the `ls` command to list directory contents in Chapter 4, “Working with Directories,” the system started a process, the `ls` command, for you.

UNIX tracks processes through a five-digit ID number known as the *pid* (short for process identifier). Each process in the system has a unique pid between 1 and 32,767. Pids eventually repeat when all the possible numbers are used. Two processes with the same pid cannot be concurrently executed on the system.

Foreground Processes

By default, every process runs in the foreground. It gets its input from the keyboard and sends its output to the screen. You can see this happen with the `ls` command. For example, when you execute the `ls` command:

```
$ ls
```

It executes and displays the contents of the current directory:

```
Desktop  Downloads Library  Music    Public  
Documents Icon?    Movies  Pictures Sites
```

While the command is running, you cannot run any other commands (start any other processes). You can enter commands, but no prompt appears and nothing happens until this command completes. For the `ls` command, which usually runs very quickly, this is not a problem, but if you have a program that runs for a long time—such as a large compile, database query, program that calculates pi, or a server—the terminal will be tied up.

Fortunately, you do not have to wait for one process to complete before you can start another. UNIX provides facilities for starting processes in the background, suspending foreground processes, and moving processes between the foreground and background.

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits. The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) to the end of the command. For example, if you execute `ls` in the background:

```
$ ls &
```

the output will be similar the following:

```
[1] 621  
$ Desktop  Downloads Library  Music    Public  
Documents Icon?    Movies  Pictures Sites
```

The first line of output, produced by the shell, tells you that the process is running in the background:

```
[1] 621
```

This line contains two pieces of information about the background process—the *job ID* (short for job identifier) and the pid. The shell assigns a job ID for every command that is executed in the background.

If you execute this command, you might notice that you do not get back a prompt after the last line of the directory listing. That's because the prompt actually appears immediately after the job/pid line, next to Desktop. You can enter a command immediately instead of waiting for `ls` to finish. If you press the Enter key now, you will see something similar to the following:

```
[1] + Done                ls &
$
```

The first line tells you that the background `ls` job finished successfully. The second is a prompt for another command.

You will see a different completion message if an error occurs. For example, if you try to list the file with the name `no_such_file`, you will get an error:

```
$ ls no_such_file &
[1] 25389
$ no_such_file: No such file or directory
```

The first line is the background process information and the second shows the prompt for the next command and the output from `ls`—the error message. If you press Enter again, the following message appears on your screen:

```
[1] + Done(2)             ls no_such_file &
$
```

This shows that the `ls` command exited with nonzero status, in this case, 2. The dollar sign (\$) on the next line is the command prompt.

Background Processes That Require Input

If you run a background process that requires input and do not redirect it to read a file instead of the keyboard, the process will stop. Pressing Enter at an empty command prompt or starting a command will return a message to that effect. For example consider the following script:

```
#!/bin/sh
read LINE
echo $LINE
exit $?
```

Call this script `read.sh` and execute it in the background as follows:

```
$ read.sh &
$
```

Because this command does not produce any output until you give it input, all you see is the command prompt. Pressing Enter at the prompt results in a message similar to the following:

```
[1] + Stopped (SIGTTIN)      read.sh &
```

This message informs you that the command `read.sh` is currently stopped due to the signal `SIGTTIN`. This signal (SIG) tells you that the program is waiting for terminal (TT) input (IN). See Chapter 19, “Signals,” for more information on signals. In `bash` and `zsh` the information about the signal is not presented.

If you get a message like this, you have two choices. You can kill the process and rerun it with input redirected, or you can bring the process to the foreground, give it the input it needs, and then let it continue as a foreground or background process. This chapter explains how to handle either of these choices.

Moving a Foreground Process to the Background

In addition to running a process in the background using `&`, you can move a foreground process into the background. While a foreground process runs, the shell does not process any new commands. Before you can enter any commands, you have to suspend the foreground process to get a command prompt. The suspend key on most UNIX systems is `Ctrl+Z`.



You can determine which key performs which function by using the `stty` command. By entering

```
$ stty -a
```

you are shown the following, along with a lot of other information:

```
intr = ^C; quit = ^\; erase = ^H; kill = ^U;
➔ eof = ^D; eol = ^@
eol2 = ^@; start = ^Q; stop = ^S; susp = ^Z;
➔ dsusp = ^Y; reprint = ^R
discard = ^O; werase = ^W; lnext = ^V
```

The entry after `susp` (`^Z` in this example) is the key that suspends a foreground process. The character `^` stands for `Ctrl`. If `Ctrl+Z` does not work for you, use the `stty` command as shown previously to determine the key for your system.

When a foreground process is suspended, a command prompt enables you to enter more commands; the original process is still in memory but is not getting any CPU time. To resume the foreground process, you have two choices—background and foreground. The `bg` command enables you to resume the suspended process in the background while the `fg` command returns it to the foreground. This section covers the `bg` command, whereas the `fg` command is covered in the next section.

For example, say you start a long-running process, in this case `long_running_process`:

```
$ long_running_process
```

While it is running, you decide that it should run in the background so your terminal is not tied up. To do that, you press the `Ctrl+Z` keys and see the following (the `^Z` is your `Ctrl+Z` keys being echoed):

```
^Z[1] + Stopped (SIGTSTP)      long_running_process
$
```

You are told the job number (1) and that the process is `Stopped`, then you get a prompt. The actual message might be different depending on the shell you are using. To resume a job in the background, you enter the `bg` command as follows:

```
$ bg
[1]      long_running_process &
$
```

As a result, the process runs in the background. Note the last character on the second line, the ampersand (&). As a reminder, the shell displays the ampersand there to remind you that the job is running in the background. It behaves just like a command where you type the ampersand at the end of the line.

By default, the `bg` command moves the most recently suspended process to the background. You can have multiple processes suspended at one time. To differentiate them, you can use the job number prefixed with a percent sign (%) on the command line.

In the following example, you start two long-running processes, suspend both of them, and put the first one into the background. The next few lines show starting and suspending two foreground processes:

```
$ long_running_process
^Z[1] + Stopped (SIGTSTP)      long_running_process
$ long_running_process2
^Z[2] + Stopped (SIGTSTP)      long_running_process2
$
```

To move the first one to the background, you use the following:

```
$ bg %1
[1]      long_running_process &
$
```

The second process is still suspended and can be moved to the background as follows:

```
$ bg %2
[2]    long_running_process2 &
$
```

The capability to specify which job to perform an action on (move to foreground or background for instance) shows the importance of having job numbers assigned to background processes.

Moving a Background Process to the Foreground (fg Command)

When you have a process that is in the background or suspended, you can move it to the foreground with the `fg` command. By default, the process most recently suspended or moved to the background moves to the foreground. You can also specify the job using its job number.



If you're ever in doubt about which job will be moved to the background or foreground, don't guess. Put the job number on the `bg` or `fg` command, prefixed with a percent sign.

Using the long-running process in the previous section, a foreground process is suspended and moved into the background in the following example:

```
$ long_running_process
^Z[1] + Stopped (SIGTSTP)    long_running_process
$ bg
[1]    long_running_process &
$
```

You can move it back to the foreground as follows:

```
$ fg %1
long_running_process
```

The second line shows you which command you moved back to the foreground. The same thing would happen if the job was moved back to the foreground after being suspended.

Keeping Background Processes Around (nohup Command)

When you log out, the default action is to terminate all the processes that you are running. You can prevent this behavior from occurring on your background processes using the `nohup` (short for no hang up) command. The `nohup` command is simple to use—

just add it before the command you actually want to run. Because `nohup` is designed to run when there is no terminal attached, it wants you to redirect output to a file. If you do not, `nohup` redirects it automatically to a file known as `nohup.out`.

Running a process in the background with `nohup` looks like the following:

```
$ nohup ls &
[1] 6695
$ Sending output to nohup.out
```

Because you did not redirect the output from `nohup`, it is automatically redirected for you. If you redirected the output, you would not see the second message. After waiting a few moments and pressing Enter, you would see the following:

```
[1] + Done                nohup ls &
$
```

Waiting for Background Processes to Finish (`wait` Command)

There are two ways to wait for a background process to finish before doing something else. You can press the Enter key every few minutes until you get the completion message, or you can use the `wait` command.

There are three ways to use the `wait` command—with no options (the default), with a process ID, or with a job number prefixed with a percent sign. The command will wait for the completion of the job or process you specify.

If you do not specify a job or process (the default setting), the `wait` command waits for all background jobs to finish. Using `wait` without any options is useful in a shell script that starts a series of background jobs. When they are all done, it can continue processing.

With the `ls` command from the previous example running, you can force a wait with the following command:

```
$ wait %1
```

You cannot enter another command until job number 1 finishes. When you use `wait`, you do not get the completion message.

Listing and Terminating Processes

You can start processes in the foreground and background, suspend them, and move them between the foreground and background, but how do you know which commands are running? There are two commands to help you find out—`jobs` and `ps`.

jobs

The `jobs` command shows you the processes that are suspended and the ones running in the background. Because `jobs` runs as a foreground process, it cannot show you active foreground processes. In the following example, you have three jobs—the first one (job 3) is running, the second (job 2) is suspended (a foreground process after `Ctrl+Z` was issued), and the third one (job 1) is stopped in the background waiting for keyboard input:

```
$ jobs
[3] +  Running                first_one &
[2] -  Stopped (SIGTSTP)      second_one
[1]   Stopped (SIGTTIN)      third_one &
```

You can manipulate these jobs with the `fg` and `bg` commands. The most recent job is job number 3 (shown with a plus sign); this is the one that `bg` or `fg` act on if no job number is supplied. The most recent job before that is job number two (shown with a minus sign).



The reason for the plus and minus symbols on the `jobs` listing is that job numbers are reassigned when one completes and another starts. In the previous example, if job number 2 finishes and you start another job, it is assigned job number 2 and a plus sign because it is the most recent job.

ps Command

Another command that shows all processes running is the `ps` (short for process status) command. By default, it shows those processes that you are running. It also accepts many options, a few of which are described here.

The simplest example (with the same three jobs running as the previous example) is the `ps` command alone:

```
$ ps
  PID TTY          TIME CMD
 6738 pts/6        0:00 first_one
 6739 pts/6        0:00 second_one
 3662 pts/6        0:00 ksh
 8062 pts/6        0:00 ps
 6770 pts/6        0:01 third_one
```

For each running process, this provides four pieces of information: the pid, the TTY (terminal running this process), the time or amount of CPU consumed by this process, and the command name running. Although you are running three jobs, you have five processes. Of course, one of the extra processes is the `ps` command itself. The remaining process, `ksh`, is the shell.

If you are using BSD or older versions of Linux, your output will be similar to the following:

```
$ ps
  PID TT  STAT      TIME COMMAND
13049 q0  Ss      0:00.06 -ksh (ksh)
13108 q0  R+      0:00.01 ps
```

For each running process, this provides you with five pieces of information: the pid, the TT (terminal running this process), STAT (the state of the job), the TIME or amount of CPU consumed by this process, and finally the command name running.

One of the most commonly used flags for ps is the `-f` (short for full) option, which provides more information as shown in the following example:

```
$ ps -f
  UID  PID  PPID  C   STIME TTY      TIME CMD
dhorvath 6738 3662  0 10:23:03 pts/6   0:00 first_one
dhorvath 6739 3662  0 10:22:54 pts/6   0:00 second_one
dhorvath 3662 3657  0 08:10:53 pts/6   0:00 -ksh
dhorvath 6892 3662  4 10:51:50 pts/6   0:00 ps -f
dhorvath 6770 3662  2 10:35:45 pts/6   0:03 third_one
```

Table 7.1 shows the meaning of each of these columns. The BSD or Linux equivalent of the `-f` option is `-ux`. The column heading in BSD and Linux might be slightly different than those described in Table 7.1.

TABLE 7.1 ps -f Columns

<i>Heading</i>	<i>Description</i>
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
unlabeled	Nice value—used in calculating process priority.
STIME	Process start time (when it began).
CMD	The command that started this process. CMD with <code>-f</code> is different from CMD without it; it shows any command-line options and arguments.

Note that the PPID of all the commands is 3662, which is the pid of the ksh instance that is executing. Because all of the processes were started in ksh, it is the parent process for all of these processes.



You might be wondering why `TIME` changed for `third_one`. Between the time I entered the `ps` and the `ps -f` commands, `third_one` used some CPU time—two seconds. On larger UNIX servers, a lot of work can be done with very little CPU time. That's why the `ps` command is showing with zero CPU time; it used time, but not enough to round up to one second.

Two more common options are `-e` (short for every) and `-u` (short for user). The `-e` option is handy if you want to see whether the database is running or whether someone is playing Zork (an old text-based computer game). The BSD and Linux equivalent of the `-e` option is the `-a` option. Because so many processes run on a busy system, it is common to pipe the output of `ps -e` or `ps -a` to a text filter like `grep` (see Chapter 15, “Text Filters”).

The `-u` option is handy if you want to see what a specific user is doing—are they busy or do they have time to chat; is your boss busy or checking to make sure you're not playing Zork? With `-u`, you specify the user you want to list after the `-u`. The BSD or Linux equivalent of the `-u` option is the `-U` option.

Killing a Process (`kill` Command)

Another handy command to use with jobs and processes is the `kill` command. As the name implies, the `kill` command kills, or terminates, a process. Just like the `fg` and `bg` commands, the job number is prefixed with a percent sign. To kill job number 1 in the earlier example regarding waiting for keyboard input, you can use the following:

```
$ kill %1
[1] - Terminated          third_one &
$
```

You can also kill a specific process by specifying the `pid` on the command line without the percent sign used with job numbers. To kill job number 2 (process 6738) in the earlier example using process ID, you can use the following:

```
$ kill 6739
$
```

Parent and Child Processes

In the `ps -f` example in the `ps` command section, each process has two ID numbers assigned to it: process ID (`pid`) and parent process ID (`ppid`). Each user process in the system has a parent process. Most commands that you execute have the shell as their parent. The parent of your shell is usually the operating system or the terminal communications process (for example, `in.telnetd` for telnet connections).

If you examine the output of `ps -ef`, you see that the parent process of all your commands is 3662, the pid of the login shell (in this ksh):

```
$ ps -f
  UID  PID  PPID  C   STIME TTY      TIME CMD
dhorvath 6738 3662  0 10:23:03 pts/6    0:00 first_one
dhorvath 6739 3662  0 10:22:54 pts/6    0:00 second_one
dhorvath 3662 3657  0 08:10:53 pts/6    0:00 -ksh
dhorvath 6892 3662  4 10:51:50 pts/6    0:00 ps -f
dhorvath 6770 3662  2 10:35:45 pts/6    0:03 third_one
```

As you can see, the ppid of ksh is 3657. The output on your system, as well as your shell and its process ID, will most likely be different. Using `ps -ef` (or `ps -aux` on some systems) and `grep` to find that number, you see the following:

```
$ ps -ef | grep 3657
dhorvath 9778 3662  4 10:52:50 pts/6    0:00 ps -f
dhorvath 9779 3662  0 10:52:51 pts/6    0:00 grep 3657
  root 3657   711  0 08:10:53 ?          0:00 in.telnetd
dhorvath 3657 3662  0 08:10:53 pts/6    0:00 -ksh
```

This tells you that the terminal session is being handled by `in.telnetd` (the telnet daemon), which is the parent of ksh. There is a parent-child relationship between processes. `in.telnetd` is the parent of ksh, which is the child of `in.telnetd`, but also the parent of `ps` and `grep`.

When a child is *forked*, or created, from its parent, it receives a copy of the parent's environment, including environment variables. The child can change its own environment, but those changes do not reflect in the parent and go away when the child exits.

Subshells

Whenever you run a shell script, in addition to any commands in the script, another copy of the shell interpreter is created. This new shell is known as a *subshell*, just as a directory contained in or under another is known as a subdirectory.

The best way to show this is with an example. For example, consider the following script, which runs `ps` and exits:

```
#!/bin/ksh
ps -ef | grep dhorvath
exit 0
```

When run, `psit` produces the following:

```
$ psit
dhorvath 9830 3662  0 13:58:42 pts/6    0:00 ksh psit
dhorvath 9831 9830 19 14:05:24 pts/6    0:00 ps -ef
dhorvath 3662 3657  0 08:10:53 pts/6    0:00 -ksh
dhorvath 9832 9830  0 13:58:42 pts/6    0:00 grep dhorvath
$
```

The subshell running as process 9830 is a child of process 3662, the original ksh shell. `ps` and `grep` are the children of process 9830 (ksh `psit`). When the `psit` script is done and exits, the subshell exits, and control is returned to the original shell.

You can also start a subshell by entering the shell name (ksh for Korn, sh for Bourne, and csh for C Shell). This feature is handy if you have one login (default) shell and want to use another. Starting out in Korn Shell and starting C Shell would look like the following:

```
$ csh
% ps -f
      UID  PID  PPID  C   STIME TTY      TIME CMD
dhorvath 3662  3657  0 08:10:53 pts/6   0:00 -ksh
dhorvath 3266  8848 11 10:50:40 pts/6   0:00 ps -f
dhorvath 8848  3662  1 10:50:38 pts/6   0:00 csh
%
```

The C shell uses the percent sign as a prompt. After the `csh` command starts the shell, the prompt becomes the percent sign. The `ps` command shows `csh` as a child process and subshell of ksh. To exit `csh` and return to the parent shell, you can use the `exit` command.

Process Permissions

By default, a process runs with the permissions of the user running it. In most cases, this makes sense, enabling you to run a command or utility only on your files. There are times, however, when users need to access files that they do not own. A good example of this is the `passwd` command, which is usually stored as `/usr/bin/passwd`. It is used to change passwords and modify `/etc/passwd` and the shadow password file, if the system is so equipped.

It does not make sense for general users to have write access to the password files; they could create users on-the-fly. The program itself has these permissions. If you look at the file using `ls`, you see the letter `s` where `x` normally appears in the owner and group permissions. The owner of `/usr/bin/passwd` is `root`, and it belongs to the `sys` group. No matter who runs it, it has the permissions of the root user.

Overlaying the Current Process (exec Command)

In addition to creating (forking) child processes, you can overlay the current process with another. The `exec` command replaces the current process with the new one. Use this command only with great caution. If you use `exec` in your primary (login) shell interpreter, that shell interpreter (ksh with pid 3662 in the previous examples) is replaced with the new process. Using the command `exec ls` at your login shell prompt gives you a directory listing and then disconnects you from the system, logging you out. Because `exec` overlays your shell (ksh, for example), there are no programs to handle commands for you when `ls` finishes and exits.

You can use `exec` to change your shell interpreter completely without creating a subshell. To convert from `ksh` to `csch`, you can use the following:

```
$ exec csh
% ps -f
      UID  PID  PPID  C   STIME TTY      TIME CMD
dhorvath 3662  3657  0  08:10:53 pts/6    0:00 csh
dhorvath 3266  3662 11 14:50:40 pts/6    0:00 ps -f
%
```

The prompt changes and `ps` shows `csch` instead of `ksh` but with the original pid and start time.

Summary

In this chapter, you looked at the four major topics involving processes provided with the shell:

- Starting a process
- Listing running processes
- Killing a process (`kill` command)
- Manipulating parent and child processes

As you write scripts and use the shell, knowing how to work with processes improves your productivity.

Questions

1. How do you run a command in the background?
2. How do you determine which processes you are running?
3. How do you change a foreground process into a background process?

Terms

Background Describes processes usually running at a lower priority and with their input disconnected from the interactive session. Input and output are usually directed to a file or other process.

Background processes Autonomous processes that run under UNIX without requiring user interaction.

Child processes See *subprocesses*.

Child shells See *subshells*.

Parent process identifier Shown in the heading of the `ps` command as `PPID`. This is the process identifier of the parent process. See also *parent processes*.

Parent processes These processes control other processes that are often referred to as child processes or subprocesses. See *processes*.

Parent shell This shell controls other shells, which are often referred to as child shells or subshells. The login shell is typically the parent shell.

Process identifier Shown in the heading of the `ps` command as `pid`. It is the unique number assigned to every process running in the system.

Processes Discrete, running programs under UNIX. The user's interactive session is a process. A process can invoke (run) and control another program that is then referred to as a subprocess. Ultimately, everything a user does is a subprocess of the operating system.

Subprocesses Run under the control of other processes, which are often referred to as parent processes. See *processes*.

Subshells Run under the control of another shell, which is often referred to as the parent shell. Typically, the login shell is the parent shell.



PART II

Shell Programming

Hour

- 8 Variables
- 9 Substitution
- 10 Quoting
- 11 Flow Control
- 12 Loops
- 13 Parameters
- 14 Functions
- 15 Text Filters
- 16 Filtering Text with Regular Expressions
- 17 Filtering Text with `awk`
- 18 Other Tools

This page intentionally left blank

HOUR 8



Variables

Variables are words that hold a *value*. The value can be any text string. The shell enables you to create, assign, and delete variables. Although the shell manages some variables, it is mostly up to the programmer to manage variables in shell scripts. By using variables, you can make your scripts flexible and maintainable.

In this chapter, we will examine the following topics:

- Creating variables
- Accessing variables
- Array variables
- Deleting variables
- Environment variables

Working with Variables

Two types of variables can be used in shell programming:

- Scalar variables
- Array variables

Scalar variables can hold only one value at a time. Array variables can hold multiple values. This section explores the use of both types of variables.

Scalar Variables

Scalar variables are defined as follows:

```
name=value
```

Here *name* is the name of the variable, and *value* is the value that the variable should hold. For example,

```
FRUIT=peach
```

defines the variable FRUIT and assigns it the value peach.



Scalar variables are often referred to as *name-value pairs*, because a variable's name and its value can be thought of as a pair.

Variable Names

The name of a variable can only contain letters (a to z or A to Z), numbers (0 to 9), and the underscore character (_). Furthermore, a variable's name can only start with a letter or an underscore. The following are examples of valid variable names:

```
_FRUIT  
FRUIT_BASKET  
TRUST_NO_1  
TWO_TIMES_2
```

but

```
2_TIMES_2_EQUALS_4
```

is not a valid variable name. To make this a valid name, we would need to add an underscore at the beginning of its name:

```
_2_TIMES_2
```

Variable names that start with numbers, such as 1, 2, or 11, are reserved for use by the shell. You can use the value stored in these variables, but you cannot set the value yourself.

The reason you cannot use other characters such as `!`, `*`, or `-` is that these characters have a special meaning for the shell. If you try to create a variable name with one of these special characters, it confuses the shell. For example, the variable names

```
FRUIT-BASKET
_2*2
TRUST_NO_1!
```

are invalid names. The error message generated for the first variable name will be similar to the following:

```
$ FRUIT-BASKET=apple
/bin/sh: FRUIT-BASKET=apple: not found.
```

Variable Values

You can store or assign any value you want in a variable. For example,

```
FRUIT=peach
FRUIT=2apples
FRUIT=apple+pear+kiwi
```

A common error with variables is assigning values that contain spaces. For example, the following assignment

```
$ FRUIT=apple orange plum
```

results in this error message:

```
sh: orange: not found.
```

Values that have spaces in them need to be *quoted*. For example, both of the following are valid assignments:

```
$ FRUIT="apple orange plum"
$ FRUIT='apple orange plum'
```

The difference between these two quoting schemes is covered in Chapter 10, “Quoting.”

Accessing Values

You can access the value stored in a variable by prefixing its name with the dollar sign (`$`). When the shell sees a `$`, it performs the following actions:

1. Reads the next word to determine the *name* of the variable.
2. Retrieves the *value* for the variable. If a value isn't found, the shell uses the empty string `" "` as the value.
3. Replaces the `$` and the *name* of the variable with the *value* of the variable.

This process, known as *variable substitution*, is covered in greater detail in Chapter 9, “Substitution.” The following example demonstrates this process:

```
$ FRUIT=peach
$ echo $FRUIT
peach
```

In this example, the shell first determines that the variable `FRUIT` has been referenced. Next it looks up the value for `FRUIT`. Finally the string `$FRUIT` is replaced with `peach`, the value of `FRUIT`, which is what the `echo` command prints.

If you do not use the dollar sign (`$`), variable substitution is not performed and the name of the variable is used directly. For example,

```
$ echo FRUIT
FRUIT
```

simply prints out `FRUIT`, not the value of the variable `FRUIT`.

The dollar sign (`$`) is used only when accessing a variable’s value. It should not be used to define a variable or assign a value to a variable. For example, the assignment

```
$ $FRUIT=apple
```

generates the following error message

```
sh: peach=apple: not found
```

assuming that the value of `FRUIT` was `peach`. If the variable `FRUIT` did not have a value, the error would have been

```
sh: =apple: not found
```

Array Variables

Arrays are a method for grouping a set of variables together using a single name. Instead of creating a new name for each variable you need, you can use a single array variable to store all the variables.

To understand how arrays work, consider the following example. Say that we are trying to represent the chapters in this book using a set of scalar variables. We could choose the following variable names to represent some of the chapters:

```
CH01
CH02
CH15
CH07
```

Each of these variable names has a specific format: the letters CH followed by the chapter number. This format serves as a way of grouping these variables together. An array variable formalizes this grouping by using an array name in conjunction with a number known as an *index*. The index is used to locate entries or elements in the array.



Arrays are not available in Bourne shell. Arrays first appeared in Korn Shell, ksh, and were adapted by the Z Shell, zsh. Recent versions (2.0 and newer) of the Bourne Again Shell, bash, include support for arrays, but older versions do not. Several Linux distributions still ship with the older version of bash.

If you are using bash, the following command allows you to determine its version:

```
$ echo $BASH_VERSION
```

If the output starts with the string '1.' as follows:

```
1.14.7(1)
```

the version of bash you are using does not support arrays. The examples in this section will not work 1.x versions of bash.

If the output starts with the string '2.' as follows:

```
2.03.0(1) -release
```

the version of bash you are using supports arrays. The examples in this section will work with 2.0 and newer versions of bash.

Creating Array Variables

The simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:

```
name[index]=value
```

Here *name* is the name of the array, *index* is the index of the item in the array that you want to set, and *value* is the value you want to set for that item. In ksh, *index* must be an integer between 0 and 1,023. No such restriction is present in bash or zsh. The only restriction is that *index* must be an integer. It cannot be a floating point or decimal number, such as 10.3, or a string, such as apricot.

As an example, the following commands

```
$ FRUIT[0]=apple  
$ FRUIT[1]=banana  
$ FRUIT[2]=orange
```

set the values of the first three items in the array named `FRUIT`. You could do the same thing with scalar variables as follows:

```
$ FRUIT_0=apple
$ FRUIT_1=banana
$ FRUIT_2=orange
```

Although this works fine for small numbers of items, the array notation is much more efficient for large numbers of items. If you have to write a script using the Bourne shell only, you can use this method for simulating arrays.

In the previous example, the array indices were set in sequence. This is not necessary. For example, the following command sets the value of the item at index `10` in the `FRUIT` array:

```
$ FRUIT[10]=plum
```

The shell does not create a bunch of blank array items to fill in the space between index `2` and index `10`; it just keeps track of those array indices that contain values.

If an array variable with the same name as a scalar variable is defined, the value of the scalar variable becomes the value of the element of the array at index `0`. For example, if the following commands are executed

```
$ FRUIT=apple
$ FRUIT[1]=peach
```

the zeroth element of `FRUIT` has the value `apple`. At this point, any accesses to the scalar variable `FRUIT` are treated as an access to the array item `FRUIT[0]`.

The second form of array initialization can be used to set multiple elements at once. The syntax for this form of initialization differs between `ksh` and `bash`. In `ksh`, the syntax is as follows:

```
set -A name value1 value2 ... valueN
```

In `bash`, the syntax is

```
name=(value1 ... valueN)
```

Either style can be used in `zsh`. Regardless of the style, *name* is the name of the array, and *value1* to *valueN* are the values of the items to be set. When setting multiple elements at once, consecutive array indices, beginning at `0`, are used.

For example the `ksh` command

```
$ set -A band derri terry mike gene
```

or the `bash` command

```
$ band=(derri terry mike gene)
```

is equivalent to the following commands:

```
$ band[0]=derri
$ band[1]=terry
$ band[2]=mike
$ band[3]=gene
```



When setting multiple array elements in bash, you can place an array index before the value:

```
myarray=( [0]=derri [3]=gene [2]=mike [1]=terry )
```

The array indices don't have to be in order.

Accessing Array Values

An array variable can be accessed as follows:

```
${name[index]}
```

Here *name* is the name of the array, and *index* is the index of the desired element. For example, if the array `FRUIT` was initialized as in previous examples, the command

```
$ echo ${FRUIT[2]}
```

produces the following output:

```
orange
```

You can access all the items in an array in one of the following methods:

```
${name[*]}
${name[@]}
```

Here *name* is the name of the array you are interested in. If the `FRUIT` array is initialized as in previous examples, the command

```
$ echo ${FRUIT[*]}
```

produces the following output:

```
apple banana orange
```

If values of any of the array items contain spaces, this form of array access will not work; you will need to use the second form. The second form quotes all the array entries so that embedded spaces are preserved. For example, define the following array item:

```
FRUIT[3]="passion fruit"
```


Assuming that `FRUIT` is defined as in previous examples, accessing the entire array using the following command

```
$ echo ${FRUIT[*]}
```

results in five items, not four:

```
apple banana orange passion fruit
```

Commands accessing `FRUIT` using this form of array access get five values, with `passion` and `fruit` treated as separate items. To get only four items, you have to use the following form:

```
$ echo ${FRUIT[@]}
```

The output from this command looks similar to the previous commands:

```
apple banana orange passion fruit
```

but commands will see only four items because the shell quotes the last item, `passion fruit`, so it is treated as a single item.

Read-Only Variables

A *read-only variable* is a variable whose value cannot be changed after it is defined.

Once a variable is specified as read-only, there is no way to get rid of it or to modify its value; it and its value persist until the shell exits.

Variables can be marked read-only using the `readonly` command. Consider the following set of commands:

```
$ FRUIT=kiwi
$ readonly FRUIT
$ echo $FRUIT
kiwi
$ FRUIT=cantaloupe
```

The last command results in an error message similar to the following:

```
/bin/sh: FRUIT: This variable is read only.
```

As you can see, we can read the value of the variable `FRUIT`, but we cannot overwrite the value stored in it.

This feature is often used in scripts to make sure that critical variables are not overwritten accidentally.

In `ksh`, `bash`, and `zsh`, `readonly` can be used to mark both array and scalar variables as read-only:

```
$ FRUITBASKET=(apple orange pear)
$ readonly FRUITBASKET
```

```
$ echo ${FRUITBASKET[1]}
orange
$ FRUITBASKET[1]=kiwi
```

The last command results in an error message similar to the following:

```
sh: FRUITBASKET[1]: is read only
```

This example used the bash style array assignment; if you are using ksh you will need to change the first command to the following:

```
$ set -A FRUITBASKET apple orange pear
```

Unsetting Variables

Unsetting a variable tells the shell to remove the variable from the list of variables that it tracks. This is like asking the shell to forget a piece of information because it is no longer required.

Both scalar and array variables can be unset using the `unset` command:

```
unset name
```

Here *name* is the name of the variable to unset. For example, the following command unsets the variable `FRUIT`:

```
unset FRUIT
```

The `unset` command cannot be used to unset variables that have been marked read-only via `readonly`. There is no way to unset a read-only variable; it persists until the shell exits.

Environment and Shell Variables

When the shell starts a program, it passes that program a set of variables called the *environment*. The environment is usually a small subset of the variables defined in the shell. Each variable in the environment is called an *environment variable*.

The variables we have examined thus far have been local variables. *Local variables* are variables whose value is restricted to a single shell. Local variables are not passed to programs started by the shell.

In addition to local variables and environment variables, there is a third category of variables called *shell variables*. These are special variables set by the shell that are required for proper operation of the shell. Some shell variables are environment variables, whereas others are local variables.

Table 8.1 compares these three categories of variables.

TABLE 8.1 A Comparison of Local, Environment, and Shell Variables

<i>Attribute</i>	<i>Local</i>	<i>Environment</i>	<i>Shell</i>
Accessible by child processes	No	Yes	Yes
Set by users	Yes	Yes	No
Set by the shell	No	No	Yes
User modifiable	Yes	Yes	No
Required by the shell	No	No	Yes

Exporting Environment Variables

Environment variables are just local variables that have been placed into the environment via the `export` command:

```
export name
```

The variable specified by *name* is placed in the environment. The process of placing variables into the environment is often referred to as *exporting* the variable. The standard shell idiom for exporting variables is

```
name=value ; export name
```

An example of this is

```
PATH=/sbin:/bin ; export PATH
```

Here a value is assigned to `PATH`, and then `PATH` is exported. Often, the assignment statement of an environment variable and the corresponding `export` statement are written on one line to clarify that the variable is an environment variable. This helps the next programmer, who has to maintain the script, quickly grasp the use of certain variables.

A single `export` command can be used to export more than one variable. For example, the command

```
export PATH HOME UID
```

exports the variables `PATH`, `HOME`, and `UID` to the environment.

Exporting Variables in `ksh`, `bash`, and `zsh`

An alternative form for exporting variables is available in `ksh`, `bash`, and `zsh`:

```
export name=value
```

In this form, the variable specified by *name* is assigned the specified *value* and then that variable is marked for export. In this form command,

```
export PATH=/sbin:/bin
```

is equivalent to

```
PATH=/sbin:/bin ; export PATH
```

In this form, any combination of *name* or *name=value* pairs can be given to the `export` command. For example, the command

```
export FMHOME=/usr/frame CLEARHOME=/usr/atria PATH
```

assigns the specified values to the variables `FMHOME` and `CLEARHOME` and then exports the variables `FMHOME`, `CLEARHOME`, and `PATH`.

Shell Variables

Shell variables are variables that the shell sets during initialization and uses internally. Table 8.2 gives a list of the most common shell variables. Some other shell variables are covered in the section “Variable Substitution” in Chapter 9.

TABLE 8.2 Shell Variables

<i>Variable</i>	<i>Description</i>
<code>PWD</code>	Indicates the current working directory as set by the <code>cd</code> command.
<code>UID</code>	Expands to the numeric user ID of the current user, initialized at shell startup.
<code>SHLVL</code>	Increments by one each time an instance of <code>bash</code> is started. This variable is useful for determining whether the built-in <code>exit</code> command ends the current session.
<code>REPLY</code>	Expands to the last input line read by the <code>read</code> built-in command when it is given no arguments. This variable is not available in Bourne shell.
<code>RANDOM</code>	Generates a random integer between 0 and 32,767 each time it is referenced. You can initialize the sequence of random numbers by assigning a value to <code>\$RANDOM</code> . If <code>\$RANDOM</code> is unset, it loses its special properties, even if it is subsequently reset. This variable is not available in Bourne shell.
<code>SECONDS</code>	Each time this parameter is referenced, it returns the number of seconds since shell invocation. If a value is assigned to <code>\$SECONDS</code> , the value returned on subsequent references is the number of seconds since the assignment plus the value assigned. If <code>\$SECONDS</code> is unset, it loses its special properties, even if it is subsequently reset. This variable is not available in Bourne shell.
<code>IFS</code>	Indicates the Internal Field Separator that is used by the parser for word splitting after expansion. <code>\$IFS</code> is also used to split lines into words with the <code>read</code> built-in command. The default value is the string, <code>\t\n</code> , where <code> </code> is the space character, <code>\t</code> is the tab character, and <code>\n</code> is the new-line character.

TABLE 8.2 continued

<i>Variable</i>	<i>Description</i>
PATH	Indicates the search path for commands. It is a colon-separated list of directories in which the shell looks for commands. A common value is PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/ucb
HOME	Indicates the home directory of the current user: the default argument for the cd built-in command.

Summary

This chapter covered using variables for shell script programming. You saw how scalar and array variables were defined, accessed, and unset. We also looked at a special category of variables known as environment variables. In subsequent chapters, we will look at how variables are used to achieve a greater degree of flexibility and clarity in shell scripts.

Questions

- Which of the following are valid variable names?
 - _FRUIT_BASKET
 - 1_APPLE_A_DAY
 - FOUR-SCORE&7YEARS_AGO
 - Variable
- Is the following sequence of array assignments valid in sh, ksh, and bash?


```
$ adams[0]=hitchhikers_guide
$ adams[1]=restaurant
$ adams[3]=thanks_for_all_the_fish
$ adams[42]=life_universe_everything
$ adams[5]=mostly_harmless
```
- Given the preceding array assignments, how would you access the array item at index 5 in the array adams? How would you access every item in the array?
- What is the difference between an environment variable and a local variable?

Terms

Array Variable An array variable is a variable that groups multiple scalar variables together using a single name. Each of the individual scalar variables is accessed via an index.

Environment The environment is a set of variables that the shell passes to every program it starts.

Environment Variable An environment variable is a variable that is a member of the environment.

Exporting The process of placing a variable in the environment is called exporting.

Local Variable A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell.

Read-Only Variable A read-only variable is a variable whose value cannot be changed.

Scalar Variable A scalar variable can hold only one value at a time.

Shell Variable A shell variable is a variable that is set by the shell and is required by the shell to function correctly.

Unsetting Unsetting a variable removes it from the list of variables tracked by the shell.

Variable A variable is a word that holds a value. The value can be any text string.

Variable Substitution Variable substitution is the process by which the shell replaces the name of a variable with its value.

This page intentionally left blank

HOUR 9



Substitution

When the shell encounters an expression that contains one or more *meta-characters*, it performs *substitutions* on that expression. Meta-characters are characters that have a special meaning in the shell. Substitution is the process by which the shell converts a string containing meta-characters into a different string that is the result of interpreting the meta-characters. In the last chapter, you saw how the \$ meta-character can be used to access a variable's value in a process known as *variable substitution*. In addition to variable substitution, the shell can also perform several other types of substitutions. This chapter looks at each of these types of substitution and their associated meta-characters in detail. Specifically the topics covered are

- Filename substitution
- Value-based variable substitution
- Command substitution
- Arithmetic substitution

Filename Substitution (Globbing)

The most common type of substitution is *filename substitution* or *globbing*. Globbing is the method by which the shell expands a string containing globbing meta-characters or *wildcards* into a list of filenames. Table 9.1 lists the wildcards used in globbing.

TABLE 9.1 Globbing Meta-Characters (Wildcards)

<i>Wildcard</i>	<i>Description</i>
*	Matches zero or more occurrences of any character
?	Matches one occurrence of any character
[<i>characters</i>]	Matches one occurrence of any of the given <i>characters</i>

Any command or script that operates on files can take advantage of globbing. The examples in this section use the `ls` command because its output clearly illustrates the results of globbing.

The * Meta-Character

The simplest form of filename substitution is the asterisk or star, `*`, meta-character. The `*` matches zero or more occurrences of any character in a filename.

When given by itself, the `*` matches all visible filenames in the current directory. For example, the command

```
$ ls *
```

lists every file and the contents of every directory in the current directory. Invisible files or directories are not listed.

Although the `*` is sometimes used by itself, its main use is in matching file prefixes and suffixes.

Matching a Prefix

To match a file prefix, the `*` can be used as follows:

```
cmd prefix*
```

Here *cmd* is the name of a command, such as `ls`, and *prefix* is the filename prefix you want to match. For example, the following command lists all the files and directories in the current directory that start with the letters CGI:

```
$ ls CGI*
CGI.java CGIGet.java CGIGetTest.java CGIPost.java CGIPostTest.java
```

By varying the prefix slightly, you can alter the list of files that are matched. For example, the command

```
$ ls CGIG*
```

generates the following list of files:

```
CGIGet.java      CGIGetTest.java
```

Varying the suffix allows you to manipulate the list of the matched filenames until the list contains just the filenames you are interested in.

Matching a Suffix

To match a file suffix, the `*` can be used as follows:

```
cmd *suffix
```

Here *cmd* is the name of a command, such as `ls`, and *suffix* is the filename suffix you want to match. For example, the following command lists all the files and directories in the current directory that end with the letters `java`:

```
$ ls *java
CGI.java CGIGet.java CGIGetTest.java CGIPost.java CGIPostTest.java
```

By varying the suffix slightly, you can alter the list of files that are matched. To list just the files that end with `Test.java`, you can adjust the command as follows:

```
$ ls *Test.java
CGIGetTest.java CGIPostTest.java
```

Varying the suffix also allows you to manipulate the list of the matched filenames in order to obtain a list of the filenames that interest you.

Matching Prefixes and Suffixes

You can match both the prefix and the suffix by using the `*` character as follows:

```
cmd prefix*suffix
```

Here *cmd* is the name of a command, such as `ls`, *prefix* is the filename prefix, and *suffix* is the filename suffix. For example, the following command lists all the files and directories in the current directory with the prefix `CGIG` and the suffix `java`:

```
$ ls CGIG*java
CGIGet.java      CGIGetTest.java
```

It is also possible to use multiple `*` in a filename substitution expression. For example, if you needed to list only those files with the prefix `CGI`, the suffix `java`, and that contain the characters `st`, you could use the following command:

```
$ ls CGI*st*java
```

The output is as follows:

```
CGIGetTest.java  CGIPost.java  CGIPostTest.java
```

Globbering is Case Sensitive

When using the *, it is important to specify the correct case for the prefix and suffix. For example, the command `ls CGI*` produces the following output:

```
CGI.java CGIGet.java CGIGetTest.java CGIPost.java CGIPostTest.java
```

whereas the command `$ ls cgi*` does not produce the same list of files.

The ? Meta-Character

One of the limitations of the * is that it matches zero or more characters. Consider a situation where you need to list all files that have names of the form `ch0X.doc`, where `X` is a single number or letter. At first glance it seems like the command

```
$ ls ch0*.doc
```

would produce the appropriate list, but inspecting the output shows otherwise:

```
ch01.doc ch01-1.doc ch01-2.doc ch02.doc ch02-1.doc ch02-2.doc  
ch03.doc ch03-1.doc ch03-2.doc
```

In order to match only one character, you need to use the question meta-character. The ? matches exactly one instance of a character. Rewriting the previous example using the ? yields:

```
$ ls ch0?.doc
```

Now the output contains only those files you were interested in:

```
ch01.doc ch02.doc ch03.doc
```

Say that you need to look for all files that have names of the form `chXY`, where `X` and `Y` are any number or character. You can use two ? meta-characters in order to obtain the desired list of files:

```
$ ls ch??.doc  
ch01.doc ch02.doc ch03.doc
```

Common Errors

If the shell cannot find any files that match an expression containing a ?, the shell treats the ? as a regular character. Because most filenames do not include a ?, this usually produces an error message. For example, the following command:

```
$ ls ch?.doc
```

produces the error message:

```
ls: ch?.doc: No such file or directory
```

For this reason, a shell script needs to validate the existence of files that are specified as arguments. The procedure for performing such checks is discussed in Chapter 11, “Flow Control.”

Matching Sets of Characters

Two potential problems with the `?` and `*` wildcards are

- Any character, including special characters such as hyphens (`-`) or underscores (`_`), is matched by these characters.
- There is no way to match only letters or only numbers.

Sometimes you need more control over the characters to be matched. Consider the situation where you need to match filenames of the form `ch0X`, where `X` is a number between 0 and 9. Neither the `*` nor the `?` operator is appropriate for this task.

In order to match sets of characters, you need to use the `[` and `]` meta-characters. The syntax for using these meta-characters is as follows:

```
cmd [chars]
```

Here *cmd* is the name of a command, such as `ls`, and *chars* is the set of characters to match. For example, the following command fulfills these requirements:

```
$ ls ch0[0123456789].doc
ch01.doc ch02.doc ch03.doc
```

Character Ranges

In the previous example, the set contained an explicit list of all the characters that you wanted to match. This can be cumbersome if you need to deal with large sets of characters. You can simplify this by specifying a *character range* with the `-` meta-character. A character range is a method for specifying a set of characters by providing the first and last character in the set. For example, the character range `0-9` specifies all the numbers between zero and nine, inclusive.

Using the range `0-9`, you can rewrite the previous example as follows:

```
$ ls ch0[0-9].doc
ch01.doc ch02.doc ch03.doc
```

Character ranges are most useful when trying to match sets of letters. For example,

```
$ ls [a-z]*
```

lists all the files starting with a lowercase letter. To match all the files starting with uppercase letters, use the following:

```
$ ls [A-Z]*
```

You can also combine multiple character ranges in a single set. For example,

```
$ ls [a-zA-Z]*
```

matches all files that start with a letter, whereas the command

```
$ ls *[a-zA-Z0-9]
```

matches all files ending with a letter or a number.

Coupling sets with other meta-characters gives you the maximum amount of flexibility in filename substitution.

Negating a Set

Consider a situation where you need a list of all files except those that contain a particular letter, for example, the letter a. You can solve this problem in two ways:

- Specify all the characters you want a filename to contain.
- Specify that the filename not include the letter a.

If you choose the first approach, you need to construct a set of all the characters that your filename can contain. You can start with:

```
[b-zA-Z0-9]
```

This set does not include the special characters that are allowed in filenames. Attempting to include all these characters creates a cumbersome set with complicated quoting:

```
[b-zA-Z0-9\ \_ \+ \= \\ \\' \" \{ \} \]
```

Compared to this, the second approach seems much simpler, because all you need to do is specify the set of characters to exclude. This is accomplished using the ! operator. When ! is the first character in a set, the shell matches only those filenames that do not include the characters in the set that follows the !. The syntax for this operator is:

```
cmd [!chars]
```

Here, *cmd* is the name of a command, such as `ls`, and *chars* is the set of characters that should not be matched.

As an example, you can list all files except those starting with the letter a using the command

```
$ ls [!a]*
```

Variable Substitution

In the previous chapter, you learned about a basic form of variable substitution, namely how to retrieve the value of a variable using the `$` meta-character. In addition to this, the shell provides several other advanced forms of variable substitution that enable shell programs to manipulate the value of a variable based on its state.

There are two broad categories of advanced variable substitution:

- Actions taken when a variable has a value
- Actions taken when a variable does not have a value

The actions can range from one time value substitution to aborting the script. These categories are broken into four forms of variable substitution. These forms are summarized in Table 9.2.

TABLE 9.2 Advanced Variable Substitution

<i>Name</i>	<i>Syntax</i>	<i>Description</i>
Default Value Substitution	<code>\${param:-word}</code>	If <i>param</i> is null or unset, <i>word</i> is substituted for <i>param</i> . The value of <i>param</i> does not change.
Default Value Assignment	<code>\${param:=word}</code>	If <i>param</i> is null or unset, <i>param</i> is set to the value of <i>word</i> .
Null Value Error	<code>\${param:?msg}</code>	If <i>param</i> is null or unset, <i>msg</i> is printed to STDERR and the shell exits.
Substitute When Set	<code>\${param:+word}</code>	If <i>param</i> is set, <i>word</i> is used instead of the value of <i>param</i> . The value of <i>param</i> does not change.

Default Value Substitution

The first form of advanced variable substitution allows a default value to be substituted when the variable's value is null. The syntax is as follows:

```
${param:-word}
```

Here *param* is the name of the variable and *word* is the default value. Substitution is performed only when *param* is unset. Furthermore, *word* is not assigned to *param*; the shell just replaces the expression with *word*. The following example illustrates the behavior:

```
$ unset MYFRUIT
$ FRUIT=${MYFRUIT:-APPLE}
$ echo MYFRUIT is $MYFRUIT, FRUIT is $FRUIT
MYFRUIT is , FRUIT is APPLE
```

Default Value Assignment

The second form of advanced variable substitution assigns a value to a variable when the variable's value is null. The syntax is as follows:

```
`${param}:=word`
```

Here *param* is the name of the variable and *word* is the value to assign if the variable's value is null. The following example illustrates the behavior:

```
$ unset FRUIT
$ echo FRUIT is $FRUIT
FRUIT is
$ echo FRUIT is `${FRUIT}:=APPLE`
FRUIT is APPLE
```

Null Value Error

Sometimes substituting or assigning default values can hide problems in a shell script. In order to spot such problems in critical parts of a shell script, you can use the third form of variable substitution that outputs a message to `STDERR` when a variable is unset. The syntax is as follows:

```
`${param}?:msg`
```

Here, *param* is the name of the variable and *msg* is the message to be printed to `STDERR`.

If a shell script or shell function requires a certain variable to be set for proper execution, this form of variable substitution can be used. For example, the following expression causes the shell to exit if the variable `$HOME` is unset:

```
: `${HOME}?:"Your home directory is undefined."
```

In addition to using the variable substitution form described previously, this example makes use of the `no-op` (short for no operation) command, `:`. This command performs no work; it just evaluates the arguments passed to it.

Substitute When Set

The final form of variable substitution is used to substitute a value when a variable is set. The syntax is as follows:

```
`${param}+word`
```

Here *param* is the name of the variable and *word* is the value to substitute if the variable is set. If *param* is unset, then nothing is substituted. This form does not alter the value of the variable. It is commonly used by scripts to indicate that the script is running in debug mode:

```
echo `${DEBUG}+: "Debug is active."`
```

Command and Arithmetic Substitution

Two additional forms of substitution are *command* and *arithmetic substitution*. Command substitution enables you to capture the output of a command, whereas arithmetic substitution enables you to perform basic integer math using the shell.

Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands. Command substitution is performed when a command is given as

```
`command`
```

Here *command* can be a simple command, a pipeline, or a list.



Make sure that you are using the backquote character, not the single quote character, when performing command substitution. Command substitution is performed by the shell only when the backquote, or backtick, character, ```, is given. Using the single quote instead of the back quote is a common error and leads to hard-to-find bugs.

Command substitution is generally used to assign the output of a command to a variable as the following examples demonstrate:

```
DATE=`date`  
USERS=`who | wc -l`  
UP=`date ; uptime`
```

In the first example, the output of the `date` command becomes the value for the variable `DATE`. In the second example, the output of the pipeline becomes the value of the variable `USERS`. In the last example, the output of the list becomes the value of the variable `UP`.

You can also use command substitution to provide arguments for other commands. For example,

```
grep `id -un` /etc/passwd
```

looks through the file `/etc/passwd` for the output of the command:

```
id -un
```

The output of this command will be the entry in `/etc/passwd` corresponding to the current user, for example:

```
ranga:*:500:500:Sriranga Veeraraghavan:/home/ranga:/bin/ksh
```




Some system administrators have special scripts that track and report users who access the password file, `/etc/passwd`. Before you execute commands that access this file, please check with your system administrator to ensure that you are not violating site policy.

Arithmetic Substitution

Arithmetic substitution allows you to perform simple integer math using the shell. It was first introduced in `ksh` and has been incorporated into `bash` and `zsh`. It is not available in the Bourne shell. Scripts that use the Bourne shell have to use an external program such as `expr` or `bc` (covered in Chapter 18, “Other Tools”) to perform basic interger math.

Arithmetic substitution is performed when an expression of the following form is encountered:

```
$( (exp) )
```

Here *exp* is a mathematical expression constructed using the operators given in Table 9.3. Standard precedence rules are used for evaluating *exp*.

TABLE 9.3 Arithmetic Substitution Operators

<i>Operator</i>	<i>Description</i>
/	The division operator. Divides two numbers and returns the result.
*	The multiplication operator. Multiplies two numbers and returns the result.
-	The subtraction operator. Subtracts two numbers and returns the result.
+	The addition operator. Adds two numbers and returns the result.
()	The parentheses clarify which expressions should be evaluated before others.

If *exp* does not evaluate to an integer (whole number), the value of *exp* is truncated. As an illustration, consider the following command:

```
$ echo $( ( 5/2 ) )
2
```

The result of the division is 2.5, but in integer math the .5 is ignored and the truncated result, 2, is returned. The result isn't rounded; everything that follows the decimal point is discarded.

Precedence Example

The following example illustrates the rules of precedence:

```
$ echo $(( (5 + 3*2) - 4) / 2 )
3
```

If you are having trouble understanding the output, just break down the operations starting with the sub-expression contained in the innermost parenthesis:

1. $(5 + 3*2)$. Because $*$ has higher precedence than $+$, this sub-expression evaluates to 11.
2. Substituting the result from Step 1 yields the sub-expression $(11 - 4)$. This evaluates to 7.
3. Substituting the result from Step 2 yields the sub-expression $7 / 2$. This evaluates to 3.5, which is truncated to 3.

Common Errors

A common error in arithmetic substitution is inserting spaces between the parentheses. There should be no spaces between the first or last set of parentheses. The correct syntax is as follows:

```
$(( exp ))
```

If a space is inserted between the parentheses, as follows:

```
$(( exp ))
$( ( exp ))
$ ( exp )
```

the shell will generate an error message. The exact error message depends on *exp*. For example, all the following commands:

```
$ echo $(( 5/2 ))
$ echo $( ( 5/2 ))
$ echo $ ( 5/2 )
```

will produce an error message similar to the following:

```
sh: command not found: 5
```

On some systems the error message is

```
sh: no such file or directory: 5
```

Summary

In this chapter, you looked at four forms of substitution available in the shell:

- Filename substitution
- Variable substitution
- Command substitution
- Arithmetic substitution

As you write scripts and use the shell to solve problems, these types of substitution will be of immense utility.

Questions

1. What combination of wildcards should you use to list all the files in the current directory that end in the form `hwXYZ.ABC?`

Here *X* and *Y* can be any number; *Z* is a number between 2 and 6; and *A*, *B*, and *C* are characters.

2. What action is performed by the following line, if the variable `MYPATH` is unset:

```
: ${MYPATH:=/usr/bin:/usr/sbin:/usr/ucb}
```

3. What is the difference between the actions performed by the command given in the previous problem and the action performed by the following command:

```
: ${MYPATH:-/usr/bin:/usr/sbin:/usr/ucb}
```

4. What is the output of the following command (figure it out by yourself before typing it into the shell):

```
echo $(( 3 * 2 + ( 4 - 3 / 4 ) ))
```

Terms

Character range A method for specifying a set of characters by giving the first and last character in the set.

Globbing The process used by the shell to produce a list of files that match a particular expression. Also known as filename substitution.

Meta-characters Characters that have a special meaning in the shell.

Substitution The process by which the shell converts a string containing meta-characters into a different string that is the result of interpreting the meta-characters.

Wildcards Meta-characters used in globbing. The two main wildcards are `*` and `?`.

HOUR 10



Quoting

In the preceding chapter, you looked at substitution, which occurs automatically whenever you enter a command containing a meta-character or a \$. The way the shell interprets these and other special characters is generally useful, but sometimes it is necessary to turn off shell substitution and let each character stand for itself. Turning off the special meaning of a character is called *quoting*, and it can be done in three ways:

- Using the backslash (\)
- Using the single quote (')
- Using the double quote (")

Quoting can be a very complex issue, even for experienced UNIX programmers. In this chapter, you look at each of these forms of quoting and learn how to use them. You learn a series of simple rules to help you understand when quoting is needed and how to do it correctly.

Quoting with Backslashes

To start out, let's use `echo` to get a better idea about how the shell treats special characters. For example,

```
$ echo Hello world
```

displays the following message on your screen:

```
Hello world
```

Watch what happens if you add the semicolon (;) meta-character in between `Hello` and `world`:

```
$ echo Hello; world
Hello
sh: world: Command not found
```

The semicolon (;) character tells the shell that it has reached the end of one command and what follows is a new command. This character enables multiple commands on one line. Because `world` is not a valid command, you get an error message (the error message on your system might be slightly different).

In order to display a meta-character, you need to *quote* it. When a character is quoted, its special meaning is disabled. In the shell, characters are quoted using the backslash (\) character. As an example, you can resolve the problem in the previous example by quoting the semicolon as follows:

```
$ echo Hello\; world
Hello; world
```

As you can see, the quoting character (\) is not displayed in the output. The shell pre-processes the command line, performing variable substitution, command substitution, and filename substitution, unless the special character that would normally invoke substitution is quoted. The backslash is then removed from the command arguments, so the command being run never sees the quoting character.



The technique of quoting a meta-character with the backslash is frequently referred to as *escaping*. The terms quoting and escaping are often used interchangeably.

You might also see the backslash referred to as the escape character.

Here is another example where escaping is needed:

```
$ echo You owe $1250
You owe 250
```

This seems like a simple echo statement, but notice that the output is not what was expected because the shell treats the \$1 in \$1250 as the shell variable \$1. The \$ meta-character must be quoted in order to avoid variable substitution:

```
$ echo You owe \$1250
```

Now you get the desired output:

```
You owe $1250
```

Now let's say you need to print a message that contains a backslash:

```
$ echo A:\ is my floppy drive
A: is my floppy drive
```

As you can see, the backslash is not present in the output. This is because a single backslash is always used to quote the next character, in this case a space. In order to obtain a backslash, you need to quote it with a backslash as follows:

```
$ echo A:\\ is my floppy drive
A:\ is my floppy drive
```

Meta-Characters and Escape Sequences

The previous examples covered three of the meta-characters that need to be quoted. The complete set of meta-characters that need to be quoted follows:

```
* ? [ ] ' " \ $ ; & ( ) | ^ ! # newline tab
```

Frequently you will see newline and tab expressed as `\n` and `\t` respectively. When the backslash precedes a normal character, such as *n* or *t*, the resulting string, called an *escape sequence*, takes on a special meaning. As you learned in Chapter 5, “Input and Output,” escape sequences make it possible to embed special characters such as newlines and tabs in messages output by scripts.

Using Single Quotes

Here is an echo command that must be modified because it contains many special shell characters:

```
$ echo <-\$1250.**>; (update?) [y|n]
```

You could quote the entire string by putting a backslash in front of each special character, but this is tedious and makes the resulting command difficult to read and understand:

```
$ echo \<-\$1250.**>; (update?) [y|n]
```

An alternative technique for quoting a large group of characters is to put a single quote (') at the beginning and end of the string. When a string is quoted using the single quote, all the meta-characters within the string lose their special meaning and are treated literally. For example, the following command is equivalent to the previous example:

```
$ echo '<-$1250.**>; (update?) [y|n]'
```



Quoting regular characters is harmless, because regular characters are treated the same whether or not they are quoted. This is true for the backslash, single quotes, and double quotes.

In the previous example, you put single quotes around a whole string, quoting both the special characters and the regular letters and digits that do not require quoting. Strictly speaking, you did not have to do this; you could have just quoted those parts of the string that contained meta-characters. Quoting everything is simply easier, both to write and maintain, and incurs no performance penalty.

If a single quote appears within a string to be output, you should not put the whole string within single quotes:

```
$ echo 'It's Friday'
```

This fails and only outputs the following character, while the cursor waits for more input:

```
>
```

The > sign is the secondary shell prompt (as stored in the PS2 shell variable), and it indicates that you have entered a multiple-line command—what you have typed so far is incomplete. Single quotes must be entered in pairs, and their effect is to quote all characters that occur between them. In case you are wondering, you cannot get around this by putting a backslash before an embedded single quote. To correct the problem you need to quote just the single quote in the word *It's* as follows:

```
$ echo It\'s Friday
```

Using Double Quotes

In many cases, you will need to quote some meta-characters but allow others to be evaluated by the shell. For example, the following echo command contains some meta-characters that must be quoted and others that should not:

```
$ echo '$USER owes <-$1250.**>; [ as of (`date +%m/%d`) ]'
```

Because the string is single quoted, the output is easy to predict—what you see is what you get:

```
$USER owes <-$1250.**>; [ as of (`date +%m/%d`) ]
```

As you can imagine, this is not exactly what you wanted; the single quotes have prevented variable substitution and command substitution from occurring, thus the variable `$USER`, which contains the username of the current user, was not replaced with the appropriate value and the `date` command was not executed. So now the problem is to quote most of the meta-characters, such as `*` and `;`, but to allow some meta-characters, such as `$` and ```, to be evaluated.

Double quotes are the solution to this problem. Double quotes disable all of the meta-characters except for `$` and ```, thus allowing variable and command substitution to be performed in a quoted string. Watch what happens if you replace the single quotes with double quotes as follows:

```
$ echo "$USER owes <-$1250.**>; [ as of (`date +%m/%d`) ]"  
Fred owes <-250.**>; [ as of (12/21) ]
```

As you can see, double quotes permit you to display many meta-characters literally while still enabling variable and command substitutions. However, as you might have noticed, the amount of money owed is incorrect because `$1` is substituted. To correct this, you need to use a backslash to escape the `$`:

```
$ echo "$USER owes <-\$1250.**>; [ as of (`date +%m/%d`) ]"
```

The escaped dollar sign is no longer a special character, so the dollar amount appears correctly in the output now:

```
ranga owes <-$1250.**>; [ as of (12/21) ]
```

If you need to print a double quote inside a double-quoted string, you need to quote it with a backslash (`\`) as follows:

```
$ echo "He said \"Hello my dear\""  
He said "Hello my dear"
```

Quoting Rules and Situations

Now that you know the basics about quoting, let's look at some additional rules that will help you use quoting effectively.

Quoting Ignores Word Boundaries

In English, you are used to quoting whole words or sentences. In shell programming, the special characters must be quoted, but it does not matter whether the regular characters are quoted in the same word, as follows:

```
$ echo "Hello; world"  
Hello; world
```

You can move the quotes off word boundaries as long as any special characters remain quoted. This command produces the same output as the preceding one:

```
$ echo Hel"lo; w"orld
```

Of course, it is easier to read the line if the quotes are on word boundaries. This simple example illustrates the manner in which quoting can be used. Quoting off of word boundaries will be useful in some of the more complex quoting situations you will encounter.

Combining Quoting in Commands

You can freely switch from one type of quoting to another within the same command. For example, the following command contains single quotes, a backslash, and double quotes:

```
$ echo The '$USER' variable contains this value \> "$USER"  
The $USER variable contains this value > |ranga|
```

As you can see from the output, you can intermix multiple forms of quoting in the same command.

Embedding Spaces in a Single Argument

To the shell, one or more spaces or tabs form a single command-line argument separator. For example, the output from the following command:

```
$ echo Name           Address
```

does not preserve the spacing:

```
Name Address
```

Even though you put multiple spaces between Name and Address, the shell regards them as special characters forming one separator. The echo command simply displays the arguments it has received separated by a single space. You can quote the spaces to achieve the desired result:

```
$ echo "Name           Address"
```

Now the multiple spaces are preserved in the output:

```
Name           Address
```

Spaces must be quoted to embed them in a single command-line argument:

```
$ mail -s Meeting tomorrow fred jane < meeting.notice
```

The `mail` command enables you to send mail to a list of users. The `-s` option enables the following argument to be used as the subject of the mail. Although the word `tomorrow` is part of the subject (`Meeting tomorrow`), it is taken as one of the users to receive the message, which results in an error. You can solve this by quoting the embedded space within the subject using any of the three types of quoting:

```
mail -s Meeting\ tomorrow fred jane < meeting.notice
mail -s 'Meeting tomorrow' fred jane < meeting.notice
mail -s "Meeting tomorrow" fred jane < meeting.notice
```



Unless the users `fred` and `jane` exist on your system, the `mail` commands in the previous examples will most likely fail to deliver mail and will result in the creation of a file named `dead.letter` in your home directory.

Quoting Newlines to Continue on the Next Line

The newline character is found at the end of each line of a UNIX shell script; it is a special character that tells the shell that it has encountered the end of a line. When a script is being created, the newline character is inserted each time you press `Enter` or `Return` at the end of a line.



Normally you can't see the newline characters in your script, but if you are using the `vi` editor, the `vi` command

```
:set list
```

marks each newline character with a dollar sign. This allows you to see where the newlines are in your scripts.

You can quote the newline character to enable a long command to extend to another line as follows:

```
$ cp file1 file2 file3 file4 file5 file6 file7 \  
> file8 file9 /tmp
```

Notice the last character in the first line is a backslash. This backslash quotes the newline character at the end of the line. The shell recognizes this and displays > (the PS2 prompt) as confirmation that you are entering a continuation line or multiple-line command. For this to work properly, you must not have any characters, including spaces, after the final backslash on the first line.

A quoted newline also acts as an argument separator just like a space or tab. For example:

```
$ echo 'Line 1
> Line 2'
```

The newline at the end of the first line of the command is quoted because it is between a pair of single quotes. The output from this command is

```
Line 1
Line 2
```

Quoting to Access Filenames Containing Special Characters

In the previous chapter, you saw that any word that contains the characters *, ?, [, and] is automatically expanded to a list of files that match the specified pattern. For example, the command:

```
$ rm ch1*
```

removes all files in the current directory whose names start with the prefix `ch1`. In this case, the * character is a special character. Most of the time, this is exactly what you want, but there is a case where you need to use quoting to remove this character's special meaning. Assume you have these files in a directory:

```
ch1      ch1*      ch1a      ch15
```

Notice that the filename `ch1*` contains the * character. Although this is certainly not recommended, sometimes you encounter files whose names contain strange characters (usually such files are created by accident). If you only want to delete the file `ch1*`, the following command is overkill:

```
$ rm ch1*
```

because it deletes all of the files that start with `ch1`. To delete just the file named `ch1*` you need to quote the *. You can use the backslash, the single quote, or the double quote for this purpose:

```
$ rm ch1\*
$ rm 'ch1*'
$ rm "ch1*"
```

Quoting the special character takes away its wildcard meaning and enables you to delete the desired file.



Avoid using special characters in filenames because you have to quote the special character each time you access that file.

Also take extra care when dealing with files that include a filename expansion meta-character in their filenames. If such a filename is supplied to the `mv` or `rm` commands without proper quoting, you might lose many files before you find the mistake.

Quoting Regular Expression Wildcards

In Chapter 16, “Filtering Text with Regular Expressions,” you learn about another type of expression known as regular expressions. Regular expressions use some of the same wildcard characters as filename substitution, as you can see in this `grep` command (which is covered in Chapter 15, “Filtering Text”):

```
grep '[0-9][0-9]*$' report2 report7
```

The quoted string `[0-9][0-9]*$` is a regular expression that `grep` searches for within the contents of files `report2` and `report7`. Wildcards in the `grep` pattern must be quoted to prevent the shell from erroneously replacing that pattern with a list of filenames that match the pattern.



You should always quote your regular expressions to protect them from shell filename expansion, but sometimes they work even if you don't quote them. The shell only expands the pattern if it finds existing files whose names match the pattern. If you happen to be in a directory where no matching files are found, the pattern is left alone, and `grep` works fine. Move to another directory, though, and the same command might fail.

Quoting the Backslash to Enable echo Escape Sequences

In Chapter 5, you saw that `echo` enables you to use escape sequences, such as `\n`, in your output. For example,

```
$ printf "Line 1\nLine 2\n"
```

displays the following:

```
Line 1
Line 2
```

You might be wondering how the quoting rules apply here. If the backslash takes away the special meaning of its following character, shouldn't you just see `n` in the output?

A backslash within double quotes is special only if it precedes one of these four characters:

- `$`
- ```
- `"`
- `\`

The `\n` within double quotes is treated as two normal characters that are passed to the `echo` command as arguments. The `printf` command enables its own set of special characters, which are indicated by a preceding backslash. The `\n` passed to `printf` tells `printf` to display a newline. In this example, the `\n` has to be quoted so that the backslash can be passed to `printf` and not removed before `printf` can see it. Watch what happens when you don't quote the backslash:

```
$ printf Line 1\nLine 2\n
```

This displays:

```
Line 1nLine 2n
```

The `\n` is not quoted, so the shell removed the backslash before `printf` sees the arguments. Because `printf` sees `n`, not `\n`, it simply displays `n`, not a newline as desired.

Quoting Wildcards for `cpio` and `find`

There are other commands like `printf` that have their own special characters that must be quoted for the shell to pass them unaltered. The `cpio` is a command that saves and restores files. It allows you to use the filename expansion meta-characters to select the files to restore. In order for `cpio` to receive these meta-characters in tact, they must be quoted as in the following example:

```
$ cpio -icvdum 'usr2/*' < /dev/rmt0
```

`-icvdum` includes options to `cpio` to specify how it should restore files from the tape device `/dev/rmt0`. The string `usr2/*` says to restore all files from directory `usr2` on tape. Again, this command sometimes works correctly even if the wildcards aren't quoted because shell expansion doesn't occur if matching files aren't found in the current path (in this case, if there is no `usr2` subdirectory in the current directory). It is best to quote these `cpio` wildcards so you can be sure the command works properly every time.



Before using the `cpio` command to restore files from a tape device such as `/dev/rmt0`, please consult your system administrator to ensure that you have sufficient permissions to use such devices.

The `find` command covered in Chapter 18, “Other Tools,” supports its own wildcards as well. For example, in the following command

```
find / -name 'ch*.doc' -print
```

`ch*.doc` is a wildcard pattern that tells `find` to display all filenames that start with `ch` and end with a `.doc` suffix. Unlike shell filename expansion, this `find` command checks all directories on the system for a match. However, the wildcard must be quoted using single quotes, double quotes, or a backslash, so the wildcard is passed to `find` and not expanded by the shell.

Summary

In this chapter, you looked at three types of quoting and when to use them:

- Backslash
- Single quote
- Double quote

In addition, you learned several quoting rules:

- A backslash takes away the special meaning of the character that follows it.
- The character doing the quoting is removed before command execution.
- Single quotes remove the special meaning of all enclosed characters.
- Quoting regular characters is harmless.
- A single quote cannot be inserted within a single quoted string.
- Double quotes remove the special meaning of most enclosed characters.
- Quoting can ignore word boundaries.
- Different types of quoting can be combined in one command.
- Quote spaces to embed them in a single argument.
- Quote the newline to continue a command on the next line.
- Use quoting to access filenames that contain special characters.
- Quote regular expression wildcards.
- Quote the backslash to enable echo escape sequences.

Questions

1. Give an echo command to display this message:
It's <party> time!
2. Give an echo command to display one line containing the following fields:
 - The contents of variable \$USER
 - A single space
 - The word “owes”
 - Five spaces
 - A dollar sign (\$)
 - The contents of the variable \$DEBT (this variable contains only digits)
 - Sample output:
fred owes \$25

Terms

Escaping Escaping a character means to put a backslash (\) just before that character. Escaping can either remove the special meaning of a character in a shell command, or it can add special meaning as you saw with \n in the echo command. The character following the backslash is called an escaped character.

Literal characters These characters have no special meaning and cause no extra action to be taken. Quoting causes the shell to treat a wildcard as a literal character.

Meta-characters A character that has an extra meaning or causes some action to be taken by the shell or other UNIX commands.

Newline This is literally the linefeed character whose ASCII value is 10. In general, the newline character is a special shell character that indicates a complete command line has been entered and can now be executed.

Quoting Literally encloses selected text within some type of quotation marks. When applied to shell commands, quoting disables shell interpretation of special characters by enclosing the characters within single or double quotes or by escaping the characters.

HOUR 11



Flow Control

The order in which commands execute in a shell script is called the *flow* of the script. In the scripts that you have looked at so far, the flow is always the same because the same set of commands executes every time. Most scripts, however, need to change their flow depending on one or more conditions. Commands that allow the flow of a script to be conditionally changed are called *conditional flow control commands*, or just *flow control commands*.

The two main flow control statements available in the shell are:

- The `if` statement
- The `case` statement

The `if` statement is normally used for the conditional execution of commands, whereas the `case` statement enables any number of command sequences to be executed depending on which one of several patterns matches a variable first.

In this hour, you will learn about flow control and two conditional statements.

The if Statement

The `if` statement performs actions depending on whether a given condition is true or false. The `if` statement uses the return code of a command to determine whether a condition is true or false. A return code of zero is treated as true, whereas a non-zero return code is treated as false. The syntax of the `if` statement is as follows:

```
if list1
then
    list2
elif list3
then
    list4
else
    list5
fi
```

Both the `elif` and the `else` statements are optional. If you have an `elif` statement, you don't need an `else` statement and vice versa. An `if` statement can be written with any number of `elif` statements.

Because the `if` statement is treated as a list, it can be also written on a single line:

```
if list1 ; then list2 ; elif list3 ; then list4 ; else list5 ; fi ;
```

Usually this form is used only for short `if` statements.

The execution of the `if` statement is as follows:

1. *list1* is executed.
2. If the exit code of *list1* is 0 (true), *list2* is executed and the `if` statement terminates.
3. Otherwise, *list3* is executed.
4. If the exit code of *list3* is 0 (true), *list4* is executed and the `if` statement terminates.
5. If the exit code of *list3* is non-zero, *list5* is executed.

An if Statement Example

The following example illustrates the use of the `if` statement:

```
if uuencode cherry.gif cherry.gif > cherry.uu ; then
    echo "Encoded cherry.gif to cherry.uu"
else
    echo "Error encoding cherry.gif"
fi
```

Look at the flow of control through this statement:

1. First, the command
`uuencode cherry.gif cherry.gif > cherry.uu`
is executed.
2. If this command is successful, the command
`echo "Encoded cherry.gif to cherry.uu"`
is executed and the `if` statement exits.
3. Otherwise the command
`echo "Error encoding cherry.gif"`
is executed, and the `if` statement exits.

You might have noticed that both the `if` and `then` statements appear on the same line in this example. Most shell programmers prefer to write `if` statements this way in order to make the statement more concise and readable.

Common Errors

Four common errors that can occur when using the `if` statement are

- Omitting the semicolon (;) before the `then` statement in the single line form.
- Using `else if` or `elsif` instead of `elif`.
- Omitting the `then` statement when an `elif` statement is used.
- Writing `if` instead of `fi` at the end of an `if` statement.

The error message generated in each of these cases varies from system to system. In the following examples, a typical error message is displayed; the actual error message on your system may use slightly different wording.

The following example illustrates the first type of error:

```
if uuencode cherry.gif cberry.gif > cherry.uu then
  echo "Encoded cherry.gif to cherry.uu"
else
  echo "Error encoding cherry.gif"
fi
```

This example is the same as the previous example, except that the semicolon, ;, preceding the `then` statement has been omitted. This `if` statement generates an error message similar to the following:

```
sh: syntax error near unexpected token `else'
```

If you encounter an error message like this, make sure that a semicolon precedes the then statement.

The second type of error can be illustrated by modifying the following example:

```
if uuencode cherry.gif cherry.gif > cherry.uu ; then
    echo "Encoded cherry.gif to cherry.uu"
elif rm cherry.uu ; then
    echo "Encoding failed, temporary files removed."
else
    echo "An error occurred."
fi
```

Here you have an `elif` statement that removes the intermediate file `cherry.uu`, if `uuencode` fails. If `elif` is changed to an `else if` as follows

```
if uuencode cherry.gif cherry.gif > cherry.uu ; then
    echo "Encoded cherry.gif to cherry.uu"
else if rm cherry.uu ; then
    echo "Encoding failed, temporary files removed."
else
    echo "An error occurred."
fi
```

an error message similar to the following is generated:

```
sh: syntax error: unexpected end of file
```

If `elif` is changed to `elsif` as follows

```
if uuencode cherry.gif cherry.gif > cherry.uu ; then
    echo "Encoded cherry.gif to cherry.uu"
elsif rm cherry.uu ; then
    echo "Encoding failed, temporary files removed."
else
    echo "An error occurred."
fi
```

an error message similar to the following is generated:

```
sh: syntax error near unexpected token 'then'
```

The following example illustrates the third type of error:

```
if uuencode cherry.gif cherry.gif > cherry.uu ; then
    echo "Encoded cherry.gif to cherry.uu"
elif rm cherry.uu
    echo "Encoding failed, temporary files removed."
else
    echo "An error occurred."
fi
```

Here the `then` statement following the `elif` statement has been omitted. This generates an error message similar to the following:

```
sh: syntax error near unexpected token 'else'
```

The following example illustrates the fourth type of error:

```
if uuencode cherry.gif cberry.gif > cherry.uu ; then
    echo "Encoded cherry.gif to cherry.uu"
else
    echo "Error encoding cherry.gif"
if
```

Here the final `fi` statement is written as `if`. This generates an error message similar to the following:

```
sh: syntax error: unexpected end of file
```

This error indicates that the `if` statement was not closed with a `fi` statement.

Using test

Usually the list given to an `if` statement is one or more `test` commands. A `test` command has the following syntax:

```
test expr
```

Here *expr* is constructed using one of the options understood by `test`. After evaluating *expr*, `test` returns either 0 (true) or 1 (false). The open bracket, `[`, is often used as a shorthand for `test`:

```
[ expression ]
```

Here *expr* is any valid expression understood by `test`. The close bracket, `]`, the space after the open bracket, `[`, and the space before the close bracket are required. Without the spaces and the close bracket, the shell cannot tell where *expr* begins and ends.

There are three main types of expressions understood by `test`:

- File tests
- String comparisons
- Numerical comparisons

File Tests

File test expressions test whether a file fits a particular criteria. The general syntax for a file test is

```
test option file
```

or

```
[ option file ]
```

Here *option* is one of the options given in Table 11.1 and *file* is the name of a file or directory.

TABLE 11.1 File Test Options for test

<i>Option</i>	<i>Description</i>
-b <i>file</i>	True if <i>file</i> exists and is a block special file.
-c <i>file</i>	True if <i>file</i> exists and is a character special file.
-d <i>pathname</i>	True if <i>pathname</i> exists and is a directory.
-e <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists.
-f <i>file</i>	True if <i>file</i> exists and is a regular file.
-g <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and has its SGID bit set.
-h <i>file</i>	True if <i>file</i> exists and is a symbolic link. This option is not available on some older systems.
-k <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and has its “sticky” bit set.
-p <i>file</i>	True if <i>file</i> exists and is a named pipe.
-r <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and is readable.
-s <i>file</i>	True if <i>file</i> exists and has a size greater than zero.
-u <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and has its SUID bit set.
-w <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and is writeable.
-x <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and is executable. A directory must be executable in order for its contents to be accessed.
-O <i>pathname</i>	True if the file or directory specified by <i>pathname</i> exists and is owned by the effective user ID of the current process.



The `-w` and `-x` options only test a file's permission flags. They do not take into account the state of the underlying disk. For example, files on read-only disks such as CD-ROMs or DVDs can have the writeable bit set, but they cannot be written to because the underlying media is read-only.

The next few examples, taken from OpenBSD's system startup script `/etc/rc`, illustrate the use of file tests. This section doesn't go over every option listed in Table 11.1; just enough to give you a general idea about how file test operators are used in the real world.

The first example illustrates the use of the `-d` option to test for the existence of a directory:

```
# /var/crash should be a directory if core dumps
# are to be saved.
if [ -d /var/crash ]; then
    savecore /var/crash
fi
```

Here the script determines whether the directory `/var/crash` exists. If the directory exists, `savecore` is executed. If the directory does not exist, the `if` statement performs no actions.

The second example illustrates the use of the `-f` option to test for the existence of a regular file:

```
if [ -f /var/account/acct ]; then
    echo 'turning on accounting'; accton /var/account/acct ;
fi
```

Here the script determines whether the file `/var/account/acct` exists. If the file exists, the script outputs the message:

```
turning on accounting
```

and executes `accton`. If the file does not exist, the `if` statement performs no actions.

The third example illustrates the use of the `-x` option to determine whether a file is executable:

```
if [ -x /usr/libexec/vi.recover ]; then
    echo 'preserving editor files'; /usr/libexec/vi.recover
fi
```

Here the script determines whether the file `/usr/libexec/vi.recover` is executable. If the file is executable, the script outputs the message:

```
preserving editor files
```

and executes `/usr/libexec/vi.recover`. If the file is not executable, the `if` statement performs no actions.

String Comparisons

The `test` and `[]` commands allow for simple string comparisons. They can be used to determine whether a string is empty and whether two strings are identical or equal. The options relating to string comparisons are listed in Table 11.2.

TABLE 11.2 String Comparison Options for the `test` Command

<i>Option</i>	<i>Description</i>
<code>-z str</code>	True if <code>str</code> has zero length.
<code>-n str</code>	True if <code>str</code> has nonzero length.
<code>str1 = str2</code>	True if <code>str1</code> and <code>str2</code> are equal.
<code>str1 != str2</code>	True if <code>str1</code> and <code>str2</code> are not equal.

Checking Whether a String Is Empty

There are several ways to determine whether a string is empty. The most common method is to use the `-z` option as follows:

```
test -z str
```

or

```
[ -z str ]
```

Here `str` is the string you want to check. As an example, consider the following `if` statement:

```
if [ -z "$FRUIT_BASKET" ] ; then
    echo "Your fruit basket is empty"
else
    echo "Your fruit basket contains: $FRUIT_BASKET"
fi
```

If the variable `$FRUIT_BASKET` does not have a value, the message:

```
Your fruit basket is empty
```

is produced. Otherwise a message that contains the value of `$FRUIT_BASKET` is produced.

If you were to use the `-n` option instead of the `-z` option, the example would change as follows:

```
if [ -n "$FRUIT_BASKET" ] ; then
    echo "Your fruit basket has the following fruit: $FRUIT_BASKET"
else
    echo "Your fruit basket is empty"
fi
```

You might have noticed that the variable `$FRUIT_BASKET` was quoted in the previous examples. Quoting handles the case when `$FRUIT_BASKET` is unset or null. If `$FRUIT_BASKET` was unset and you did not quote it, an error message similar to the following would be displayed:

```
test: argument expected
```

Without quotes, after the shell performs variable substitution, the statement looks like the following:

```
[ -z ]
```

The `test` command complains that a required argument is missing, because the *str* argument to `-z` is missing. When quoting is used, the same statement looks like the following after variable substitution:

```
[ -z "" ]
```

Here *str* is "", so `test` works correctly.



When bash is presented with a variable that does not have a value, it automatically uses the value "". Thus you do not have to quote your variables in bash.

Although the quoting is not required in bash, you should still include it in your script for the sake of clarity and maintainability.

Equality of Strings

The `test` and `[` commands enable you to determine whether two strings are equal. Two strings are considered equal if they contain the identical sequence of characters. For example, the following strings are considered equal:

```
"There are more things in heaven and earth"  
"There are more things in heaven and earth"
```

Whereas the following strings are not considered equal because of differences in capitalization:

```
"than are dreamt of in your philosophy"  
"Than are dreamt of in your Philosophy"
```

The syntax for checking whether two strings are equal is

```
test str1 = str2
```

or

```
[ str1 = str2 ]
```


Here *str1* and *str2* are the two strings being compared. If these two strings are equal, the test succeeds and returns true (0). If the two strings are not equal, the test fails and returns false (1).

The following example, slightly modified from OpenBSD's */etc/rc*, illustrates a common use of string comparisons:

```
# if $portmap == YES, the portmapper is started.
if [ "$portmap" = "YES" ]; then
    echo -n ' portmap'; portmap
fi
```

Here *str1* is the value of `$portmap` and *str2* is the string YES. The `if` statement uses the `=` operator to determine whether the value stored in `$portmap` is equal to YES. If `$portmap` is equal to YES, a message is issued and the program `portmap` is executed.

Note that `$portmap` is quoted in this example. Just like in previous examples, quoting is used to prevent problems resulting from variable substitution when `$portmap` happens to be unset or null. If `$portmap` was not quoted and it happened to be null, an error message similar to the following would be produced:

```
test: argument expected
```

An alternative technique to quoting is sometimes used to avoid these types of errors. Basically it involves prefixing *str1* and *str2* with an extra character, usually *x*. The syntax for this technique is

```
test Xstr1 = Xstr2
```

or

```
[ Xstr1 = Xstr2 ]
```

If either *str1* or *str2* is null, the string *x* is used instead of null. Rewriting the previous example using this technique yields:

```
# if $portmap == YES, the portmapper is started.
if [ X$portmap = X"YES" ]; then
    echo -n ' portmap'; portmap
fi
```

If `$portmap` is null then the strings *x* and *xyes* are compared; because these strings do not match, the test fails. If `$portmap` is YES then the strings *xyes* and *xyes* are compared; because these strings match, the test succeeds.

Inequality of Strings

You can determine whether two strings are not equal using the `!=` operator. The syntax for this operator is similar to that of the `=` operator:

```
test str1 != str2
```

or

```
[ str1 != str2 ]
```

Here *str1* and *str2* are the two strings being compared. If these two strings are not equal, the test succeeds and returns 0 (true). If the two strings are equal, the test fails and returns 1 (false).

The following example, slightly modified from OpenBSD's `/etc/rc`, illustrates the use of the `!=` operator:

```
if [ "$lpd_flags" != "NO" ]; then
    echo -n ' printer'; lpd $lpd_flags
fi
```

Here *str1* is the value of the variable `$lpd_flags` and *str2* is the string `NO`. You can determine whether the value of `$lpd_flags` is not `NO`. If the value is something other than `NO`, the program issues a message and executes `lpd` with the value of `$lpd_flags` as its argument.

Just as in previous examples, quoting was used in order to handle the case when `$lpd_flags` is unset or null. An alternate technique that is sometimes used involves prefixing *str1* and *str2* with an extra character, usually `X`. The syntax for this technique is

```
test Xstr1 != Xstr2
```

or

```
[ Xstr1 != Xstr2 ]
```

If either *str1* or *str2* is null, the string `X` is used instead of null. Rewriting the previous example using this technique yields:

```
if [ X$lpd_flags != X"NO" ]; then
    echo -n ' printer'; lpd $lpd_flags
fi
```

If `$lpd_flags` is null, the strings `X` and `XNO` are compared; because these strings do not match, the test succeeds. If `$lpd_flags` is `NO` then the strings `XNO` and `XNO` are compared; because these strings match, the test fails.

Numerical Comparisons

The `test` and `[` commands can also be used to compare integers. The basic syntax is

```
test int1 op int2
```

or

```
[ int1 op int2 ]
```

Here *int1* and *int2* can be any positive or negative integer and *op* is one of the operators listed in Table 11.3. If either *int1* or *int2* is a string, not an integer, it is treated as 0.

TABLE 11.3 Numerical Comparison Operators for the `test` Command

<i>Operator</i>	<i>Description</i>
<i>int1</i> -eq <i>int2</i>	True if <i>int1</i> equals <i>int2</i> .
<i>int1</i> -ne <i>int2</i>	True if <i>int1</i> is not equal to <i>int2</i> .
<i>int1</i> -lt <i>int2</i>	True if <i>int1</i> is less than <i>int2</i> .
<i>int1</i> -le <i>int2</i>	True if <i>int1</i> is less than or equal to <i>int2</i> .
<i>int1</i> -gt <i>int2</i>	True if <i>int1</i> is greater than <i>int2</i> .
<i>int1</i> -ge <i>int2</i>	True if <i>int1</i> is greater than or equal to <i>int2</i> .

A common task in a shell script is checking the exit code from a program. The numerical comparison operators allow you to easily check the exit status of a command and perform different actions depending on whether a command executed correctly. For example, consider the following command:

```
ln -s /usr/local/bin/bash /usr/bin
```

If you execute this command on the command line, you can see any error messages and intervene to fix the problem. In a shell script, error messages are ignored and the script continues to execute. In most cases it is a mistake to ignore errors.

The exit status of the last command is stored in the variable `$?` , so you can use this variable to check whether a command was successful as follows:

```
if [ $? -eq 0 ] ; then
    echo "Command was successful." ;
else
    echo "An error was encountered."
    exit
fi
```

Recall that an exit code of 0 indicates success and a non-zero exit code indicates failure. If the command exits with an exit code of 0, the “success” message is issued; otherwise, an error message is issued and `exit` is called.



In some scripts you may see the = operator used in place of the -eq operator. Some extremely old versions of the shell did not include the -eq operator, thus shell programmers were forced to use the = operator instead. All modern shells, including Bourne shell, ksh, bash and zsh support the -eq operator.

By using the -ne operator, you can simplify the previous example as follows:

```
if [ $? -ne 0 ] ; then
    echo "An error was encountered."
    exit
fi
echo "Command was successful."
```

Here you check to see whether the command failed. If so, you issue an error message and exit; otherwise, you continue with the rest of the program (in this case you just issue the “success” message). This version is slightly more efficient than the previous example that uses an else clause.



In some scripts, you might see the != operator used in place of the -ne operator. Some older versions of the shell did not include the -ne operator, thus shell programmers were forced to use the != operator instead. All modern shells, including the Bourne shell, ksh, bash, and zsh, support the -ne operator.

Compound Expressions

So far you have dealt with individual expressions, but many times you need to combine expressions in order to test for a particular condition. When two or more expressions are combined, the result is called a *compound* expression.

You can create compound expressions by using test and ['s built-in operators, or by using the conditional execution operators, && and ||. Another way to create a compound expression is to use the negation operator, !, which negates an expression. Table 11.4 summarizes these operators..

TABLE 11.4 Operators for Creating Compound Expressions

Operator	Description
! <i>expr</i>	True if <i>expr</i> is false.
<i>expr1</i> -a <i>expr2</i>	True if both <i>expr1</i> and <i>expr2</i> are true.
<i>expr1</i> -o <i>expr2</i>	True if either <i>expr1</i> or <i>expr2</i> is true.

The syntax for creating compound expressions using the built-in operators is

```
test expr1 op expr2
```

or

```
[ expr1 op expr2 ]
```

Here *expr1* and *expr2* are any valid test expressions, and *op* is `-a` (short for and) or `-o` (short for or). If the `-a` operator is used, both *expr1* and *expr2* must be true in order for the compound expression to be true. If the `-o` operator is used, either *expr1* or *expr2* must be true in order for the compound expression to be true.

The syntax for creating compound expressions using the conditional operators is

```
test expr1 op test expr2
```

or

```
[ expr1 ] op [ expr2 ]
```

Here *expr1* and *expr2* are any valid test expressions, and *op* is `&&` (and) or `||` (or). If the `&&` operator is used, both *expr1* and *expr2* must be true in order for the compound expression to be true. If the `||` operator is used, either *expr1* or *expr2* must be true in order for the compound expression to be true.

The following `if` statement, taken from OpenBSD's `/etc/rc`, illustrates a compound expression constructed using the built-in operator `-a`:

```
if [ -f /sbin/kbd -a -f /etc/kbdtype ]; then
    kbd `cat /etc/kbdtype`
fi
```

This `if` statement is executed as follows:

1. First the test

```
-f /sbin/kbd
```

is performed. If the file `/sbin/kbd` exists then the test returns true (`0`), otherwise the test returns false and the `if` statement performs no actions.

2. If `/sbin/kbd` exists, the second test

```
-f /etc/kbdtype
```

is performed. If the file `/etc/kbdtype` exists then the test returns true (`0`), otherwise the test returns false and the `if` statement performs no actions. If the first test failed, this test is not performed.

3. If both files exist, the command

```
kbd `cat /etc/kbdtype`
```

is executed.

As an illustration of how the conditional operators are used, the previous example can be rewritten to use conditional operators as follows:

```
if [ -f /sbin/kbd ] && [ -f /etc/kbdtype ]; then
    kbd `cat /etc/kbdtype`
fi
```

You just replaced the `-a` operator with `] && [`. The execution of this version is similar to that of the previous example:

1. First the expression

```
[ -f /sbin/kbd ]
```

is evaluated. If the file `/sbin/kbd` exists then the expression returns true (0), otherwise the test returns false and the `if` statement performs no actions.

2. If `/sbin/kbd` exists, the expression

```
[ -f /etc/kbdtype ]
```

is evaluated. If the file `/etc/kbdtype` exists then the test returns true (0), otherwise the test return false and the `if` statement performs no actions. If the first test failed, this test is not performed.

3. If both files exist, the command

```
kbd `cat /etc/kbdtype`
```

is executed.

The difference between the two versions is that in the first version `list1` is a single command, whereas in the second version `list1` is a compound command.

Some programmers prefer the version that uses the conditional operators because the individual tests are isolated. Other programmers prefer the second form because it invokes the `[` command only once and thus might be marginally more efficient for large numbers of tests. If your shell scripts need to be portable, you should use the conditional operators.



In the previous examples you used only two expressions in your compound expressions. You are not limited to two expressions. You can combine any number of expressions into one compound expression.

Negating an Expression

Negation reverses the result of a test expression. An expression that would have been true is treated as false and vice versa. The basic syntax of the negation operator is

```
test ! expr
```

or

```
[ ! expr ]
```

Here *expr* is any valid test expression.

The following example, taken from OpenBSD's `/etc/rc` startup script, illustrates the use of the `!` operator:

```
if [ ! -f /etc/motd ]; then
    install -c -o root -g wheel -m 664 /dev/null /etc/motd
fi
```

This example creates the file `/etc/motd` (the message of the day on UNIX systems) using the `install` command if it does not exist or is not a regular file. The execution is as follows:

1. First the test
`-f /etc/motd`
is performed.
2. The result of the test is negated because of the `!` operator. If the file `/etc/motd` exists and is a regular file, the compound expression returns false (1); otherwise, it returns true.
3. If the result of the previous step is true, the file `/etc/motd` is created; otherwise, the `if` statement performs no actions.

This example can also be written as either of the following commands:

```
test ! -f /etc/motd && install -c -o root -g wheel -m 664
➤ /dev/null /etc/motd[ -f /etc/motd ] && install -c -o root -g
➤ wheel -m 664 /dev/null /etc/motd
```

This achieves the same result because `install` is executed only if the `test` or `[` commands return true.

The case Statement

The case statement is the second form of flow control available in the shell. Its syntax is as follows:

```
case word in
    pattern1)
        list1
        ;;
    pattern2)
        list2
        ;;
    ...
    patternN)
        listN
        ;;
esac
```

Here the string *word* is compared to each of the patterns from *pattern1* to *patternN*. When a matching pattern is found, the list following the matching pattern is then executed.

When a list finishes executing, the special command `;;` indicates that flow should jump to the end of the case statement. The `;;` is similar to the `break` command in the C programming language. If no matches are found, the case statement does not perform any actions. The minimum number of patterns is one. There is no limit on the maximum number of patterns.

Some programmers prefer to use a more concise form of the case statement, written as follows:

```
case word in
    pattern1) list1 ;;
    ...
    patternN) listN ;;
esac
```

This form should be used only if the list of commands to be executed is short.

A case Statement Example

The following example illustrates the use of the case statement:

```
FRUIT=kiwi
case "$FRUIT" in
    apple) echo "Apple pie is quite tasty." ;;
    banana) echo "I like banana nut bread." ;;
    kiwi) echo "New Zealand is famous for kiwi." ;;
esac
```


The execution of the case statement is as follows:

1. The string contained in the variable `FRUIT` is expanded to `kiwi`.
2. The string `kiwi` is compared against the first pattern, `apple`. Because they don't match, the program goes on to the next pattern.
3. The string `kiwi` is compared against the next pattern, `banana`. Because they don't match, the program goes on to the next pattern.
4. The string `kiwi` is compared against the final pattern, `kiwi`. Because they match, the following message is produced:

```
New Zealand is famous for kiwi.
```

Common Errors

Two common errors that are encountered while using the case statement are as follows:

- Ommitting the `;;` at the end of a list.
- Writing `case` instead of `esac` at the end of the case statement.

To illustrate the first type of error, the previous example is modified so that the `;;` is missing after the first list:

```
FRUIT=kiwi
case "$FRUIT" in
  apple) echo "Apple pie is quite tasty."
  banana) echo "I like banana nut bread." ;;
  kiwi) echo "New Zealand is famous for kiwi." ;;
esac
```

This omission produces an error message similar to the following:

```
bash: syntax error near unexpected token `banana)'
```

What this error message means is that while the shell was trying to execute the list for the pattern `apple`, it saw the start of the pattern `banana`. Because this pattern started before the shell encountered the end of the list for the pattern `apple`, an error message was produced. To illustrate the second type of error, the ending `esac` is changed to `case`:

```
FRUIT=kiwi
case "$FRUIT" in
  apple) echo "Apple pie is quite tasty." ;;
  banana) echo "I like banana nut bread." ;;
  kiwi) echo "New Zealand is famous for kiwi." ;;
case
```

This change produces an error message similar to the following:

```
bash: syntax error near unexpected token `case'
```

What this error message means is that the shell did not see the appropriate closing `esac` for the case statement.

Using Patterns

In the previous example, you used fixed strings as the pattern. When used in this fashion, the case statement is basically an `if` statement. For example, the `if` statement corresponding to the case statement in previous example is

```
if [ "$FRUIT" = apple ] ; then
    echo "Apple pie is quite tasty."
elif [ "$FRUIT" = banana ] ; then
    echo "I like banana nut bread."
elif [ "$FRUIT" = kiwi ] ; then
    echo "New Zealand is famous for kiwi."
fi
```

Although the case statement is more concise and readable, the real power of the case statement does not lie in enhancing the readability of your scripts; its power lies in the fact that it uses *patterns* rather than fixed strings to perform matching. A pattern is a string that consists of regular characters and special *wildcard* characters. The pattern determines whether a match is present. The case statement patterns use the same special characters as patterns for pathname expansion covered in Chapter 9, “Substitution.” The patterns can also include the OR operator, `|`.

An example of a simple case statement that uses patterns is as follows:

```
case $- in
    *i*) # an interactive shell
        PS1="\u005Cuname -n`$ "
        PATH="$PATH:$HOME/bin"
        export PS1 PATH ;;
esac
```

The special variable `$-` contains a list of the shell options. In this case, you determine whether that list contains the letter `i`, which indicates that the shell is interactive. Checking if `$-` contains the letter `i` is the most portable method for determining whether the shell is running in interactive mode or in non-interactive mode. In this example, you set up the prompt, `PS1`, and the command search path, `PATH`, if the shell is running in interactive mode; otherwise, no actions are performed.

Summary

In this chapter, you examined the two main flow control mechanisms available in the shell: `if` and `case`. You also looked at the `test` command and its use in `if` statements. Specifically, the chapter covered the following topics:

- Performing file tests
- Performing string comparisons
- Performing numerical comparisons
- Using compound expressions

In the next chapter, you will examine loops, which are complementary to flow control statements.

Questions

1. What is the difference between the following commands?

```
if [ -e /usr/local/bin/bash ] ; then /usr/local/bin/bash ; fi
if [ -x /usr/local/bin/bash ] ; then /usr/local/bin/bash ; fi
```

2. Given the following variable declarations,

```
HOME=/home/ranga
BINDIR=/home/ranga/bin
```

what is the output of the following `if` statement?

```
if [ $HOME/bin = $BINDIR ] ; then
    echo "Your binaries are stored in your home directory."
fi
```

3. Write a test command that can be used to test if `/usr/bin` is a directory or a symbolic link.
4. Given the following `if` statement, write an equivalent `case` statement:

```
if [ "$ANS" = "Yes" -o "$ANS" = "yes" -o "$ANS" = "y" -o "$ANS" = "Y" ] ;
then
    ANS="y"
else
    ANS="n"
fi
```

Terms

Conditional flow control commands Commands that allow the flow of a script to be conditionally changed. Also called flow control commands.

Compound expression When two or more expressions are combined, the result is called a compound expression.

Flow control commands See conditional flow control commands.

Negating expressions Reverses the result of a test expression. An expression that would have been true is treated as false and vice versa.

This page intentionally left blank

HOUR 12



Loops

Loops enable you to execute a series of commands multiple times. The two main types of loops are the `while` loop and the `for` loop. In addition to these two types of loops, `ksh`, `bash`, and `zsh` support an additional type of loop called the `select` loop. It can be used to present a menu of choices to a shell scripts user. In this chapter, you will examine loops in detail. Specifically, this chapter covers the following topics:

- The `while` loop
- The `for` loop
- The `select` loop
- Loop control

The `while` Loop

The `while` loop enables you to execute a set of commands repeatedly until some condition occurs. It is normally used when you need to manipulate the value of a variable repeatedly. The basic syntax of the `while` loop is

```
while cmd
do
    list
done
```

Here, *cmd* is a single command, whereas *list* is a list of one or more commands. Although *command* can be any valid UNIX command, it is usually a test expression of the type covered in Chapter 11, “Flow Control.” The list called *list* is commonly referred to as the *body* of the *while* loop. The *do* and *done* keywords are not considered part of the body of the loop because the shell uses them only for determining where the *while* loop begins and ends.

The execution of a *while* loop proceeds according to the following steps:

1. Execute *cmd*.
2. If the exit status of *cmd* is nonzero, exit from the *while* loop.
3. If the exit status of *cmd* is zero, execute *list*.
4. When *list* finishes executing, return to Step 1.

If both *cmd* and *list* are short, the *while* loop can be written on a single line as follows:

```
while cmd ; do list ; done
```

Here is a simple example that uses the *while* loop to display the numbers from zero to nine:

```
x=0
while [ $x -lt 10 ]
do
    echo $x
    x=`expr $x + 1`
done
```

Its output looks like this:

```
0
1
2
3
4
5
6
7
8
9
```

Each time this loop executes, the variable *x* is checked to see whether it has a value that is less than 10. If the value of *x* is less than 10, this test expression has an exit status of 0. In this case, the current value of *x* is displayed and then *x* is incremented by 1. If *x* is equal to 10 or greater than 10, the test expression returns 1, causing the *while* loop to exit.



The previous example uses the `expr` command to increment and decrement the variable `$x`. If you are not familiar with the `expr` command, don't worry, it is covered in Chapter 18, "Other Tools."

Nesting while Loops

It is possible to use a `while` loop as part of the body of another `while` loop as follows:

```
while cmd1 ; # this is loop1, the outer loop
do
    list1
    while cmd2 ; # this is loop2, the inner loop
    do
        list2
    done
    list3
done
```

Here `cmd1` and `cmd2` are single commands, whereas `list1`, `list2`, and `list3` are sets of one or more commands. Both `list1` and `list3` are optional.

In situations in which there are two `while` loops, `loop1` and `loop2`, `loop1` is referred to as the *main loop* or *outer loop*, and `loop2` is referred to as the *inner loop*. When describing the inner loop, `loop2`, many programmers say that it is *nested* one level deep. The term *nested* refers to the fact that `loop2` is located in the body of `loop1`. If you had a `loop3` located in the body of `loop2`, it would be nested two levels deep. The level of nesting is relative to the outermost loop. There are no restrictions on how deeply nested loops can be, but you should try to avoid nesting loops more deeply than four or five levels to avoid difficulties in finding and fixing problems in your scripts.

As an illustration of loop nesting, let's add another countdown loop inside the loop that you used to count to nine:

```
x=0
while [ "$x" -lt 10 ] ; # this is loop1
do
    y="$x"
    while [ "$y" -ge 0 ] ; # this is loop2
    do
        printf "$y "
        y=`expr $y - 1`
    done
    echo
    x=`expr $x + 1`
done
```


The main change introduced is the variable *y*. It is set to the value of *x* - 1 before *loop2* executes. Because of this, each time *loop2* executes, it displays all the numbers greater than 0 and less than *x* in reverse order. The output looks like the following:

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Validating User Input with `while`

Say that you need to write a script that needs to ask the user for the name of a directory. You could use the following steps to get information from the users:

1. Ask the user a question.
2. Read the user's response.
3. Determine whether the user responded with the name of a directory.

But what should you do when the user gives you a response that is not a directory?

The simplest choice would be to do nothing, but this is not very user friendly. Your script can be much more user friendly by informing the user of the error and asking for the name of a directory again. The `while` loop is perfect for doing this. In fact, a common use for the `while` loop is to determine whether user input has been gathered correctly. Usually a strategy similar to the following is employed:

1. Set a variable's value to null.
2. Start a `while` loop that exits when the variable's value is not null.
3. In the `while` loop, ask the user a question and read in the users response.
4. Validate the response.
5. If the response is invalid the variable's value is set to null. This enables the `while` loop to repeat.
6. If the response is valid, the variable's value is not changed. It continues to hold the user's response. Because the variable's value is not null, the `while` loop exits.

A `while` loop that implements this is

```
RESPONSE=
while [ -z "$RESPONSE" ] ;
do
```

```

    echo "Enter the name of a directory where your files are
    located:\c "
    read RESPONSE
    if [ ! -d "$RESPONSE" ] ; then
        echo "ERROR: Please enter a directory pathname."
        RESPONSE=
    fi
done

```

Here, you store the user's response in the variable `RESPONSE`. Initially this variable is set to null, enabling the `while` loop to begin executing. When the `while` loop first executes, the user is prompted as follows:

```
Enter the name of a directory where your files are located:
```

The user can type the name of a directory at this prompt. When the user finishes typing and presses Enter, `read` stores the input into the variable `RESPONSE`. You then check to make sure the input is a directory. If the input is not a directory, you issue an error message and repeat. An error message is produced so that the user knows what was wrong with the input. If the user does not enter any value, the variable `RESPONSE` is still set to null. In this case the value stored in the variable `RESPONSE` is not a directory, thus the error message is produced.

Input Redirection and `while`

The `while` loop can also be combined with input redirection and `read` in order to read a file one line at a time. The basic syntax is

```

while read LINE
do
: # manipulate file here
done < file

```

In the body of the `while` loop, you can manipulate each line of the specified *file*. A simple example of this is

```

while read LINE
do
    case $LINE in
        *root*) echo $LINE ;;
    esac
done < /etc/passwd

```

Here only the lines that contain the string `root` in the file `/etc/passwd` are displayed. The output will be similar to the following:

```
root:x:0:1:Super-User:/:/sbin/sh
```

while and Subshells

A problem with the loop used in the previous example is that it is executed in a subshell in Bourne shell and older versions of ksh. This means that any changes to the script environment, such as exporting variables and changing the current working directory, might not be present after the `while` loop completes. As an example, consider the following script:

```
#!/bin/sh
if [ -f "$1" ] ; then
    i=0
    while read LINE
    do
        i=`expr $i + 1`
    done < "$1"
    echo $i
fi
```

This script tries to count the number of lines in the file specified to it as an argument. Executing this script on the file

```
$ cat dirs.txt
/tmp
/usr/local
/opt/bin
/var
```

can produce the following incorrect result:

```
0
```

Although you are incrementing the value of `$i` using the command

```
i=`expr $i + 1`
```

when the `while` loop completes, the value of `$i` is not preserved. In this case, you need to change a variable's value inside the `while` loop and then use that value outside the loop. One way to solve this problem is to redirect STDIN prior to entering the loop and then restore STDIN after the loop completes. The basic syntax is

```
exec n<&0 < file
while read LINE
do
    : # manipulate file here
done
exec 0<&n n<&-
```

Here, n is an integer greater than 2, and *file* is the name of the file you want to read. Usually n is chosen as a small number such as 3, 4, or 5. This allows you to construct a shell version of the `cat` command as follows:

```
#!/bin/sh
if [ $# -ge 1 ] ; then
  for FILE in $@
  do
    exec 4<&0 < "$FILE"
    while read LINE ; do echo $LINE ; done
    exec 0<&4 4<&-
  done
fi
```

The `until` Loop

The `while` loop is perfect for a situation where you need to execute a set of commands while some condition is true, but sometimes you need to execute a set of commands until a condition is true. The `until` loop, available in `ksh`, `bash` and `zsh`, provides this functionality. Its syntax is

```
until cmd
do
  list
done
```

Here *cmd* is a single command, whereas *list* is a set of one or more commands. Although *cmd* can be any valid UNIX command, it is usually a `test` expression of the type covered in Chapter 11, “Flow Control.”

The execution of an `until` loop is similar to that of the `while` loop:

1. Execute *cmd*.
2. If the exit status of *cmd* is nonzero, exit from the `until` loop.
3. If the exit status of *cmd* is zero, execute *list*.
4. When *list* finishes executing, return to Step 1.

If both *cmd* and *list* are short, the `until` loop can be written on a single line as follows:

```
until cmd ; do list ; done
```

In most cases an `until` loop is identical to a `while` loop with *cmd* negated using the `!` operator. For example, the following `while` loop

```
x=1
while [ ! $x -ge 10 ]
do
  echo $x
  x=`expr $x + 1`
done
```

is equivalent to the following `until` loop:

```
x=1;
until [ $x -ge 10 ]
do
    echo $x
    x=`expr $x + 1`
done
```

The `until` loop offers no advantages over the equivalent `while` loop. Because it isn't supported by the Bourne shell, most programmers do not favor it. It is covered here for the sake of completeness.

The `for` and `select` Loops

Unlike the `while` loop, which exits when a certain condition is false, the `for` and `select` loops operate on lists of items. This section covers these two loops in detail.

The `for` Loop

The `for` loop enables you to execute a set of commands repeatedly for each item in a list. One of its most common uses is in performing the same set of commands for a large number of files. The basic syntax is

```
for name in word1 word2 ... wordN
do
    list
done
```

Here *name* is the name of a variable and *word1* to *wordN* are sequences of characters separated by spaces (words). Each time the `for` loop executes, the value of the variable *name* is set to the next word in the list of words, *word1* to *wordN*. The first time, *name* is set to *word1*; the second time, it's set to *word2*; and so on. This means that the number of times a `for` loop executes depends on the number of words that are specified. For example, if the following words were specified to a `for` loop

```
there comes a time
```

the loop would execute four times. In each iteration of the `for` loop, the commands specified in *list* are executed.

A `for` loop can be written on a single line as follows:

```
for name in word1 word2 ... wordN ; do list ; done
```

If *list* and the number of words are short, the single line form is often chosen; otherwise, the multiple-line form is preferred.

A simple for loop example is

```
for i in 0 1 2 3 4 5 6 7 8 9
do
    echo $i
done
```

This loop counts to nine as follows:

```
0
1
2
3
4
5
6
7
8
9
```

Note that although the output is identical to the `while` loop, the `for` loop does something altogether different. In each iteration, `$i` is set to the next item in the list. When the list is finished, the loop exits. In this example, you chose the list to be the numbers from 0 to 9. In the `while` loop, the next number to display was being computed, and it was not part of a predetermined list.

If you change the list slightly, notice how the output changes:

```
for i in 0 1 2 4 3 5 8 7 9
do
    echo $i
done

0
1
2
4
3
5
8
7
9
```

Manipulating a Set of Files

Say that you need to copy a bunch of files from one directory to another and change the permissions on the copy. You could do this by copying each file and changing the permissions manually.

A better solution is to determine the commands you need to execute in order to copy a file and change its permissions, and then have the computer do this for every file you

were interested in. In fact this is one of the most common uses of the `for` loop—iterating over a set of filenames and performing some operations on those files.

The procedure to do this follows:

1. Create a `for` loop with a variable named `file` or `FILE`. Other favored names include `i`, `j`, and `k`. The name of the variable is usually singular.
2. Create a list of files to manipulate. This is frequently accomplished using the filename substitution technique discussed in Chapter 9, “Substitution.”
3. Manipulate the files in the body of the loop.

An example of this is the following `for` loop:

```
for FILE in $HOME/.bash*
do
    cp $FILE ${HOME}/public_html
    chmod a+r ${HOME}/public_html/${FILE}
done
```

In this loop, you use filename substitution to obtain a list of files in your home directory that start with `.bash*`. In the body of the loop, each of these files is copied to the directory `public_html` and made readable by everyone.

The `select` Loop

The `select` loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices. The `select` loop was introduced in `ksh` and has been adapted by `bash` and `zsh`. It is not available in the Bourne shell.

The basic syntax of the `select` loop is

```
select name in word1 word2 ... wordN
do
    list
done
```

Here *name* is the name of a variable and *word1* to *wordN* are sequences of characters separated by spaces (words). The set of commands to execute after the user has made a selection is specified by *list*.

The execution process of a `select` loop is as follows:

1. Each item in *list1* is displayed along with a number.
2. A prompt, usually `#!`, is displayed.
3. When the user enters a value, `$REPLY` is set to that value.

4. If `$REPLY` contains a number of a displayed item, the variable specified by *name* is set to the item in *list1* that was selected. Otherwise, the items in *list1* are displayed again.
5. When a valid selection is made, *list2* executes.
6. If *list2* does not exit from the `select` loop using one of the loop control mechanisms such as `break`, the process starts over at Step 1.

If the user enters more than one valid value, `$REPLY` contains all the user's choices. In this case, the variable specified by *name* is not set.

An Example of the `select` Loop

One common use of the `select` loop is in scripts that configure software. The following example is a simplified version of one such script. The actual configuration commands have been omitted because they are not relevant in this discussion.

```
select COMPONENT in comp1 comp2 comp3 all none
do
    case $COMPONENT in
        comp1|comp2|comp3) CompConf $COMPONENT ;;
        all) CompConf comp1
            CompConf comp2
            CompConf comp3
            ;;
        none) break ;;
        *) echo "ERROR: Invalid selection, $REPLY." ;;
    esac
done
```

The menu presented by the `select` loop looks like the following:

```
1) comp1
2) comp2
3) comp3
4) all
5) none
#?
```

As you can see, each of the items in the list

```
comp1 comp2 comp3 all none
```

are displayed with a number preceding them. The user can enter one of these numbers to select a particular component. If a valid selection is made, the `select` loop executes a case statement contained in its body. This case statement performs the correct action based on the user's input. Here the correct action is either calling a command named `CompConf`, exiting the loop, or displaying an error message.



If you are using `zsh`, the output from the previous example will be slightly different than shown above. In `zsh` the menu items are listed on a single line rather than being listed on separate lines:

```
1) comp1  2) comp2  3) comp3  4) all  5) none
?#
```

As you can see, the prompt is still listed on a separate line.

Changing the Prompt

You can change the prompt displayed by the `select` loop by altering the variable `PS3`. If `PS3` is not set, the default prompt, `#?`, is displayed. For example, the commands

```
$ PS3="Please make a selection => " ; export PS3
```

change the menu displayed in the previous example to the following:

```
1) comp1
2) comp2
3) comp3
4) all
5) none
Please make a selection =>
```

Notice that the value of `PS3` used has a space as its last character. This ensures that user input does not run into the prompt and thus makes the menu user friendly.



Versions of `bash` prior to 2.0 do not always use the value stored in `PS3` for the prompt of the `select` loop. This problem is not present in `ksh` or `zsh`.

Loop Control

So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you will need to stop a loop or skip iterations of the loop. In this section you'll examine the `break` and `continue` commands that are used to control the execution of loops.

Infinite Loops and the `break` Command

Recall that the `while` loop terminated when a particular condition was met. This happened when the task of the `while` loop completed. If you make a mistake in specifying

the termination condition of a `while` loop, it can continue forever. For example, say you forgot to specify the `$` before the `x` in the test expression:

```
x=0
while [ x -lt 10 ]
do
    echo $x
    x=`expr $x + 1`
done
```

This loop would continue to display numbers forever. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called *infinite* loops.

In most cases infinite looping is not desired and stems from programming errors, but in certain instances they can be useful. For example, say that you need to wait for a particular event, such as someone logging on to a system. You can use an infinite loop to check every few seconds whether the event has occurred. Because you don't know how many times you need to execute the loop, you need to exit the infinite loop using the `break` command. The `break` command terminates or breaks a loop.

You can create infinite loops using the `while` loop by specifying `cmd` as either `:` or `/bin/true`. The basic syntax of the infinite `while` loop is

```
while :
do
    list
done
```

In most infinite loops, the `while` loop usually exits from within `list` via a `break` command.

Consider the following interactive script that reads and executes commands:

```
while :
do
    read CMD
    case $CMD in
        [qQ][[qQ][uU][iI][tT]) break ;;
        *) $CMD ;;
    esac
done
```

In this loop a command is read at the beginning of each iteration. If that command is either `q` or `Quit`, the loop exits; otherwise, the loop tries to process the command.

Breaking Out of Nested Loops

The `break` command also accepts as an argument an integer, greater than or equal to 1, indicating the number of levels to break out of. This feature is useful in nested loops. Consider the following nested `for` loops:

```
for i in 1 2 3 4 5
do
  mkdir -p /mnt/backup/docs/ch0${i}
  if [ $? -eq 0 ] ; then
    for j in doc c h m pl sh
    do
      cp $HOME/docs/ch0${i}/*.${j} /mnt/backup/docs/ch0${i}
      if [ $? -ne 0 ] ; then break 2 ; fi
    done
  else
    echo "Could not make backup directory."
  fi
done
```

In this loop, you are making a backup of several important files to a backup directory. The outer loop takes care of creating the backup directory, whereas the inner loop copies the important files based on the extension. In the inner loop, you use a `break` command with the argument 2. This indicates that if an error occurs while copying, both loops should be terminated, rather than just the inner loop.



The previous example makes copies of several files in your home directory `$HOME` and places them in a directory under `/mnt`. Please ensure that you are permitted (by your system administrator) to copy files from your home directory to `/mnt` before executing this example.

The `continue` Command

The `continue` command is similar to the `break` command, except that it causes only the current iteration of the loop to exit, rather than the entire loop. This command is useful when an error has occurred but you want to try to execute the next iteration of the loop. As an example, the following loop doesn't exit if one of the input files is bad:

```
for FILE in $FILES ;
do
  if [ ! -f "$FILE" ] ; then
    echo "ERROR: $FILE is not a file."
    continue
  fi
  # process the file
done
```

If one of the filenames in `$FILES` is not a file, this loop skips it, rather than exiting.

Summary

Loops allow you to execute sets of commands repeatedly. In this chapter, you have examined the following types of loops:

- while
- until
- for
- select

You have also examined the concept of nested loops, infinite loops, and loop control. The next chapter introduces the concept of parameters, which require extensive use of loops.

Questions

1. What changes are required to the following `while` loop

```
x=0
while [ $x -lt 10 ]
do
    echo "$x \c"
    y=$((x-1))
    x=$((x+1))
    while [ $y -ge 0 ] ; do
        y=$((y-1))
        echo "$y \c"
    done
    echo
done
```

so that the output looks like the following:

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9
```

2. Write a `select` loop that lists each file in the current directory and enables the user to view the file by selecting its number. In addition to listing each file, use the string `Exit` Program as the key to exit the loop. If the user selects an item that is not a regular file, the program should identify the problem. If no input is given, the menu should be redisplayed.

Terms

Body The set of commands executed by a loop.

Infinite loops Loops that execute forever without terminating. See *loops*.

Iteration A single execution of the body of a loop.

Loops Enable you to execute a series of commands multiple times. Two main types of loops are the `while` and `for` loops.

Nested loops When a loop is located inside the body of another loop, it is said to be nested within another loop.

HOUR 13



Parameters

As you saw in previous chapters, the general format for the invocation of programs in UNIX is

```
cmd opts files
```

Here *cmd* is the command name, *opts* is any option that you need to specify, and *files* is an optional list of files on which the command should operate. Consider the following example:

```
$ ls -l *.doc
```

Here *ls* is the command, *-l* is the only option, and **.doc* is the list of files for *ls* to operate on.

Because most UNIX users are familiar with this interface, it is best to use this format in shell scripts. This means that scripts that can have options must be able to read and interpret them correctly.

This chapter examines the following topics related to the handling of options passed to a shell script:

- Special variables related to option parsing and command execution
- Handling options manually using a case statement
- Handling options using the `getopts` command

For scripts that support only one or two options, the first method is easy to implement and works quite well, but many scripts allow any combination of several options to be given. For such scripts, the `getopts` command is very useful because it affords the maximum flexibility in parsing options.

Special Variables

The shell defines several special variables that are relevant to option parsing and command execution. Table 13.1 describes all of these special variables.

TABLE 13.1 Special Shell Variables

<i>Variable</i>	<i>Description</i>
<code>\$0</code>	The name of the command being executed. For shell scripts, this is the path which invoked it.
<code>\$n</code>	These variables correspond to the arguments with which a script was invoked. Here <i>n</i> is a positive decimal number corresponding to the position of an argument (the first argument is <code>\$1</code> , the second argument is <code>\$2</code> , and so on).
<code>\$#</code>	The number of arguments supplied to a script.
<code>*\$</code>	All the arguments are double quoted. If a script receives two arguments, <code>*\$</code> is equivalent to <code>\$1 \$2</code> .
<code>@\$</code>	All the arguments are individually double quoted. If a script receives two arguments, <code>@\$</code> is equivalent to <code>\$1 \$2</code> .
<code> \$?</code>	The exit status of the last command executed.
<code> \$\$</code>	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
<code> \$!</code>	The process number of the last background command.

Using `$0`

Let's start by looking at `$0`. This variable is commonly used to determine the behavior of scripts that can be invoked with more than one name. Consider the following script:

```
#!/bin/sh
case $0 in
  *listtar) TARGS="-tvf $1" ;;
  *maketar) TARGS="-cvf $1.tar $1" ;;
esac
tar $TARGS
```

You can use this script to list the contents of a tar file (short for tape archive, a common format for distributing files in UNIX) or to create a tar file based on the name with which the script is invoked. The tar file to read or create is specified as the first argument, \$1, to the script.

Let's call this script `mytar` and make two symbolic links to it, called `listtar` and `maketar`, as follows:

```
$ ln -s mytar listtar
$ ln -s mytar maketar
```

If the script is invoked with the name `maketar` and is given a directory or filename, it creates a tar file with the contents of that directory or file. Say you have a directory called `fruits` with the following contents:

```
$ ls fruits
apple  banana  mango  peach  pear
```

You can invoke the script as `maketar` to obtain a tar file called `fruit.tar` containing this directory, by issuing the following command:

```
$ ./maketar fruits
```

If you want to list the contents of the tar file this command creates, you can invoke the script as follows:

```
$ ./listtar fruits.tar
rwxr-xr-x 500/100      0 Nov 17 08:48 1998 fruits/
rw-r--r-- 500/100      0 Nov 17 08:48 1998 fruits/apple
rw-r--r-- 500/100      0 Nov 17 08:48 1998 fruits/banana
rw-r--r-- 500/100      0 Nov 17 08:48 1998 fruits/mango
rw-r--r-- 500/100      0 Nov 17 08:48 1998 fruits/pear
rw-r--r-- 500/100      0 Nov 17 08:48 1998 fruits/peach
```

The exact output depends on the version of `tar` on your system. Some versions include more detailed output than is shown here.

Usage Statements

Another common use for \$0 is in the *usage statement* of a script. The usage statement is a short message that a script outputs in order to inform a user of the proper invocation syntax for the script. All scripts used by more than one user should include usage statements.

In general, the usage statement is something like the following:

```
echo "Usage: $0 [options][files]"
```


Returning to the `mytar` script, adding a usage statement would be a helpful, especially if the script is executed with a name other than `listtar` or `maketar`. You can implement this change as follows:

```
case $0 in
  *listtar) TARGS="-tvf $1" ;;
  *maketar) TARGS="-cvf $1.tar $1" ;;
  *) echo "Usage: $0 [file|directory]"
     exit 0
     ;;
esac
```

Now if the script is invoked as say, `mytar`, it will output the following message:

```
Usage: mytar [file|directory]
```

Although this message describes the usage of the script correctly, it does not inform you that the script's name was given incorrectly. There are two possible methods for rectifying this:

- Hard coding the valid names in the usage statement
- Changing the script to use its arguments to decide in which mode it should run

To demonstrate the use of options, the next section uses the latter method.

Options and Arguments

Options are given on the command line to change the behavior of a script or program. For example, the `-a` option of the `ls` command changes the behavior of the `ls` command from listing all visible files to listing all files (as explained in Chapter 3). This section shows you how to use options to change the behavior of scripts.

Often you will see or hear options called *arguments*. The difference between the two is subtle. A command's arguments are all of the separate strings or words that appear on the command line after the command name, whereas *options* are only those arguments that change the behavior of the command.

For example, in the following example:

```
$ ls -aF fruit
```

the command is `ls`, and its arguments are `-aF` and `fruit`. The options to the `ls` command are `-aF`.

Dealing with Arguments

To illustrate the use of options, let's change the `mytar` script to use its first argument, `$1`, as the mode argument and `$2` as the tar file to read or create. You can implement this change as follows:

```
USAGE="Usage: $0 [-c|-t] [file|directory]"
case "$1" in
  -t) TARGS="-tvf $2" ;;
  -c) TARGS="-cvf $2.tar $2" ;;
  *) echo "$USAGE"
     exit 0
     ;;
esac
```

The three main changes are as follows:

- All references to `$1` have been changed to `$2` because the second argument is now the filename.
- `listtar` has been replaced by `-t`.
- `maketar` has been replaced by `-c`.

Now running `mytar` produces the following output:

```
Usage: ./mytar [-c|-t] [file|directory]
```

To create a tar file of the directory `fruits` with this version, use the command

```
$ ./mytar -c fruits
```

To list the contents of the resulting tar file, `fruits.tar`, use the command

```
$ ./mytar -t fruits
```

Using `basename`

Currently, the usage statement of `mytar` outputs the entire path with which the script was invoked. What it should really output is just the name of the script. You can correct this by using the `basename` command.

The `basename` command takes an absolute or relative pathname to a file or directory and returns just the file or directory name from that path. The basic syntax is as follows:

```
basename file
```

For example,

```
$ basename /usr/bin/sh
```

prints the following:

```
sh
```

Using `basename`, you can change the variable `$USAGE` in the `mytar` script as follows:

```
USAGE="Usage: `basename $0` [-c|-t] [file|directory]"
```

so that the usage statement produces the desired output:

```
Usage: mytar [-c|-t] [file|directory]
```

You can also use the `basename` command in the first version of the `mytar` script to avoid using the `*` wildcard character in the case statement:

```
#!/bin/sh
case `basename $0` in
  listtar) TARGS="-tvf $1" ;;
  maketar) TARGS="-cvf $1.tar $1" ;;
  *) echo "Usage: $0 [file|directory]"
     exit 0
     ;;
esac
tar $TARGS
```

In this version, `basename` allows you to match the exact names with which scripts can be called. As an illustration of a potential problem with the original version, you can see that if the script is called

```
$ ./makelisttar
```

the original version would use the first case statement, even though it was incorrect, but the new version would fall through and report an error.

Emulating `basename`

Some older Linux and BSD systems do not include the `basename` command. If you are using such a system, you can emulate this command by creating a shell script that provides the equivalent functionality. A shell script version of `basename` might look like the following:

```
#!/bin/sh

if [ -n "$1" ] ; then
  echo "$1" | sed -e 's/^\.*\///'
else
  echo "Usage: basename [file]" 1>&2
  exit 1
fi

exit 0
```

Don't worry if you don't understand how the `sed` command works. You'll read more about it in Chapter 16, "Filtering Text with Regular Expressions."

Common Argument Handling Problems

Now that the `mytar` script uses options to set the mode in which the script will run, you need to solve another problem. Namely, what should you do if the second argument, `$2`, is not provided? You don't have to worry about what happens if the first argument, `$1`, is not provided because the case statement deals with this situation via the default case, `*`.

The simplest method for checking the necessary number of arguments is to see whether the number arguments, `$#`, match the number of required arguments. You can add this check to the script as follows:

```
#!/bin/sh

USAGE="Usage: `basename $0` [-c|-t] [file|directory]"

if [ $# -lt 2 ] ; then
    echo "$USAGE"
    exit 1
fi

case "$1" in
    -t) TARGS="-tvf $2" ;;
    -c) TARGS="-cvf $2.tar $2" ;;
    *) echo "$USAGE"
       exit 0
       ;;
esac

tar $TARGS
```

Handling Additional Files

The `mytar` script is mostly finished, but you can still make a few improvements. For example, it only deals with the first file that is given as an argument, and it does not determine whether this argument is really a file.

You can add the processing of all file arguments by using the special shell variable `$@`. Let's start by modifying the `-t` option to work with this variable:

```
case "$1" in
    -t) TARGS="-tvf"
       for i in "$@"
       do
           if [ -f "$i" ] ; then
               tar $TARGS "$i"
           fi
       done
       ;;
    -c) TARGS="-cvf $2.tar $2" ;
```

```
        tar $TARGS
        ;;
    *) echo "$USAGE" ;
       exit 0
       ;;
esac
```

The main changes are

- The `-t` case now includes a `for` loop that processes the arguments.
- There is an `if` statement in the `for` loop that determines whether the argument is a file. If an argument is a file, `tar` is executed on that file.

\$* and \$@

The arguments specified to a shell script are stored in two special variables, `$*` and `$@`. The main difference between these two special variables is how they store arguments: `$*` stores each argument without preserving quoting, whereas `$@` stores each argument by preserving quoting.

The behavior of `$*` can sometimes cause a problem. For example, if your script has a file-name containing spaces as an argument:

```
mytar -t "my tar file.tar"
```

using `$*` instead of `$@` would create a problem because the `for` loop would be executed three times for files named `my`, `tar`, and `file.tar`, instead of just once for the file you requested, `my tar file.tar`. By using `$@`, you avoid this problem because each argument is stored as it was quoted on the command line.

A Few Minor Issues

There are two minor issues in `mytar` that you should deal with:

- `mytar` treats all its arguments, including the first argument, `$1`, as files. Because you are using the first argument to indicate the mode in which the script runs you should not consider it as a file. This will reduce the number of times the `for` loop is executed, and will prevent the script from trying to run `tar` on a file with the name `-t`.
- Another issue involves what the script should do when an operation fails. In the case of the list operation, if `tar` cannot list the contents of a file, you should skip the file and print an error.

You can solve the first issue by using `shift` to remove the first argument. You can solve the second issue by using the variable `$?` to check the exit status of `tar`. If you implement these changes, your script becomes:

```
#!/bin/sh

USAGE="Usage: `basename $0` [-c|-t] [files|directories]"

if [ $# -lt 2 ] ; then
    echo "$USAGE" ;
    exit 1 ;
fi

case "$1" in
    -t) shift
        TARGS="-tvf" ;
        for i in "$@" ;
        do
            if [ -f "$i" ] ; then
                FILES=`tar $TARGS "$i" 2>/dev/null`
                if [ $? -eq 0 ] ; then
                    echo ; echo "$i" ; echo "$FILES"
                else
                    echo "ERROR: $i not a tar file."
                fi
            else
                echo "ERROR: $i not a file."
            fi
        done
        ;;
    -c) shift
        TARGS="-cvf" ;
        tar $TARGS archive.tar "$@"
        ;;
    *) echo "$USAGE"
        exit 0
        ;;
esac
exit $?
```

Option Parsing in Shell Scripts

In the previous example, you manually handled the options passed to your script. In this second example, you will explore a second method, using the `getopts` command. The syntax of the `getopts` command is as follows:

```
getopts option-string var
```

Here *option-string* is a string consisting of all the single character options `getopts` should consider and *var* is the name of the variable that the option should be set to. Usually *var* is a variable named `OPTION`.

The process by which `getopts` parses the options given on the command line is as follows:

1. `getopts` examines all the command-line arguments, looking for arguments starting with the `-` character.
2. When an argument starting with the `-` character is found, it compares the characters following the `-` to the characters given in the *option-string*.
3. If a match is found, the specified *var* is set to the option; otherwise, *var* is set to the `?` character.
4. Steps 1 through 3 are repeated until all the options have been considered.
5. When parsing has finished, `getopts` returns a nonzero exit code. This allows it to be easily used in loops. Also, when `getopts` has finished, it sets the variable `OPTIND` to the index of the last argument.

Another feature of `getopts` is its capability to indicate options requiring an additional parameter. This can be accomplished by following the option with a colon, `:`, character. In this case, after an option is parsed, the additional parameter is set to the value of the variable named `OPTARG`.



Some early versions of `bash` (1.x) did not completely support the `getopts` command. If you are using an older version of `bash`, you might encounter some errors when executing the examples in this section. If possible, you should upgrade to `bash` 2.0 or newer or use an alternate shell such as `ksh` or `zsh` when executing these examples.

Using `getopts`

To get a feeling for how `getopts` works and how to deal with options, you will write a script that simplifies the process of uuencoding a file.

For readers who are not familiar with `uuencode`, it is a program that was originally used to encode binary files (executable files) into ASCII text so that they could be e-mailed or transferred via FTP. Today, MIME encoding has taken the place of uuencoding for e-mail attachments, but it is still used for posting binaries to newsgroups and transferring binaries via modem.

You'll first examine the interface of this script, which makes it easier to understand the implementation. Your script should be able to accept the following options:

- -f to indicate the input filename
- -o to indicate the output filename
- -v to indicate the script should be verbose

The `getopts` command to implement these requirements is

```
getopts e:o:v OPTION
```

This indicates that all the options except for `-v` require an additional parameter. The support variables that are required are

- `VERBOSE`, which stores the value of the verbose flag. By default, the value of this variable is `false`.
- `INFILE`, which stores the name of the input file.
- `OUTFILE`, which stores the name of the output filename. If this value is unset, `uudecode` uses the same name as the input file and appends the `.uu` extension to it.

The following loop implements these requirements:

```
VERBOSE=false
while getopts f:o:v OPTION ;
do
    case "$OPTION" in
        f) INFILE="$OPTARG" ;;
        o) OUTFILE="$OPTARG" ;;
        v) VERBOSE=true ;;
        \?) echo "$USAGE" ;
            exit 1
            ;;
    esac
done
```

Now that you have dealt with option parsing, you need to deal with error conditions. For example, what should your script do if the input file is not specified? The simplest behavior would be to exit with an error, but with a little more work, you can make the script much more user friendly.

By using the fact that `getopts` sets the variable `OPTIND` to the value of the last option that was scanned, you can have the script assume that the first argument after this is the input filename. If no additional arguments remain, you should exit. Your error checking can be implemented as follows:

```
shift `echo "$OPTIND - 1" | bc`
if [ -z "$1" -a -z "$INFILE" ] ; then
```



```

        echo "ERROR: Input file was not specified."
        exit 1
    fi

    if [ -z "$INFILE" ] ; then
        INFILE="$1"
    fi

```

Here you use the `shift` command to discard the arguments given to the script by one minus the last argument processed by `getopts`. The exact number of arguments to shift is calculated by the `bc` command, which is a command-line calculator. Its usage is explained in detail in Chapter 18. Strictly speaking, you do not have to shift the arguments; it just simplifies the `if` statement.

After shifting the arguments, you need to check whether the new `$1` contains a value. If it does not contain a value, you output an error message and exit; otherwise, you set `INFILE` to the filename specified by `$1`.

You also need to set the output filename, in case the `-o` option was not specified. You can use variable substitution to accomplish this

```

: ${OUTFILE:=${INFILE}.uu}

```

Here you set the name of the output file to the input file plus the `.uu` extension, if an output file is not given. You use the `:` command to prevent the shell from trying to execute the result of the variable substitution.

After you have made sure that all the inputs are correct, the actual work is quite simple. The `uuencode` command that you use is as follows:

```

uuencode $INFILE $INFILE > $OUTFILE ;

```

You should also check whether the input file is really a file before doing this command, so the actual body of your program is:

```

if [ -f "$INFILE" ] ; then
    uuencode "$INFILE" "$INFILE" > "$OUTFILE" ;
fi

```

At this point the script is fully functional, but you still need to add support for verbose reporting. This changes the preceding `if` statement to the following:

```

if [ -f "$INFILE" ] ; then
    if [ "$VERBOSE" = "true" ] ; then
        echo "uuencoding $INFILE to $OUTFILE... \c"
    fi
    uuencode "$INFILE" "$INFILE" > "$OUTFILE"
    RET=$?
    if [ "$VERBOSE" = "true" ] ; then

```

```

        MSG="Failed"
        if [ $RET -eq 0 ] ; then
            MSG="Done."
        fi
        echo $MSG
    fi
fi

```

You could simplify the verbose reporting to print a statement after the uuencode completes, but issuing two statements, one before the operation starts and one after the operation completes, is much more user-friendly. This method clearly indicates that the operation is being performed.

The complete script is as follows:

```

#!/bin/sh

USAGE="Usage: `basename $0` [-v] [-f] [filename] [-o] [filename]";
VERBOSE=false

while getopts f:o:v OPTION ;
do
    case "$OPTION" in
        f) INFILE="$OPTARG" ;;
        o) OUTFILE="$OPTARG" ;;
        v) VERBOSE=true ;;
        \?) echo "$USAGE"
            exit 1
            ;;
    esac
done

shift `echo "$OPTIND - 1" | bc`

if [ -z "$1" ] && [ -z "$INFILE" ] ; then
    echo "ERROR: Input file was not specified."
    exit 1
fi
if [ -z "$INFILE" ] ; then
    INFILE="$1"
fi

: ${OUTFILE:=${INFILE}.uu}

if [ -f "$INFILE" ] ; then
    if [ "$VERBOSE" = "true" ] ; then
        echo "uuencoding $INFILE to $OUTFILE... \c"
    fi
    uuencode $INFILE $INFILE > $OUTFILE
    RET=$?

```

```
if [ "$VERBOSE" = "true" ] ; then
    MSG="Failed"
    if [ $RET -eq 0 ] ; then
        MSG="Done."
    fi
    echo $MSG
fi
fi
exit 0
```

Assuming this script is called `uu`, you can use it to uuencode files in all of the following ways:

```
uu ch13.doc
uu -f ch13.doc
uu -f ch13.doc -o ch13.uu
```

In each of the preceding commands, file `ch13.doc` is uuencoded. The last command places the result into the file `ch13.uu` instead of the default `ch13.doc.uu`; this might be required if the document needs to be used on a DOS or Windows system.

Because this script uses `getopts`, any of the commands given previously can run in verbose mode by simply specifying the `-v` option.

Summary

In this chapter, you examined how to deal with arguments and options in shell script. Specifically you looked at the following methods:

- Manually handling arguments and options using a case statement
- Handling options using `getopts`

You worked through two examples that illustrate the implementation and rationale behind each method. In addition, you saw several special variables that pertain to arguments and command execution.

As you will see in later chapters, using options greatly increases the flexibility and the reusability of your shell scripts.

Questions

1. Add `tar` file extraction to the `mytar` script.

Assume that the `-x` option indicates that the user wants to extract `tar` files and that the correct value of `TAR_ARGS` for extracting `tar` files is `-xvf`.

2. Add the `extract` option to the `uu` script. Assume that the `-x` option indicates that the file should be extracted, and that the command

```
uudecode $INFILE
```

is used to extract a uuencoded file.

Terms

Usage statement A short message that a script outputs in order to inform a user of the proper invocation syntax for the script.

This page intentionally left blank

HOUR 14



Functions

Shell functions provide a way of mapping a name to a list of commands. Functions are similar to subroutines and procedures in other programming languages. You can also think of them as miniature shell scripts, complete with exit codes and arguments. The main difference between a script and a function is that a new instance of the shell is started for a shell script, whereas functions run in the current instance of the shell.

This chapter is divided into the following two sections:

- Using functions
- Understanding scope, recursion, return codes, and data sharing

The first section introduces the syntax for defining functions and illustrates their use, whereas the second section covers more advanced topics relating to the interaction of scripts and functions.

Using Functions

Functions are defined as follows:

```
name () { list ; }
```

Here, *name* is the name of the function and *list* is a list of commands. The list of commands, *list*, is referred to as the body of the function. The parentheses, (and), that follow *name* are required.

The job of a function is to bind *name* to *list*, so that whenever *name* is specified *list* is executed. When a function is defined, *list* is not executed; the shell parses *list* to ensure that there are no syntax errors and stores *name* in its list of commands. The following example illustrates a basic function definition:

```
ls1() { ls -l ; }
```

Here you define the function `ls1` and specify *list* as `ls -l`.

An alternative form of function definition is available in `ksh`, `bash`, and `zsh`:

```
function name { list ; }
```

Here, *name* is the name of the function and *list* is the list of commands to be executed. This form of function definition is not available in the Bourne shell. Scripts that need to be ported to older systems should not use this form for function definition.

Executing Functions

You can execute or call a function that has been defined by specifying its *name*. For example, you can execute the function `ls1`, defined in the previous example, as follows:

```
$ ls1
```

This causes the shell to execute the body of the function, in this case the command `ls -l`, and output the result. The output will be similar to the following:

```
total 6
drwxrwxrwt  3 root  wheel  512 Oct 29 08:59 ./
drwxr-xr-x  25 root  wheel  512 Oct 29 00:02 ../
drwxrwxrwt  2 root  wheel  512 Nov  3 17:49 vi.recover/
```

Functions are normally defined on the command line or within a script. Once defined, the function acts as a valid command in all the sub-shells started by that shell or script. For example, if you enter the command:

```
$ ls1() { ls -l ; }
```

The function `ls1` becomes a valid command name that can be accessed by specifying `ls1`. It is accessible in sub-shells as well:

```
$ ( ls1 )
total 6
drwxrwxrwt  3 root  wheel  512 Oct 29 08:59 ./
```

```
drwxr-xr-x 25 root wheel 512 Oct 29 00:02 ../
drwxrwxrwt  2 root wheel 512 Nov  3 17:49 vi.recover/
```

A function defined in a script is accessible within that script and any sub-shells started by that script. For example, consider the following script:

```
#!/bin/sh
lsl() { ls -l ; }
cd "$1" && lsl
```

The function `lsl` is only available in that script.

Arguments

Just as you can execute commands with arguments, you can also execute functions with arguments. The general syntax for invoking a function is as follows:

```
name arg1 ... argN
```

Here, *name* is the name of the function and *arg1 ... argN* are the arguments to the function. The arguments specified to a function are accessed in the same way as arguments specified to a shell script; the individual arguments are available as `$1`, `$2`, and so on, whereas the set of all the arguments is available as `$@`.

The following function illustrates the use of individual arguments:

```
printMsg () { echo "$1: $2" ; }
```

This function uses `echo` to print a message with a colon, `:`, which separates the first two arguments when it's executed as follows:

```
printMsg Error Failed
```

the output is

```
Error: Failed
```

As defined, this function can handle only two arguments; it ignores all the others. In order to make the function a bit more useful, it needs to be able to handle an arbitrary number of arguments. Because all of the arguments specified to a function are available in the variable `$@`, you can use it as follows:

```
printMsg() {
    PREFIX="$1"
    shift
    echo "$PREFIX: $@"
}
```

Here, you have redefined the function `printMsg`. It saves its first argument in `$PREFIX` and then uses `echo` to print the message in the desired format. You use `shift` to remove

the first argument from `$@` before calling `echo`. Now you can execute the function with any number of arguments and the message will be printed properly. For example, if `printMsg` is executed as follows:

```
printMsg Info All Quiet on the Western Front
```

the output is

```
Info: All Quiet on the Western Front
```

Function Chaining

Function chaining is the process of calling a function from another function. The following script illustrates function chaining:

```
#!/bin/sh

orange () {
    echo "Now in orange"
    banana          # call func2()
}

banana () {
    echo "Now in banana"
}

orange
```

This script defines two functions, `orange` and `banana`, and then executes `orange`. The first function, `orange`, outputs a message and then calls the function `banana`. The second function, `banana`, just outputs a message. The output from this script is

```
Now in orange
Now in banana
```

Common Errors

Two common errors with declaring and using functions are

- Omitting the parentheses, `()`, in a function definition.
- Specifying the parentheses, `()`, in a function invocation.

The following example illustrates the first type of error:

```
ls1 { ls -l ; }
```

Here, the parentheses are missing after `ls1`. This is an invalid function definition and will result in an error message similar to the following:

```
sh: syntax error: '}' unexpected
```

The following command illustrates the second type of error:

```
$ ls1()
```

Here, the function `ls1` is executed along with the parentheses, `()`. This will not work because the shell interprets it as a redefinition of the function with the name `ls1`. Usually such an invocation results in a prompt similar to the following:

```
>
```

This is a prompt produced by the shell when it expects you to provide more input. The input it expects is the body of the function `ls1`.

Aliases Versus Functions

An *alias* is an abbreviation or an alternative name, usually mnemonic, for a command. Aliases were first introduced in `csh` and were later adopted by `ksh`, `bash`, and `zsh`. They are not supported in the Bourne shell.

Aliases are defined using the `alias` command:

```
alias name="cmd"
```

Here *name* is the name of the alias and *cmd* is the command to execute when *name* is specified. Aliases are similar to functions in that they associate a command with a name. Two key differences are

- In an alias, *cmd* cannot be a compound command or a list.
- In an alias, there is no way to manipulate the argument list (`$@`).

Due to their limited capabilities, aliases are not commonly used in shell programs. They are discussed here for the sake of completeness.

As an example, the following command defines the alias `ls1` and specifies that the command `ls -l` should be executed when the command `ls1` is specified:

```
alias ls1="ls -l"
```

This alias is equivalent to the function:

```
ls1 () { ls -l "$@" ; }
```

A common use for aliases is to specify a default set of options to a command. For example, say you have the following alias:

```
alias ls="ls -a"
```

When the `ls` command is given, the shell executes `ls -a` instead of plain `ls` without options. It is possible to mimic this behavior with a function such as:

```
name () { path "$@" ; }
```

Here, *name* is the name of the command to be “aliased” and *path* is the fully qualified path to the command. For example, the following function is equivalent to the alias given in the previous example:

```
ls () { /bin/ls -a "$@" ; }
```

Unalias

Once an alias has been defined, it can be unset using the `unalias` command:

```
unalias name
```

Here, *name* is the name of the alias to be unset. For example, the following command unsets the alias `ls1`:

```
unalias ls1
```

Unsetting Functions

Once a function has been defined, it can be undefined via the `unset` command:

```
unset name
```

Here, *name* is the name of the function you want to unset. For example, the following command unsets the previously defined function `ls1()`:

```
unset ls1
```

After a function has been unset it cannot be executed.

Understanding Scope, Recursion, Return Codes, and Data Sharing

Now that you have a basic understanding of the use and operation of functions in shell scripts, let’s look at more advanced topics such as scope, recursion, return codes, and data sharing.

Scope

The term *scope* refers to the region within a program where a variable’s value can be accessed. There are two types of scope:

- *Global scope* If a variable has global scope, its value can be accessed from anywhere within a script. Variables with global scope are referred to as *global variables*.
- *Local scope* If a variable has local scope, its value can only be accessed within the function in which it is declared. Variables with local scope are referred to as *local variables*.

By default all variables, except for the special variables associated with function arguments, have global scope. In `ksh`, `bash`, and `zsh`, variables with local scope can be declared using the `typeset` command. The `typeset` command is discussed later in this chapter. This command is not supported in the Bourne shell, so it is not possible to have programmer-defined local variables in scripts that rely strictly on the Bourne shell.

Global Variables

The following script illustrates the behavior of global variables:

```
#!/bin/sh

pearFunc () {
    pear=2;                # set $pear
    echo "In pearFunc(): pear is $pear" # print out its value
}

pearFunc                  # call pearFunc
echo "Outside of pearFunc(): pear is $pear" # print out $pear
```

First the script defines a function, `pearFunc`, that sets the value of the global variable `$pear` (all variables are global by default) and outputs that value. Then the script executes `pearFunc`. Finally, the script prints the value of `$pear` outside of the function. The output is

```
In pearFunc(): pear is 2
Outside of pearFunc(): pear is 2
```

As you can see from the output, the value assigned to the variable `$pear` in the function `pearFunc` is accessible outside of `pearFunc`.

A common use for global variables is to communicate information from a function to the main script, as illustrated in the following script:

```
#!/bin/sh

readPass () {
    PASS=""                # clear password
    echo -n "Enter Password: " # print the prompt
    stty -echo             # turn off terminal echo to prevent peeping!
    read PASS              # read the password
    stty echo              # restore terminal echo
    echo                   # printout a new line to make output nice
}

readPass
echo Password is $PASS
```

This script uses the `readPass` function to read in a password from the user. The `readPass` function reads the password and stores it in the global variable `PASS`. The script then accesses the password using the variable `PASS`.

The `readPass` function is quite simple. It function starts by undefining `PASS`. Then it issues a prompt for the password and deactivates terminal echo using the `stty -echo` command. Terminal echo is deactivated because you don't want someone other than the user to inadvertently see the password. Next, you read the password and store its value in `PASS` by using the `read` command. Finally, you restore terminal echo using the `stty echo` command and echo a new line.

Local Variables

Local variables are defined using `typeset` command:

```
typeset var1[=val1] ... varN[=valN]
```

Here, `var1 ... varN` are variable names and `val1 ... valN` are values to assign to the variables. The values are optional as the following example illustrates:

```
typeset fruit1 fruit2=banana
```

This command declares two local variables, `fruit1` and `fruit2`, and assigns the value `banana` to the variable `fruit2`.

The following script illustrates the behavior of local variables:

```
#!/bin/sh

pearFunc () {
    typeset pear=2;                # set $pear
    echo "In pearFunc(): pear is $pear" # print out its value
}

pearFunc                          # call pearFunc
echo "Outside of pearFunc(): pear is $pear" # print out $pear
```

First, the script defines a function, `pearFunc`, which sets the value of a local variable `$pear` and outputs that value. Then the script executes `pearFunc`. Finally, the script prints the value of `$pear` outside of the function. The output is

```
In pearFunc(): pear is 2
Outside of pearFunc(): pear is
```

From the output, you can see that when the value of `$pear` is accessed within the `pearFunc` it has the value 2, but when the value of `$pear` is accessed outside the function, it has no value.

