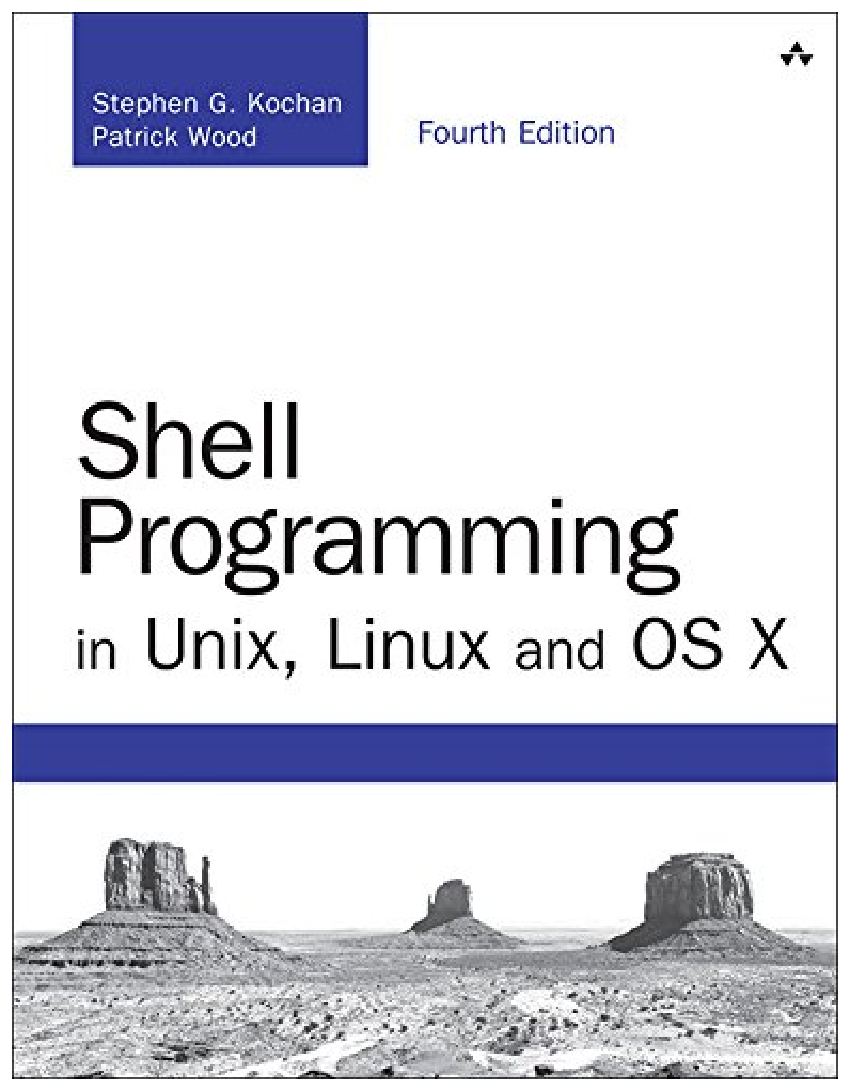
Shell Programming in Unix, Linux and OS X (Developer's Library)

by

Linda Stevens



Synopsis

Shell Programming in Unix, Linux and OS X is a thoroughly updated revision of Kochan and Wood’s classic Unix Shell Programming tutorial. Following the methodology of the original text, the book focuses on the POSIX standard shell, and teaches you how to develop programs in this useful programming environment, taking full advantage of the underlying power of Unix and Unix-like operating systems.After a quick review of Unix utilities, the book’s authors take you step-by-step through the process of building shell scripts, debugging them, and understanding how they work within the shell’s environment. All major features of the shell are covered, and the large number of practical examples make it easy for you to build shell scripts for your particular applications. The book also describes the major features of the Korn and Bash shells. Learn how to…Take advantage of the many utilities provided in the Unix systemWrite powerful shell scriptsUse the shell’s built-in decision-making and looping constructsUse the shell’s powerful quoting mechanismsMake the most of the shell’s built-in history and command editing capabilitiesUse regular expressions with Unix commandsTake advantage of the special features of the Korn and Bash shellsIdentify the major differences between versions of the shell languageCustomize the way your Unix system responds to youSet up your shell environmentMake use of functionsDebug scriptsContents at a Glance1 A Quick Review of the Basics2 What Is the Shell?3 Tools of the Trade4 And Away We Go5 Can I Quote You on That? 6 Passing Arguments7 Decisions, Decisions8 ‘Round and ‘Round She Goes9 Reading and Printing Data10 Your Environment11 More on Parameters12 Loose Ends13 Rolo Revisited14 Interactive and Nonstandard Shell FeaturesA Shell SummaryB For More Information

Sort review

About the AuthorMajestic Coloring publishes adult coloring books with a variety of difficulty and styles for everyone to enjoy. We include designs with quality, hand-curated and/or manipulated licensed images, as well as original designs from artists around the globe. Get a Free Coloring Book! This 'n That includes 12 fun designs for your coloring enjoyment. All bundled up in one convenient PDF file to download and print at your leisure. Visit our site and download your copy. MajesticColoring.com

Download to continue reading...

Look inside the book

About This E-BookEPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.Shell Programming in Unix, Linux and OS XFourth EditionStephen G. KochanPatrick Wood800 East 96th Street, Indianapolis, Indiana 46240Shell Programming in Unix, Linux and OS X, Fourth EditionCopyright © 2017 by Pearson Education, Inc.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. Nor is any liability assumed for damages resulting from the use of the information contained herein.Printed in the United States of AmericaFirst Printing: August 2016The Library of Congress Control Number is on file.EditorMark TaberCopy EditorLarry SulkyTechnical EditorBrian TiemannDesignerChuti PrasertsithPage LayoutcodeMantraTrademarksAll terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. The publisher 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 DisclaimerEvery 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 author 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.Special SalesFor information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.For government sales inquiries, please contactgovernmentsales@pearsoned.comFor questions about sales outside the U.S., please

contactinternational@pearsoned.comDeveloper's LibraryESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALSDeveloper's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.All books in the Developer's Library are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.Key titles include some of the best, most widely acclaimed books within their topic areas:Developer's Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online atDeveloper's LibraryContents at a GlanceIntroduction1 A Quick Review of the Basics2 What Is the Shell?3 Tools of the Trade4 And Away We Go5 Can I Quote You on That?6 Passing Arguments7 Decisions, Decisions8 'Round and 'Round She Goes9 Reading and Printing Data10 Your Environment11 More on Parameters12 Loose Ends13 Rolo Revisited14 Interactive and Nonstandard Shell FeaturesA Shell SummaryB For More InformationIndexTable of ContentsIntroductionHow This Book Is OrganizedAccessing the Free Web Edition1 A Quick Review of the BasicsSome Basic CommandsDisplaying the Date and Time: The date CommandFinding Out Who's Logged In: The who CommandEchoing Characters: The echo CommandWorking with FilesListing Files: The ls CommandDisplaying the Contents of a File: The cat CommandCounting the Number of Words in a File: The wc CommandCommand OptionsMaking a Copy of a File: The cp CommandRenaming a File: The mv CommandRemoving a File: The rm CommandWorking with DirectoriesThe Home Directory and PathnamesDisplaying Your Working Directory: The pwd CommandChanging Directories: The cd CommandMore on the ls CommandCreating a Directory: The mkdir CommandCopying a File from One Directory to AnotherMoving Files Between DirectoriesLinking Files: The ln CommandRemoving a Directory: The rmdir CommandFilename SubstitutionThe AsteriskMatching Single CharactersFilename NuancesSpaces in FilenamesOther Weird CharactersStandard Input/Output, and I/O RedirectionStandard Input and Standard OutputOutput RedirectionInput RedirectionPipesFiltersStandard ErrorMore on CommandsTyping More Than One Command on a LineSending a Command to the BackgroundThe ps CommandCommand Summary2 What Is the Shell?The Kernel and the UtilitiesThe Login ShellTyping Commands to the ShellThe Shell's ResponsibilitiesProgram ExecutionVariable and Filename SubstitutionI/O RedirectionHooking up a PipelineEnvironment ControlInterpreted Programming Language3 Tools of the TradeRegular ExpressionsMatching Any Character: The Period (.)Matching the Beginning of the Line: The Caret (^)Matching the End of the Line: The Dollar Sign $Matching a Character Set: The [...] ConstructMatching Zero or More Characters: The Asterisk (*)Matching a Precise Number of Subpatterns: \{...\}Saving Matched Characters: \(...\)cutThe -d and -f OptionspasteThe -d OptionThe -s OptionsedThe -n OptionDeleting LinestrThe -s OptionThe -d OptiongrepRegular Expressions and grepThe -v OptionThe -l OptionThe -n OptionsortThe -u OptionThe -r OptionThe -o OptionThe -n OptionSkipping FieldsThe -t OptionOther OptionsuniqThe -d OptionOther Options4 And Away

We GoCommand FilesCommentsVariablesDisplaying the Values of VariablesUndefined Variables Have the Null ValueFilename Substitution and VariablesThe ${variable} ConstructBuilt-in Integer Arithmetic5 Can I Quote You on That?The Single QuoteThe Double QuoteThe BackslashUsing the Backslash for Continuing LinesThe Backslash Inside Double QuotesCommand SubstitutionThe Back QuoteThe $(...) ConstructThe expr Command6 Passing ArgumentsThe $# VariableThe $* VariableA Program to Look Up Someone in the Phone BookA Program to Add Someone to the Phone BookA Program to Remove Someone from the Phone Book${n}The shift Command7 Decisions, DecisionsExit StatusThe $? VariableThe test CommandString OperatorsAn Alternative Format for testInteger OperatorsFile OperatorsThe Logical Negation Operator !The Logical AND Operator -aParenthesesThe Logical OR Operator -oThe else ConstructThe exit CommandA Second Look at the rem ProgramThe elif ConstructYet Another Version of remThe case CommandSpecial Pattern-Matching CharactersThe -x Option for Debugging ProgramsBack to the caseThe Null Command :The && and || Constructs8 'Round and 'Round She GoesThe for CommandThe $@ VariableThe for Without the ListThe while CommandThe until CommandMore on LoopsBreaking Out of a LoopSkipping the Remaining Commands in a LoopExecuting a Loop in the BackgroundI/O Redirection on a LoopPiping Data into and out of a LoopTyping a Loop on One LineThe getopts Command9 Reading and Printing DataThe read CommandA Program to Copy FilesSpecial echo Escape CharactersAn Improved Version of mycpA Final Version of mycpA Menu-Driven Phone ProgramThe $$ Variable and Temporary FilesThe Exit Status from readThe printf Command10 Your EnvironmentLocal VariablesSubshellsExported Variablesexport -pPS1 and PS2HOMEPATHYour Current DirectoryCDPATHMore on SubshellsThe .CommandThe exec CommandThe (...) and { ...; } ConstructsAnother Way to Pass Variables to a SubshellYour .profile FileThe TERM VariableThe TZ Variable11 More on ParametersParameter Substitution${parameter}${parameter:-value}${parameter:=value}${parameter:?value}${parameter:+value}Pattern Matching Constructs ${#variable}The $0 VariableThe set CommandThe -x Optionset with No ArgumentsUsing set to Reassign Positional ParametersThe -- OptionOther Options to setThe IFS VariableThe readonly CommandThe unset Command12 Loose EndsThe eval CommandThe wait CommandThe $! VariableThe trap Commandtrap with No ArgumentsIgnoring SignalsResetting TrapsMore on I/O<&- and >&-In-line Input RedirectionShell ArchivesFunctionsRemoving a Function DefinitionThe return CommandThe type Command13 Rolo RevisitedData Formatting ConsiderationsroloaddludisplayremchangelistallSample Output14 Interactive and Nonstandard Shell FeaturesGetting the Right ShellThe ENV FileCommand-Line EditingCommand HistoryThe vi Line Edit ModeAccessing Commands from Your HistoryThe emacs Line Edit ModeAccessing Commands from Your HistoryOther Ways to Access Your HistoryThe history CommandThe fc CommandThe r CommandFunctionsLocal VariablesAutomatically Loaded FunctionsInteger ArithmeticInteger TypesNumbers in Different BasesThe alias CommandRemoving AliasesArraysJob ControlStopped Jobs and the fg and bg CommandsMiscellaneous FeaturesOther Features of the cd CommandTilde SubstitutionOrder of SearchCompatibility

SummaryA Shell SummaryStartupCommandsCommentsParameters and VariablesShell VariablesPositional ParametersSpecial ParametersParameter SubstitutionCommand Re-entryThe fc Commandvi Line Edit ModeQuotingTilde SubstitutionArithmetic ExpressionsFilename SubstitutionI/O RedirectionExported Variables and Subshell ExecutionThe (...) ConstructThe { ...; } ConstructMore on Shell VariablesFunctionsJob ControlShell JobsStopping JobsCommand SummaryThe : CommandThe . CommandThe alias CommandThe bg CommandThe break CommandThe case CommandThe cd CommandThe continue CommandThe echo CommandThe eval CommandThe exec CommandThe exit CommandThe export CommandThe false CommandThe fc CommandThe fg CommandThe for CommandThe getopts CommandThe hash CommandThe if CommandThe jobs CommandThe kill CommandThe newgrp CommandThe pwd CommandThe read CommandThe readonly CommandThe return CommandThe set CommandThe shift CommandThe test CommandThe times CommandThe trap CommandThe true CommandThe type CommandThe umask CommandThe unalias CommandThe unset CommandThe until CommandThe wait CommandThe while CommandB For More InformationOnline DocumentationDocumentation on the WebBooksO'Reilly & AssociatesPearsonIndexAbout the AuthorsStephen Kochan is the author or co-author of several best-selling titles on Unix and the C language, including Programming in C, Programming in Objective-C, Topics in C Programming, and Exploring the Unix System. He is a former software consultant for AT&T Bell Laboratories, where he developed and taught classes on Unix and C programming.Patrick Wood is the CTO of the New Jersey location of Electronics for Imaging. He was a member of the technical staff at Bell Laboratories when he met Mr. Kochan in 1985. Together they founded Pipeline Associates, Inc., a Unix consulting firm, where he was vice president. They co-authored Exploring the Unix System, Unix System Security, Topics in C Programming, and Unix Shell Programming.We Want to Hear from You!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.We welcome your comments. You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books better.Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we 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 email address.Email: feedback@developerslibrary.infoMail: Reader Feedback Addison-Wesley Developer's Library 800 East 96th Street Indianapolis, IN 46240 USAReader ServicesVisit our website and register this book at for convenient access to any updates, downloads, or errata that might be available for this book.IntroductionIt's no secret that the family of Unix and Unix-like operating systems has emerged over the last few decades as the most pervasive, most widely used group of operating systems in computing today. For programmers who have been using Unix for many years, this came as no surprise: The Unix

system provides an elegant and efficient environment for program development. That's exactly what Dennis Ritchie and Ken Thompson sought to create when they developed Unix at Bell Laboratories way back in the late 1960s.NoteThroughout this book we'll use the term Unix to refer generically to the broad family of Unix-based operating systems, including true Unix operating systems such as Solaris as well as Unix-like operating systems such as Linux and Mac OS X.One of the strongest features of the Unix system is its wide collection of programs. More than 200 basic commands are distributed with the standard operating system and Linux adds to it, often shipping with 700–1000 standard commands! These commands (also known as tools) do everything from counting the number of lines in a file, to sending electronic mail, to displaying a calendar for any desired year.But the real strength of the Unix system comes not from its large collection of commands but from the elegance and ease with which these commands can be combined to perform far more sophisticated tasks.The standard user interface to Unix is the command line, which actually turns out to be a shell, a program that acts as a buffer between the user and the lowest levels of the system itself (the kernel). The shell is simply a program that reads in the commands you type and converts them into a form more readily understood by the system. It also includes core programming constructs that let you make decisions, loop, and store values in variables.The standard shell distributed with Unix systems derives from AT&T's distribution, which evolved from a version originally written by Stephen Bourne at Bell Labs. Since then, the IEEE has created standards based on the Bourne shell and the other more recent shells. The current version of this standard, as of this writing, is the Shell and Utilities volume of IEEE Std 1003.1-2001, also known as the POSIX standard. This shell is what we use as the basis for the rest of this book.The examples in this book were tested on a Mac running Mac OS X 10.11, Ubuntu Linux 14.0, and an old version of SunOS 5.7 running on a Sparcstation Ultra-30. All examples, with the exception of some Bash examples in Chapter 14, were run using the Korn shell, although all of them also work fine with Bash.Because the shell offers an interpreted programming language, programs can be written, modified, and debugged quickly and easily. We turn to the shell as our first choice of programming language and after you become adept at shell programming, you will too.How This Book Is OrganizedThis book assumes that you are familiar with the fundamentals of the system and command line; that is, that you know how to log in; how to create files, edit them, and remove them; and how to work with directories. In case you haven't used the Linux or Unix system for a while, we'll examine the basics in Chapter 1, "A Quick Review of the Basics." In addition, filename substitution, I/O redirection, and pipes are also reviewed in the first chapter.Chapter 2, "What Is the Shell?," reveals what the shell really is, how it works, and how it ends up being your primary method of interacting with the operating system itself. You'll learn about what happens every time you log in to the system, how the shell program gets started, how it parses the command line, and how it executes other programs for you. A key point made in Chapter 2 is that the shell is just another program; nothing more, nothing less.Chapter 3, "Tools of the Trade," provides tutorials on tools useful in writing shell programs. Covered in this chapter are cut, paste, sed, grep, sort, tr, and

uniq. Admittedly, the selection is subjective, but it does set the stage for programs that we'll develop throughout the remainder of the book. Also in Chapter 3 is a detailed discussion of regular expressions, which are used by many Unix commands, such as sed, grep, and ed.Chapters 4 through 9 teach you how to put the shell to work for writing programs. You'll learn how to write your own commands; use variables; write programs that accept arguments; make decisions; use the shell's for, while, and until looping commands; and use the read command to read data from the terminal or from a file. Chapter 5, "Can I Quote you on That?", is devoted entirely to a discussion of one of the most intriguing (and often confusing) aspects of the shell: the way it interprets quotes.By that point in the book, all the basic programming constructs in the shell will have been covered, and you will be able to write shell programs to solve your particular problems.Chapter 10, "Your Environment," covers a topic of great importance for a real understanding of the way the shell operates: the environment. You'll learn about local and exported variables; subshells; special shell variables, such as HOME, PATH, and CDPATH; and how to set up your .profile file.Chapter 11, "More on Parameters," and Chapter 12, "Loose Ends," tie up some loose ends, and Chapter 13, "Rolo Revisited," presents a final version of a phone directory program called rolo that is developed throughout the book.Chapter 14, "Interactive and Nonstandard Shell Features," discusses features of the shell that either are not formally part of the IEEE POSIX standard shell (but are available in most Unix and Linux shells) or are mainly used interactively instead of in programs.Appendix A, "Shell Summary," summarizes the features of the IEEE POSIX standard shell.Appendix B, "For More Information," lists references and resources, including the Web sites where different shells can be downloaded.The philosophy this book uses is to teach by example. We believe that properly chosen examples do a far better job of illustrating how a particular feature is used than ten times as many words. The old "A picture is worth ..." adage seems to apply just as well to coding.We encourage you to type in each example and test it on your own system, for only by doing can you become adept at shell programming. Don't be afraid to experiment. Try changing commands in the program examples to see the effect, or add different options or features to make the programs more useful or robust.Accessing the Free Web EditionYour purchase of this book in any format includes access to the corresponding Web Edition, which provides several special features to help you learn: The complete text of the book online Interactive quizzes and exercises to test your understanding of the material Updates and corrections as they become availableThe Web Edition can be viewed on all types of computers and mobile devices with any modern web browser that supports HTML5.To get access to the Web Edition of Shell Programming with Unix, Linux, and OS X all you need to do is register this book:1. Go to .2. Sign in or create a new account.4. Answer the questions as proof of purchase.The Web Edition will appear under the Digital Purchases tab on your Account page. Click the Launch link to access the product.1. A Quick Review of the BasicsThis chapter provides a review of the Unix system, including the file system, basic commands, filename substitution, I/O redirection, and pipes.Some Basic CommandsDisplaying the Date and Time: The date CommandThe date command tells the system to print the date and

time:$ dateThu Dec 3 11:04:09 MST 2015$date prints the day of the week, month, day, time (24-hour clock, the system's time zone), and year. Throughout the code examples in this book, whenever we use boldface type like this, it's to indicate what you, the user, type in. Normal face type like this is used to indicate what the Unix system prints. Italic type is used for comments in interactive sequences.Every Unix command is submitted to the system with the pressing of the Enter key. Enter says that you are finished typing things in and are ready for the Unix system to do its thing.Finding Out Who's Logged In: The who CommandThe who command can be used to get information about all users currently logged in to the system:Click here to view code image $ whopat tty29 Jul 19 14:40ruth tty37 Jul 19 10:54steve tty25 Jul 19 15:52$Here, three users are logged in: pat, ruth, and steve. Along with each user ID is listed the tty number of that user and the day and time that user logged in. The tty number is a unique identification number the Unix system gives to each terminal or network device that a user is on when they log into the system.The who command also can be used to get information about yourself:Click here to view code image$ who am ipat tty29 Jul 19 14:40$who and who am i are actually the same command: who. In the latter case, the am and i are arguments to the who command. (This isn't a good example of how command arguments work; it's just a curiosity of the who command.)Echoing Characters: The echo CommandThe echo command prints (or echoes) at the terminal whatever else you happen to type on the line (there are some exceptions to this that you'll learn about later):Click here to view code image$ echo this is a testthis is a test$ echo why not print out a longer line with echo?why not print out a longer line with echo?$ echo A blank line is displayed$ echo one two three four fiveone two three four five $You will notice from the preceding example that echo squeezes out extra blanks between words. That's because on a Unix system, the words are important while the blanks are only there to separate the words. Generally, the Unix system ignores extra blanks (you'll learn more about this in the next chapter).Working with FilesThe Unix system recognizes only three basic types of files: ordinary files, directory files, and special files. An ordinary file is just that: any file on the system that contains data, text, program instructions, or just about anything else. Directories, or folders, are described later in this chapter. Finally, as its name implies, a special file has a special meaning to the Unix system and is typically associated with some form of I/O.A filename can be composed of just about any character directly available from the keyboard (and even some that aren't) provided that the total number of characters contained in the name is not greater than 255. If more than 255 characters are specified, the Unix system simply ignores the extra characters.The Unix system provides many tools that make working with files easy. Here we'll review some of the basic file manipulation commands.Listing Files: The ls CommandTo see what files you have stored in your directory, you can type the ls command:$ lsREAD_MEnamestmp$This output indicates that three files called READ_ME, names, and tmp are contained in the current directory. (Note that the output of ls may vary from system to system. For example, on many Unix systems ls produces multicolumn output when sending its output to a terminal; on others, different colors may be used for different types of files. You can always

force single-column output with the –1 option—that's the number one.)Displaying the Contents of a File: The cat CommandYou can examine the contents of a file by using the cat command. (That's short for "concatenate," if you're thinking feline thoughts.) The argument to cat is the name of the file whose contents you want to examine.$ cat namesSusanJeffHenryAllanKen $Counting the Number of Words in a File: The wc CommandWith the wc command, you can get a count of the total number of lines, words, and characters contained in a file. Once again, the name of the file is expected to be specified as the argument to this command:Click here to view code image$ wc names 5 7 27 names$The wc command lists three numbers followed by the filename. The first number represents the number of lines in the file (5), the second the number of words (7), and the third the number of characters (27).Command OptionsMost Unix commands allow the specification of options at the time a command is executed. These options generally follow the same format:-letterThat is, a command option is a minus sign followed immediately by a single letter. For example, to count just the number of lines contained in a file, the option -l (that's the letter l) is given to the wc command:$ wc -l names 5 names$To count just the number of characters in a file, the -c option is specified:$ wc -c names 27 names$Finally, the -w option can be used to count the number of words contained in the file:$ wc -w names 7 names$Some commands require that the options be listed before the filename arguments. For example, sort names -r is acceptable, whereas wc names -l is not. Still, the former is unusual and most Unix commands are designed for you to specify command options first, as exemplified by wc -l names.Making a Copy of a File: The cp CommandTo make a copy of a file, use the cp command. The first argument to the command is the name of the file to be copied (known as the source file), and the second argument is the name of the file to place the copy into (known as the destination file). You can make a copy of the file names and call it saved_names as follows:$ cp names saved_names$Execution of this command causes the contents of the file names to be copied into a new file named saved_names. As with many Unix commands, the fact that no output other than a command prompt was displayed after the cp command was typed indicates that the command executed successfully.Renaming a File: The mv CommandA file can be renamed with the mv ("move") command. The arguments to the mv command follow the same format as the cp command. The first argument is the name of the file to be renamed, and the second argument is the new name. So, to change the name of the file saved_names to hold_it, for example, the following command would do the trick:$ mv saved_names hold_it$Be careful! When executing an mv or cp command, the Unix system does not care whether the file specified as the second argument already exists. If it does, the contents of the file will be lost. For example, if a file called old_names exists, executing the commandcp names old_nameswould copy the filenames to old_names, destroying the previous contents of old_names in the process. Similarly, the commandmv names old_nameswould rename names to old_names, even if the file old_names existed prior to execution of the command.Removing a File: The rm CommandUse the rm command to remove a file from the system. The argument to rm is simply the name of the file to

be removed:$ rm hold_it$You can remove more than one file at a time with the rm command by simply specifying all such files on the command line. For example, the following would remove the three files wb, collect, and mon:$ rm wb collect mon$Working with DirectoriesSuppose that you had a set of files consisting of various memos, proposals, and letters. Further suppose that you had another set of files that were computer programs. It would seem logical to group this first set into a directory called documents and the latter into a directory called programs. Figure 1.1 illustrates such a directory organization.Figure 1.1 Example directory structureThe file directory documents contains the files plan, dact, sys.A, new.hire, no.JSK, and AMG.reply. The directory programs contains the files wb, collect, and mon. At some point, you may decide to further categorize the files in a directory. This can be done by creating subdirectories and then placing each file into the appropriate subdirectory. For example, you might want to create subdirectories called memos, proposals, and letters inside your documents directory, as shown in Figure 1.2.Figure 1.2 Directories containing subdirectoriesdocuments contains the subdirectories memos, proposals, and letters. Each of these subdirectories in turn contains two files: memos contains plan and dact; proposals contains sys.A and new.hire; and letters contains no.JSK and AMG.reply.Although each file in a given directory must have a unique name, files contained in different directories do not. So you could have a file in your programs directory called dact, even though a file by that name also exists in the memos subdirectory.The Home Directory and PathnamesThe Unix system always associates each user of the system with a particular directory. When you log in to the system, you are placed automatically into your own directory (called your home directory).Although the location of users' home directories can vary from one system to the next, let's assume that your home directory is called steve and that this directory is actually a subdirectory of a directory called users. Therefore, if you had the directories documents and programs, the overall directory structure would actually look something like Figure 1.3. A special directory named / (pronounced "slash") is shown at the top of the directory tree. This directory is known as the root.Figure 1.3 Hierarchical directory structureWhenever you are "inside" a particular directory (called your current working directory), the files contained within that directory are immediately accessible, without specifying any path information. If you want to access a file from another directory, you can either first issue a command to "change" to the appropriate directory and then access the particular file, or you can specify the particular file by its pathname.A pathname enables you to uniquely identify a particular file to the Unix system. In the specification of a pathname, successive directories along the path are separated by the slash character /. A pathname that begins with a slash character is known as a full or absolute pathname because it specifies a complete path from the root. For example, the pathname /users/steve identifies the directory steve contained within the directory users. Similarly, the pathname /users/steve/documents references the directory documents as contained in the directory steve within users. As a final example, the pathname /users/steve/documents/letters/AMG.reply identifies the file AMG.reply contained along the appropriate directory path.To help reduce the typing that would otherwise be required, Unix provides certain notational conveniences. A

pathname that does not begin with a slash is known as a relative pathname: the path is relative to your current working directory. For example, if you just logged in to the system and were placed into your home directory /users/steve, you could directly reference the directory documents simply by typing documents. Similarly, the relative pathname programs/mon could be typed to access the file mon contained inside your programs directory.By convention, .. always references the directory that is one level higher than the current directory, known as the parent directory. For example, if you were in your home directory /users/steve, the pathname .. would reference the directory users. If you had issued the appropriate command to change your working directory to documents/letters, the pathname .. would reference the documents directory, ../.. would reference the directory steve, and ../proposals/new.hire would reference the file new.hire contained in the proposals directory. There is usually more than one way to specify a path to a particular file, a very Unix-y characteristic.Another notational convention is the single period., which always refers to the current directory. That'll become more important later in the book when you want to specify a shell script in the current directory, not one in the PATH. We'll explain this in more detail soon.Displaying Your Working Directory: The pwd CommandThe pwd command is used to help you "get your bearings" by telling you the name of your current working directory.Recall the directory structure from Figure 1.3. The directory that you are placed in after you log in to the system is called your home directory. You can assume from Figure 1.3 that the home directory for the user steve is /users/steve. Therefore, whenever steve logs in to the system, he will automatically be placed inside this directory. To verify that this is the case, the pwd (print working directory) command can be issued:$ pwd/users/steve$The output from the command verifies that steve's current working directory is /users/steve.Changing Directories: The cd CommandYou can change your current working directory by using the cd command. This command takes as its argument the name of the target or destination directory.Let's assume that you just logged in to the system and were placed in your home directory, /users/steve. This is depicted by the arrow in Figure 1.4.Figure 1.4 Current working directory is steveYou know that two directories are directly "below" steve's home directory: documents and programs. This can be easily verified at the terminal by issuing the ls command:$ lsdocumentsprograms$The ls command lists the two directories documents and programs the same way it listed other ordinary files in previous examples.To change your current working directory, issue the cd command, followed by the name of the new directory:$ cd documents$After executing this command, you will be placed inside the documents directory, as depicted in Figure 1.5.Figure 1.5 cd documentsYou can verify at the terminal that the working directory has been changed by using the pwd command:$ pwd/users/steve/documents$The easiest way to move up one level in a directory is to reference the .. shortcut with the commandcd ..because by convention .. always refers to the directory one level up (see Figure 1.6).$ cd ..$ pwd/users/steve$Figure 1.6 cd ..If you wanted to change to the letters directory, you could get there with a single cd command by specifying the relative path documents/letters (see Figure 1.7):Click here to view code image$ cd documents/letters$ pwd/users/steve/documents/letters$Figure 1.7 cd documents/lettersYou

can get back up to your home directory by using a single cd command to go up two directories as shown:$ cd ../..$ pwd/users/steve$Or you can get back to the home directory using a full pathname rather than a relative one:$ cd /users/steve$ pwd/users/steve$Finally, there is a third way to get back to the home directory that is also the easiest. Typing the command cd without an argument always moves you back to your home directory, no matter where you are in the file system:$ cd$ pwd/users/steve$More on the ls CommandWhen you type the ls command, the files contained in the current working directory are listed. But you can also use ls to obtain a list of files in other directories by supplying an argument to the command. First let's get back to your home directory:$ cd$ pwd/users/steve$Now let's take a look at the files in the current working directory:$ lsdocumentsprograms$If you supply the name of one of these directories to the ls command, you can get a list of the contents of that directory. So you can find out what's contained in the documents directory by typing the command ls documents:$ ls documentslettersmemosproposals$To take a look at the subdirectory memos, you can follow a similar procedure:$ ls documents/memosdactplan$If you specify a nondirectory file argument to the ls command, you simply get that filename echoed back at the terminal:$ ls documents/memos/plandocuments/memos/plan$Confused? There's an option to the ls command that lets you determine whether a particular file is a directory, among other things. The -l option (the letter l) provides a more detailed description of the files in a directory. If you were currently in steve's home directory, here's what the -l option to the ls command produces:Click here to view code image$ ls –ltotal 2drwxr-xr-x 5 steve DP3725 80 Jun 25 13:27 documentsdrwxr-xr-x 2 steve DP3725 96 Jun 25 13:31 programs$The first line of the display is a count of the total number of blocks (1,024 bytes) of storage that the listed files use. Each successive line displayed by the ls -l command contains detailed information about a file in the directory. The first character on each line indicates what type of file it is: d for a directory, - for a file, b, c, l, or p for a special file.The next nine characters on the line define the access permissions of that particular file or directory. These access modes apply to the file's owner (the first three characters), other users in the same group as the file's owner (the next three characters), and finally all other users on the system (the last three characters). Generally, they indicate whether the specified class of user can read the file, write to the file, or execute the contents of the file (in the case of a program or shell script).The ls -l command then shows the link count (see "Linking Files: The ln Command," later in this chapter), the owner of the file, the group owner of the file, how large the file is (that is, how many characters are contained in it), and when the file was last modified. The information displayed last on the line is the filename itself.NoteMany modern Unix systems have gone away from using groups, so while those permissions are still shown, the group owner for a specific file or directory is often omitted in the output of the ls command.You should now be able to glean a lot of information from the ls -l output for a directory full of files:Click here to view code image$ ls -l programstotal 4-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 1 steve DP3725 89 Jun 25 13:30 wb$The dash in the first column of each line indicates that the

three files collect, mon, and wb are ordinary files and not directories. Now, can you figure out how big are they?Creating a Directory: The mkdir CommandUse mkdir to create directories. The argument to this command is simply the name of the directory you want to create. For example, assume that you are still working with the directory structure depicted in Figure 1.7 and that you want to create a new directory called misc at the same level as the directories documents and programs. If you were currently in your home directory, typing the command mkdir misc would achieve the desired effect:$ mkdir misc$Now if you run ls, you will have the new directory listed: $ lsdocumentsmiscprograms$The directory structure now appears as shown in Figure 1.8.Figure 1.8 Directory structure with newly created misc directoryCopying a File from One Directory to AnotherThe cp command can be used to copy a file from one directory into another. For example, you can copy the file wb from the programs directory into a file called wbx in the misc directory as follows:$ cp programs/wb misc/wbx$Because the two files are in different directories, they can safely have the exact same name:$ cp programs/wb misc/wb$When the destination file is going to have the same name as the source file (in a different directory, of course), it is necessary to specify only the destination directory as the second argument:$ cp programs/wb misc$When this command gets executed, the Unix system recognizes that the second argument is a directory and copies the source file into that directory. The new file is given the same name as the source file.You can copy more than one file into a directory by listing the files to be copied prior to the name of the destination directory. If you were currently in the programs directory, the command$ cp wb collect mon ../misc$would copy the three files wb, collect, and mon into the misc directory, with the same filenames.To copy a file from another directory into your current location in the file system and give the file the same name, use the handy "." shortcut for the current directory:$ pwd/users/steve/misc$ cp ../programs/collect .$The preceding command copies the file collect from the directory ../programs into the current directory (/users/steve/misc).Moving Files Between DirectoriesYou recall that the mv command can be used to rename a file. Indeed, there is no "rename" command in Unix. However, when the two arguments reference different directories, the file is actually moved from the first directory into the second.To demonstrate, go from the home directory to the documents directory:$ cd documents$Suppose that now you decide that the file plan contained in the memos directory is really a proposal so you want to move it from the memos directory into the proposals directory. The following would do the trick:Click here to view code image$ mv memos/plan proposals/plan $As with the cp command, if the source file and destination file have the same name, only the name of the destination directory need be supplied, so there's an easier way to move this file:$ mv memos/plan proposals$Also like the cp command, a group of files can be simultaneously moved into a directory by simply listing all files to be moved before the name of the destination directory:$ pwd/users/steve/programs$ mv wb collect mon ../misc$This would move the three files wb, collect, and mon into the directory misc.You can also use the mv command to change the name of a directory, as it happens. For example, the following renames the programs directory to bin.$ mv programs bin$Linking Files: The ln CommandSo far everything we've

talked about with file management has assumed that a given collection of data has one and only one filename, wherever it may be located in the file system. It turns out that Unix is more sophisticated than that and can assign multiple filenames to the same collection of data.The main command for creating these duplicate names for a given file is the ln command.The general form of the command isln from toThis links the file from to the file to.Recall the structure of steve's programs directory from Figure 1.8. In that directory, he has stored a program called wb. Suppose that he decides that he'd also like to call the program writeback. The most obvious thing to do would be to simply create a copy of wb called writeback:$ cp wb writeback$The drawback with this approach is that now twice as much disk space is being consumed by the program. Furthermore, if steve ever changes wb, he may forget to duplicate the change in writeback, resulting in two different, out of sync copies of what he thinks is the same program. Not so good, Steve!By linking the file wb to the new name, these problems are avoided:$ ln wb writeback$Now instead of two copies of the file existing, only one exists with two different names: wb and writeback. The two files have been logically linked by the Unix system.As far as you're concerned, it appears as though you have two different files. Executing an ls command shows the two files separately:$ lscollectmonwbwriteback$Where it gets interesting is when you use ls -l:Click here to view code image$ ls -ltotal 5-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 writeback $Look closely at the second column of the output: The number shown is 1 for collect and mon and 2 for wb and writeback. This is the number of links to a file, normally 1 for nonlinked, nondirectory files. Because wb and writeback are linked, however, this number is 2 for these files (or, more correctly, this file with two names).You can remove either of the two linked files at any time, and the other will not be removed:Click here to view code image$ rm writeback$ ls -ltotal 4-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 1 steve DP3725 89 Jun 25 13:30 wb$Note that the number of links on wb went from 2 to 1 because one of its links was removed.Most often, ln is used to allow a file to appear in more than one directory simultaneously. For example, suppose that pat wanted to have access to steve's wb program. Instead of making a copy for himself (subject to the same data sync problems described previously) or including steve's programs directory in his PATH (which has security risks as described in Chapter 10, "Your Environment"), he can simply link to the file from his own program directory:Click here to view code image$ pwd/users/pat/bin pat's program directory$ ls -ltotal 4-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ ln /users/steve/wb . link wb to pat's bin$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr $Note that steve is still listed as the owner of wb, even when viewing the contents of pat's directory. This makes sense, because there's really only one copy of the file and it's owned by steve.The only stipulation on linking files is that for ordinary links the files to be linked together

must reside on the same file system. If they don't, you'll get an error from ln when you try to link them. (To determine the different file systems on your system, execute the df command. The first field on each line of output is the name of a file system.)To create links to files on different file systems (or on different networked systems), you can use the -s option to the ln command. This creates a symbolic link. Symbolic links behave a lot like regular links, except that the symbolic link points to the original file; if the original file is removed, the symbolic link no longer works.Let's see how symbolic links work with the previous example:Click here to view code image$ rm wb$ ls -ltotal 4-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ ln -s /users/steve/wb ./symwb Symbolic link to wb$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcatlrwxr-xr-x 1 pat DP3822 15 Jul 20 15:22 symwb -> /users/steve/wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$Note that pat is listed as the owner of symwb, and the file type shown as the very first character in the ls output is l, which indicates a symbolic link. The size of the symbolic link is 15 (the file actually contains the string /users/steve/wb), but if we attempt to access the contents of the file, we are presented with the contents of the file it's linked to, /users/steve/wb:Click here to view code image$ wc symwb 5 9 89 symwb$The -L option to the ls command can be used with the -l option to get a detailed list of information on the file the symbolic link points to:Click here to view code image$ ls -Ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr $Removing the file that a symbolic link points to invalidates the symbolic link (because symbolic links are maintained as filenames), but it doesn't remove it:Click here to view code image$ rm /users/steve/wb Assume pat can remove this file$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcatlrwxr-xr-x 1 pat DP3822 15 Jul 20 15:22 wb -> /users/steve/wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ wc wbCannot open wb: No such file or directory$This type of file is called a dangling symbolic link and should be removed unless you have a specific reason to keep it around (for example, if you intend to replace the removed file).One last note before leaving this discussion: The ln command follows the same general format as cp and mv, meaning that you can create links to a bunch of files within a specific target directory using the formatln files directoryRemoving a Directory: The rmdir CommandYou can remove a directory with the rmdir command. Rather than let you accidentally remove dozens or hundreds of files, however, rmdir won't let you proceed unless the specified directory is completely empty of files and subdirectories.To remove the directory /users/pat, we could use the following:Click here to view code image$ rmdir /users/patrmdir: pat: Directory not empty $Phew! That would have been a mistake! Instead, let's remove the misc directory that you created earlier:$ rmdir /users/steve/misc$Once again, the preceding command works only if no files or directories are contained in the misc directory; otherwise, the following happens, as also shown earlier:Click here to view code image$ rmdir /users/steve/miscrmdir: /users/steve/misc: Directory not empty$If you still want to remove the misc directory, you would first have to remove all the files contained in that directory before reissuing the rmdir command.As an alternative

method for removing a directory and its contents, you can use the -r option to the rm command. The format is simple:rm -r dirwhere dir is the name of the directory that you want to remove. rm removes the indicated directory and all files (including directories) in it, so be careful with this powerhouse command.Want to go full turbo? Add the -f flag and it forces the action without prompting you on a command-by-command basis. It can completely trash your system if you're not careful, however, so many admins simply avoid rm -rf entirely!Filename SubstitutionThe AsteriskOne powerful feature of the Unix system that is handled by the shell is filename substitution. Let's say that your current directory has these files in it:$ lschaptlchapt2chapt3chapt4$Suppose that you want to display their contents en masse. Easy: cat allows you to display the contents of as many files as you specify on the command line. Like this:Click here to view code image$ cat chaptl chapt2 chapt3 chapt4 ...$But that's tedious. Instead, you can take advantage of filename substitution by simply typing:$ cat * ...$The shell automatically substitutes the names of all the files in the current directory that match the pattern *. The same substitution occurs if you use * with another command too, of course. How about echo?$ echo *chaptl chapt2 chapt3 chapt4$Here the * is again replaced with the names of all the files contained in the current directory, and the echo command simply displays that list to you.Any place that * appears on the command line, the shell performs its substitution:Click here to view code image$ echo * : *chaptl chapt2 chapt3 chapt4 : chaptl chapt2 chapt3 chapt4$The * is part of a rich file substitution language, actually, and it can also be used in combination with other characters to limit which filenames are matched.For example, let's say that in your current directory you have not only chapt1 through chapt4 but also files a, b, and c:$ lsabcchaptlchapt2chapt3chapt4$To display the contents of just the files beginning with chap, you can type in$ cat chap* . . .$The chap* matches any filename that begins with chap. All such filenames matched are substituted on the command line before the specified command is even invoked.The * is not limited to the end of a filename; it can be used at the beginning or in the middle as well:$ echo *t1chaptl$ echo *t*chaptl chapt2 chapt3 chapt4$ echo *x*x$In the first echo, the *t1 specifies all filenames that end in the characters t1. In the second echo, the first * matches everything up to a t and the second everything after; thus, all filenames containing a t are printed. Because there are no files ending with x, no substitution occurs in the last case. Therefore, the echo command simply displays *x.Matching Single CharactersThe asterisk (*) matches zero or more characters, meaning that x* matches the file x as well as x1, x2, xabc, and so on. The question mark (?) matches exactly one character. So cat? will display all files that have filenames of exactly one character, just as cat x? prints all files with two-character names beginning with x. Here we see this behavior illustrated again with echo:Click here to view code image$ lsaaaaaxalicebbbcccreport1report2report3$ echo ?a b c$ echo a?aa$ echo ??aa bb cc $ echo ??*aa aax alice bb cc report1 report2 report3$In the preceding example, the ?? matches two characters, and the * matches zero or more characters up to the end. The net effect is to match all filenames of two or more characters.Another way to match a single character is to give a list of characters to match within square brackets [ ]. For example, [abc] matches the letter a, b,

or c. It's similar to the ?, but it allows you to choose which characters are valid matches.You can also specify a logical range of characters with a dash, a huge convenience! For example, [0–9] matches the characters 0 through 9. The only restriction in specifying a range of characters is that the first character must be alphabetically less than the last character, so that [z–f] is not a valid range specification, while [f–z] is.By mixing and matching ranges and characters in the list, you can perform complicated substitutions. For example, [a–np–z]* matches all files that start with the letters a through n or p through z (or more simply stated, any filename that doesn't start with the lowercase letter o).If the first character following the [is a !, the sense of the match is inverted. That is, any character is matched except those enclosed in the brackets. So[!a–z]matches any character except a lowercase letter, and*[!o]matches any file that doesn't end with the lowercase letter o.Table 1.1 gives a few more examples of filename substitution.Table 1.1 Filename Substitution ExamplesFilename NuancesSpaces in FilenamesA discussion of command lines and filenames wouldn't be complete without talking about the bane of old-school Unix people and very much the day-to-day reality of Linux, Windows, and Mac users: spaces in filenames.The problem arises from the fact that the shell uses spaces as delimiters between words. In other words the phrase echo hi mom is properly parsed as an invocation to the command echo, with two arguments hi and mom.Now imagine you have a file called my test document. How do you reference it from the command line? How do you view it or display it using the cat command?Click here to view code image$ cat my test documentcat: my: No such file or directorycat: test: No such file or directorycat: document: No such file or directoryThat definitely doesn't work. Why? Because cat wants a filename to be specified and instead of seeing one, it sees three: my, test, and document.There are two standard solutions for this: Either escape every space by using a backslash, or wrap the entire filename in quotes so that the shell understands that it's a single word with spaces, rather than multiple words.Click here to view code image$ cat "my test document"This is a test document and is fullof scintillating information to edifyand amaze.$ cat my\ test\ documentThis is a test document and is fullof scintillating information to edifyand amaze.That solves the problem and is critical to know as you proceed with file systems that quite likely have lots of directories and files that contain spaces as part of their filenames.Other Weird CharactersWhile the space might be the most difficult and annoying of special characters that can appear in filenames, occasionally you'll find others show up that can throw a proverbial monkey-wrench into your command line efforts.For example, how would you deal with a filename that contains a question mark? In the next section, you'll learn that the character "?" has a specific meaning to the shell. Most modern shells are smart enough to sidestep the duplication of meaning, but, again, quoting the filename or using backslashes to denote that the special character is part of the filename is required:Click here to view code image $ ls -l who\ me\?-rw-r--r-- 1 taylor staff 0 Dec 4 10:18 who me?Where this really gets interesting is if you have a backslash or quote as part of the filename, something that can happen inadvertently, particularly for files created by graphically oriented programs on a Linux or Mac system. The trick? Use single quotes to escape a filename that includes a double quote,

and vice versa. Like this:Click here to view code image$ ls -l "don't quote me" 'She said "yes"'-rw-r--r-- 1 taylor staff 0 Dec 4 10:18 don't quote me-rw-r--r-- 1 taylor staff 0 Dec 4 10:19 She said "yes"This topic will come up again as we proceed, but now you know how to side-step problems with directories or files that contain spaces or other non-standard characters.Standard Input/Output, and I/O RedirectionStandard Input and Standard OutputMost Unix system commands take input from your screen and send the resulting output back to your screen. In Unix nomenclature, the screen is generally called the terminal, a reference that harkens back to the earliest days of computing. Nowadays it's more likely to be a terminal program you're running within a graphical environment, whether it's a Linux window manager, a Windows computer, or a Mac system.A command normally reads its input from standard input, which is your computer keyboard by default. It's a fancy way of clarifying that you "type in" your information. Similarly, a command normally writes its output to standard output, which is also your terminal or terminal app by default. This concept is depicted in Figure 1.9.Figure 1.9 Typical Unix commandAs an example, recall that executing the who command results in the display of all users that are currently logged-in. More formally, the who command writes a list of the logged-in users to standard output. This is depicted in Figure 1.10.Figure 1.10 who commandIt turns out that just about every single Unix command can take the output of a previous command or file as its input too, and can even send its output to another command or program. This concept is hugely important to understanding the power of the command line and why it's so helpful to know all of these commands even when a graphical interface might be also available for your use.Before we get there, however, consider this: if the sort command is invoked without a filename argument, the command takes its input from standard input. As with standard output, this is your terminal (or keyboard) by default.When input is entered this way, an end-of-file sequence must be specified after the last line is typed, and, by Unix convention, that's Ctrl+d; that is, the sequence produced by simultaneously pressing the Control (or Ctrl, depending on your keyboard) key and the d key.As an example, let's use the sort command to sort the following four names: Tony, Barbara, Harry, Dirk. Instead of first entering the names into a file, we'll enter them directly from the terminal:$ sortTonyBarbaraHarryDirkCtrl+dBarbaraDirkHarryTony$Because no filename was specified to the sort command, the input was taken from standard input, the terminal. After the fourth name was typed in, the Ctrl and d keys were pressed to signal the end of the data. At that point, the sort command sorted the four names and displayed the results on the standard output device, which is also the terminal. This is depicted in Figure 1.11.Figure 1.11 sort commandThe wc command is another example of a command that takes its input from standard input if no filename is specified on the command line. The following shows an example of this command used to count the number of lines of text entered from the terminal:$ wc -lThis is text thatis typed on thestandard input device.Ctrl+d 3$Note that the Ctrl+d that is used to terminate the input is not counted as a separate line by the wc command because it's interpreted by the shell, not handed to the command. Furthermore, because the -l flag was specified to the wc command, only the count of the number of lines (3) is presented as the

output of the command.Output RedirectionThe output from a command normally intended for standard output can be easily “diverted” to a file instead. This capability is known as output redirection and is also essential to understanding the power of Unix.If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to the file file instead:$ who > users$This command causes the who command to be executed and its output to be written into the file users. Notice that no output appears. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. We can check this, of course:$ cat usersoko tty01 Sep 12 07:30ai tty15 Sep 12 13:32ruth tty21 Sep 12 10:10pat tty24 Sep 12 13:07steve tty25 Sep 12 13:03$If a command has its output redirected to a file and the file already contains some data, that data will be overwritten and lost.$ echo line 1 > users$ cat usersline 1$But now consider this example, remembering that users already contains the output of the earlier who command:$ echo line 2 >> users$ cat usersline 1line 2$If you’re paying close attention you’ll notice that this echo command uses a different type of output redirection, indicated by the characters >>. This character pair causes the standard output from the command to be appended to the contents of the specified file. The previous contents are not lost; the new output simply gets added to the end.By using the redirection append characters >>, you can use cat to append the contents of one file onto the end of another:Click here to view code image$ cat file1This is in file1.$ cat file2This is in file2.$ cat file1 >> file2 Append file1 to file2$ cat file2This is in file2.This is in file1.$Recall that specifying more than one filename to cat results in the display of the first file followed immediately by the second file, and so on. This means there’s a second way to accomplish the same result:Click here to view code image$ cat file1This is in file1.$ cat file2This is in file2.$ cat file1 file2This is in file1.This is in file2.$ cat file1 file2 > file3 Redirect it instead$ cat file3This is in filel.This is in file2.$In fact, that’s where the cat command gets its name: When used with more than one file, its effect is to concatenate the files together.Input RedirectionJust as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. And as the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command. Of course, only commands that normally take their input from standard input can have their input redirected from a file in this manner.To redirect input, type the < character followed by the name of the file that the input is to be read from. To count the number of lines in the file users, for example, you already know that you can execute the command wc -l users:$ wc -l users 2 users$It turns out that you can also count the number of lines in the file by redirecting standard input for the wc command:$ wc -l < users 2$Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.This points out a subtle distinction between the execution of the two commands. In the first case, wc knows that it is reading its input from the file users. In the second case, it only sees the raw data which is being fed to it via standard input. The shell redirects the input so that it comes from the file users and not the terminal (more about

this in the next chapter). As far as wc is concerned, it doesn't know whether its input is coming from the terminal or from a file, so it can't report the filename!PipesAs you will recall, the file users that was created previously contains a list of all the users currently logged in to the system. Because you know that there will be one line in the file for each user logged in to the system, you can easily determine the number of login sessions by counting the number of lines in the users file:$ who > users$ wc -l < users 5$This output indicates that currently five users are logged in or that there are five login sessions, the difference being that users, particularly administrators, often log in more than once. Now you have a command sequence you can use whenever you want to know how many users are logged in.Another approach to determine the number of logged-in users bypasses the intermediate file. As referenced earlier, Unix lets you "connect" two commands together. This connection is known as a pipe, and it enables you to take the output from one command and feed it directly into the input of another. A pipe is denoted by the character |, which is placed between the two commands. To create a pipe between the who and wc -l commands, you type who | wc -l:$ who | wc -l 5$The pipe that is created between these two commands is depicted in Figure 1.12.Figure 1.12 Pipeline process: who | wc -lWhen a pipe is established between two commands, the standard output from the first command is connected directly to the standard input of the second command. You know that the who command writes its list of logged-in users to standard output. Furthermore, you know that if no filename argument is specified to the wc command, it takes its input from standard input. Therefore, the list of logged-in users that is output from the who command automatically becomes the input to the wc command. Note that you never see the output of the who command at the terminal because it is piped directly into the wc command. This is depicted in Figure 1.13.Figure 1.13 Pipeline processA pipe can be made between any two programs, provided that the first program writes its output to standard output, and the second program reads its input from standard input.As another example, suppose you wanted to count the number of files contained in your directory. Knowledge of the fact that the ls command displays one line of output per file enables you to use the same type of approach as before:$ ls | wc -l 10$The output indicates that the current directory contains 10 files.It is also possible to create a more complicated pipeline that consists of more than two programs, with the output of one program feeding into the input of the next. As you become a more sophisticated command line user, you'll find many situations where pipelines can be tremendously powerful.FiltersThe term filter is often used in Unix terminology to refer to any program that can take input from standard input, perform some operation on that input, and write the results to standard output. More succinctly, a filter is any program that can be used to modify the output of other programs in a pipeline. So in the pipeline in the previous example, wc is considered a filter. ls is not because it does not read its input from standard input. As other examples, cat and sort are filters, whereas who, date, cd, pwd, echo, rm, mv, and cp are not.Standard ErrorIn addition to standard input and standard output, there is a third virtual device known as standard error. This is where most Unix commands write their error messages. And as with the other two "standard" places, standard

error is associated with your terminal or terminal app by default. In most cases, you never know the difference between standard output and standard error:Click here to view code image$ ls n* List all files beginning with nn* not found$Here the “not found” message is actually being written to standard error by the ls command. You can verify that this message is not being written to standard output by redirecting the ls command’s output:$ ls n* > foon* not found$As you can see, the message is still printed out at the terminal and was not added to the file foo, even though you redirected standard output.The preceding example shows the raison d’être for standard error: so that error messages will still get displayed at the terminal even if standard output is redirected to a file or piped to another command.You can also redirect standard error to a file (for instance, if you’re logging a program’s potential errors during long-term operation) by using the slightly more complex notationcommand 2> fileNote that no space is permitted between the 2 and the >. Any error messages normally intended for standard error will be diverted into the specified file, similar to the way standard output gets redirected.$ ls n* 2> errors$ cat errorsn* not found$More on CommandsTyping More Than One Command on a LineYou can type more than one command on a line provided that you separate them with a semicolon. For example, you can find out the current time and your current working directory by typing in the date and pwd commands on the same line:$ date; pwdSat Jul 20 14:43:25 EDT 2002/users/pat/bin$You can string out as many commands as you want on the line, as long as each command is delimited by a semicolon.Sending a Command to the BackgroundNormally, you type in a command and then wait for the results of the command to be displayed at the terminal. For all the examples you have seen thus far, this waiting time is typically short—a fraction of a second.Sometimes, however, you may have to run commands that require a few minutes or longer to complete. In those cases, you’ll have to wait for the command to finish executing before you can proceed further, unless you execute the command in the background.It turns out that while your Unix or Linux system seems like it’s focused completely on what you’re doing, all systems are actually multitasking, running multiple commands simultaneously at any given time. If you’re on an Ubuntu system, for example, it might have the window manager, a clock, a status monitor and your terminal window all running simultaneously. You too can run multiple commands simultaneously from the command line. That’s the idea of putting a command “into background,” letting you work on other tasks while it completes.The notational convention for pushing a command or command sequence into background is to append the ampersand character &. This means that the command will no longer tie up your terminal, and you can then proceed with other work. The standard output from the command will still be directed to your terminal, though in most cases the standard input will be dissociated from your terminal. If the command does try to read from standard input, it will stop and wait for you to bring it to the foreground (we’ll discuss this in more detail in Chapter 14, “Interactive and Nonstandard Shell Features”).Here’s an example:Click here to view code image$ sort bigdata > out & Send the sort to the background[1] 1258 Process id$ date Your terminal is immediately available to do other workSat Jul 20 14:45:09 EDT 2002$When a

command is sent to the background, the Unix system automatically displays two numbers. The first is called the command's job number and the second the process ID, or PID. In the preceding example, 1 is the job number and 1258 the process ID. The job number is used as a shortcut for referring to a specific background job by some shell commands. (You'll learn more about this in Chapter 14.) The process ID uniquely identifies the command that you sent to the background and can be used to obtain status information about the command. This is done with the processor status—ps—command.The ps CommandThe ps command gives you information about the processes running on the system. Without any options, it prints the status of just your processes. If you type in ps at your terminal, you'll get a few lines back describing the processes you have running:$ ps PID TTY TIME CMD13463 pts/16 00:00:09 bash19880 pts/16 00:00:00 ps$The ps command (typically; your system might vary) prints out four columns of information: PID, the process ID; TTY, the terminal number that the process was run from; TIME, the amount of computer time in minutes and seconds that process has used; and CMD, the name of the process. (The bash process in the preceding example is the shell that was started when we logged in, and it's used 9 seconds of computer time.) Until the command is finished, it shows up in the output of the ps command as a running process, so process 19880 in the preceding example is the ps command itself.When used with the -f option, ps prints out more information about your processes, including the parent process ID (PPID), the time the process started (STIME), and the command arguments:Click here to view code image$ ps -fUID PID PPID C STIME TTY TIME CMDsteve 13463 13355 0 12:12 pts/16 00:00:09 bashsteve 19884 13463 0 13:39 pts/16 00:00:00 ps -f$Command SummaryTable 1.2 summarizes the commands reviewed in this chapter. In this table, file refers to a file, file(s) to one or more files, dir to a directory, and dir(s) to one or more directories.Table 1.2 Command SummaryAbout This E-BookEPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.About This E-BookEPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font,

font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer’s Web site.Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.Shell Programming in Unix, Linux and OS XFourth EditionStephen G. KochanPatrick Wood800 East 96th Street, Indianapolis, Indiana 46240Shell Programming in Unix, Linux and OS XFourth EditionStephen G. KochanPatrick Wood800 East 96th Street, Indianapolis, Indiana 46240Shell Programming in Unix, Linux and OS X, Fourth EditionCopyright © 2017 by Pearson Education, Inc.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. Nor is any liability assumed for damages resulting from the use of the information contained herein.Printed in the United States of AmericaFirst Printing: August 2016The Library of Congress Control Number is on file.EditorMark TaberCopy EditorLarry SulkyTechnical EditorBrian TiemannDesignerChuti PrasertsithPage LayoutcodeMantraTrademarksAll terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. The publisher 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 DisclaimerEvery 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 author 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.Special SalesFor information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.For government sales inquiries, please contactgovernmentsales@pearsoned.comFor questions about sales outside the U.S., please contactinternational@pearsoned.comShell Programming in Unix, Linux and OS X, Fourth EditionCopyright © 2017 by Pearson Education, Inc.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. Nor is any liability assumed for damages resulting from the use of the information contained herein.ISBN-13: 978-0-13-4449600-9ISBN-10: 0-13-449600-0Printed in the United States of AmericaFirst Printing: August 2016The Library of Congress Control Number is on file.EditorMark TaberCopy EditorLarry SulkyTechnical EditorBrian TiemannDesignerChuti PrasertsithPage LayoutcodeMantraTrademarksAll terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. The publisher 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 DisclaimerEvery 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 author 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.Special SalesFor information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.For government sales inquiries, please contactgovernmentsales@pearsoned.comFor questions about sales outside the U.S., please contactinternational@pearsoned.comDeveloper's LibraryESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALSDeveloper's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.All books in the Developer's Library are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.Key titles include some of the best, most widely acclaimed books within their topic areas:Developer's Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online atDeveloper's LibraryDeveloper's LibraryESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALSDeveloper's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.All books in the Developer's Library are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.Key titles include some of the best, most widely acclaimed books within their topic areas:PHP & MySQL Web DevelopmentLuke Welling & Laura ThomsonISBN-13: 978-0-321-83389-1MySQLPaul DuBoisISBN-13: 978-0-672-32938-8Linux Kernel DevelopmentRobert LoveISBN-13: 978-0-672-32946-3Python Essential ReferenceDavid BeazleyISBN-13: 978-0-672-32862-6Programming in Objective-CStephen G. KochanISBN-13:

978-0-321-56615-7Programming in CStephen G. KochanISBN-13: 978-0-321-77641-9Developer’s Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online at Developer’s LibraryContents at a GlanceIntroduction1 A Quick Review of the Basics2 What Is the Shell?3 Tools of the Trade4 And Away We Go5 Can I Quote You on That?6 Passing Arguments7 Decisions, Decisions8 'Round and 'Round She Goes9 Reading and Printing Data10 Your Environment11 More on Parameters12 Loose Ends13 Rolo Revisited14 Interactive and Nonstandard Shell FeaturesA Shell SummaryB For More InformationIndexContents at a GlanceIntroduction1 A Quick Review of the Basics2 What Is the Shell?3 Tools of the Trade4 And Away We Go5 Can I Quote You on That?6 Passing Arguments7 Decisions, Decisions8 'Round and 'Round She Goes9 Reading and Printing Data10 Your Environment11 More on Parameters12 Loose Ends13 Rolo Revisited14 Interactive and Nonstandard Shell FeaturesA Shell SummaryB For More InformationIndexTable of ContentsIntroductionHow This Book Is OrganizedAccessing the Free Web Edition1 A Quick Review of the BasicsSome Basic CommandsDisplaying the Date and Time: The date CommandFinding Out Who’s Logged In: The who CommandEchoing Characters: The echo CommandWorking with FilesListing Files: The ls CommandDisplaying the Contents of a File: The cat CommandCounting the Number of Words in a File: The wc CommandCommand OptionsMaking a Copy of a File: The cp CommandRenaming a File: The mv CommandRemoving a File: The rm CommandWorking with DirectoriesThe Home Directory and PathnamesDisplaying Your Working Directory: The pwd CommandChanging Directories: The cd CommandMore on the ls CommandCreating a Directory: The mkdir CommandCopying a File from One Directory to AnotherMoving Files Between DirectoriesLinking Files: The ln CommandRemoving a Directory: The rmdir CommandFilename SubstitutionThe AsteriskMatching Single CharactersFilename NuancesSpaces in FilenamesOther Weird CharactersStandard Input/Output, and I/O RedirectionStandard Input and Standard OutputOutput RedirectionInput RedirectionPipesFiltersStandard ErrorMore on CommandsTyping More Than One Command on a LineSending a Command to the BackgroundThe ps CommandCommand Summary2 What Is the Shell?The Kernel and the UtilitiesThe Login ShellTyping Commands to the ShellThe Shell’s ResponsibilitiesProgram ExecutionVariable and Filename SubstitutionI/O RedirectionHooking up a PipelineEnvironment ControlInterpreted Programming Language3 Tools of the TradeRegular ExpressionsMatching Any Character: The Period (.)Matching the Beginning of the Line: The Caret (^)Matching the End of the Line: The Dollar Sign $Matching a Character Set: The [...] ConstructMatching Zero or More Characters: The Asterisk (*)Matching a Precise Number of Subpatterns: \{...\}Saving Matched Characters: \(...\)cutThe -d and -f OptionspasteThe -d OptionThe -s OptionsedThe -n OptionDeleting LinestrThe -s OptionThe -d OptiongrepRegular Expressions and grepThe -v OptionThe -l OptionThe -n OptionsortThe -u OptionThe -r OptionThe -o OptionThe -n OptionSkipping FieldsThe -t OptionOther OptionsuniqThe -d OptionOther Options4 And Away We GoCommand FilesCommentsVariablesDisplaying the Values of VariablesUndefined

Variables Have the Null ValueFilename Substitution and VariablesThe ${variable} ConstructBuilt-in Integer Arithmetic5 Can I Quote You on That?The Single QuoteThe Double QuoteThe BackslashUsing the Backslash for Continuing LinesThe Backslash Inside Double QuotesCommand SubstitutionThe Back QuoteThe $(...) ConstructThe expr Command6 Passing ArgumentsThe $# VariableThe $* VariableA Program to Look Up Someone in the Phone BookA Program to Add Someone to the Phone BookA Program to Remove Someone from the Phone Book${n}The shift Command7 Decisions, DecisionsExit StatusThe $? VariableThe test CommandString OperatorsAn Alternative Format for testInteger OperatorsFile OperatorsThe Logical Negation Operator !The Logical AND Operator -aParenthesesThe Logical OR Operator -oThe else ConstructThe exit CommandA Second Look at the rem ProgramThe elif ConstructYet Another Version of remThe case CommandSpecial Pattern-Matching CharactersThe -x Option for Debugging ProgramsBack to the caseThe Null Command :The && and || Constructs8 'Round and 'Round She GoesThe for CommandThe $@ VariableThe for Without the ListThe while CommandThe until CommandMore on LoopsBreaking Out of a LoopSkipping the Remaining Commands in a LoopExecuting a Loop in the BackgroundI/O Redirection on a LoopPiping Data into and out of a LoopTyping a Loop on One LineThe getopts Command9 Reading and Printing DataThe read CommandA Program to Copy FilesSpecial echo Escape CharactersAn Improved Version of mycpA Final Version of mycpA Menu-Driven Phone ProgramThe $$ Variable and Temporary FilesThe Exit Status from readThe printf Command10 Your EnvironmentLocal VariablesSubshellsExported Variablesexport -pPS1 and PS2HOMEPATHYour Current DirectoryCDPATHMore on SubshellsThe .CommandThe exec CommandThe (...) and { ...; } ConstructsAnother Way to Pass Variables to a SubshellYour .profile FileThe TERM VariableThe TZ Variable11 More on ParametersParameter Substitution${parameter}${parameter:-value}${parameter:=value}${parameter:?value}${parameter:+value}Pattern Matching Constructs ${#variable}The $0 VariableThe set CommandThe -x Optionset with No ArgumentsUsing set to Reassign Positional ParametersThe -- OptionOther Options to setThe IFS VariableThe readonly CommandThe unset Command12 Loose EndsThe eval CommandThe wait CommandThe $! VariableThe trap Commandtrap with No ArgumentsIgnoring SignalsResetting TrapsMore on I/O<&- and >&-In-line Input RedirectionShell ArchivesFunctionsRemoving a Function DefinitionThe return CommandThe type Command13 Rolo RevisitedData Formatting ConsiderationsroloaddludisplayremchangelistallSample Output14 Interactive and Nonstandard Shell FeaturesGetting the Right ShellThe ENV FileCommand-Line EditingCommand HistoryThe vi Line Edit ModeAccessing Commands from Your HistoryThe emacs Line Edit ModeAccessing Commands from Your HistoryOther Ways to Access Your HistoryThe history CommandThe fc CommandThe r CommandFunctionsLocal VariablesAutomatically Loaded FunctionsInteger ArithmeticInteger TypesNumbers in Different BasesThe alias CommandRemoving AliasesArraysJob ControlStopped Jobs and the fg and bg CommandsMiscellaneous FeaturesOther Features of the cd CommandTilde SubstitutionOrder of SearchCompatibility SummaryA Shell SummaryStartupCommandsCommentsParameters and VariablesShell

VariablesPositional ParametersSpecial ParametersParameter SubstitutionCommand Re-entryThe fc Commandvi Line Edit ModeQuotingTilde SubstitutionArithmetic ExpressionsFilename SubstitutionI/O RedirectionExported Variables and Subshell ExecutionThe (...) ConstructThe { ...; } ConstructMore on Shell VariablesFunctionsJob ControlShell JobsStopping JobsCommand SummaryThe : CommandThe . CommandThe alias CommandThe bg CommandThe break CommandThe case CommandThe cd CommandThe continue CommandThe echo CommandThe eval CommandThe exec CommandThe exit CommandThe export CommandThe false CommandThe fc CommandThe fg CommandThe for CommandThe getopts CommandThe hash CommandThe if CommandThe jobs CommandThe kill CommandThe newgrp CommandThe pwd CommandThe read CommandThe readonly CommandThe return CommandThe set CommandThe shift CommandThe test CommandThe times CommandThe trap CommandThe true CommandThe type CommandThe umask CommandThe unalias CommandThe unset CommandThe until CommandThe wait CommandThe while CommandB For More InformationOnline DocumentationDocumentation on the WebBooksO'Reilly & AssociatesPearsonIndexTable of ContentsIntroductionHow This Book Is OrganizedAccessing the Free Web Edition1 A Quick Review of the BasicsSome Basic CommandsDisplaying the Date and Time: The date CommandFinding Out Who's Logged In: The who CommandEchoing Characters: The echo CommandWorking with FilesListing Files: The ls CommandDisplaying the Contents of a File: The cat CommandCounting the Number of Words in a File: The wc CommandCommand OptionsMaking a Copy of a File: The cp CommandRenaming a File: The mv CommandRemoving a File: The rm CommandWorking with DirectoriesThe Home Directory and PathnamesDisplaying Your Working Directory: The pwd CommandChanging Directories: The cd CommandMore on the ls CommandCreating a Directory: The mkdir CommandCopying a File from One Directory to AnotherMoving Files Between DirectoriesLinking Files: The ln CommandRemoving a Directory: The rmdir CommandFilename SubstitutionThe AsteriskMatching Single CharactersFilename NuancesSpaces in FilenamesOther Weird CharactersStandard Input/Output, and I/O RedirectionStandard Input and Standard OutputOutput RedirectionInput RedirectionPipesFiltersStandard ErrorMore on CommandsTyping More Than One Command on a LineSending a Command to the BackgroundThe ps CommandCommand Summary2 What Is the Shell?The Kernel and the UtilitiesThe Login ShellTyping Commands to the ShellThe Shell's ResponsibilitiesProgram ExecutionVariable and Filename SubstitutionI/O RedirectionHooking up a PipelineEnvironment ControlInterpreted Programming Language3 Tools of the TradeRegular ExpressionsMatching Any Character: The Period (.)Matching the Beginning of the Line: The Caret (^)Matching the End of the Line: The Dollar Sign $Matching a Character Set: The [...] ConstructMatching Zero or More Characters: The Asterisk (*)Matching a Precise Number of Subpatterns: \{...\}Saving Matched Characters: \(...\)cutThe -d and -f OptionspasteThe -d OptionThe -s OptionsedThe -n OptionDeleting LinestrThe -s OptionThe -d OptiongrepRegular Expressions and grepThe -v OptionThe -l OptionThe -n OptionsortThe -u

OptionThe -r OptionThe -o OptionThe -n OptionSkipping FieldsThe -t OptionOther OptionsuniqThe -d OptionOther Options4 And Away We GoCommand FilesCommentsVariablesDisplaying the Values of VariablesUndefined Variables Have the Null ValueFilename Substitution and VariablesThe ${variable} ConstructBuilt-in Integer Arithmetic5 Can I Quote You on That?The Single QuoteThe Double QuoteThe BackslashUsing the Backslash for Continuing LinesThe Backslash Inside Double QuotesCommand SubstitutionThe Back QuoteThe $(...) ConstructThe expr Command6 Passing ArgumentsThe $# VariableThe $* VariableA Program to Look Up Someone in the Phone BookA Program to Add Someone to the Phone BookA Program to Remove Someone from the Phone Book${n}The shift Command7 Decisions, DecisionsExit StatusThe $? VariableThe test CommandString OperatorsAn Alternative Format for testInteger OperatorsFile OperatorsThe Logical Negation Operator !The Logical AND Operator -aParenthesesThe Logical OR Operator -oThe else ConstructThe exit CommandA Second Look at the rem ProgramThe elif ConstructYet Another Version of remThe case CommandSpecial Pattern-Matching CharactersThe -x Option for Debugging ProgramsBack to the caseThe Null Command :The && and || Constructs8 'Round and 'Round She GoesThe for CommandThe $@ VariableThe for Without the ListThe while CommandThe until CommandMore on LoopsBreaking Out of a LoopSkipping the Remaining Commands in a LoopExecuting a Loop in the BackgroundI/O Redirection on a LoopPiping Data into and out of a LoopTyping a Loop on One LineThe getopts Command9 Reading and Printing DataThe read CommandA Program to Copy FilesSpecial echo Escape CharactersAn Improved Version of mycpA Final Version of mycpA Menu-Driven Phone ProgramThe $$ Variable and Temporary FilesThe Exit Status from readThe printf Command10 Your EnvironmentLocal VariablesSubshellsExported Variablesexport -pPS1 and PS2HOMEPATHYour Current DirectoryCDPATHMore on SubshellsThe .CommandThe exec CommandThe (...) and { ...; } ConstructsAnother Way to Pass Variables to a SubshellYour .profile FileThe TERM VariableThe TZ Variable11 More on ParametersParameter Substitution${parameter}${parameter:-value}${parameter:=value}${parameter:?value}${parameter:+value}Pattern Matching Constructs ${#variable}The $0 VariableThe set CommandThe -x Optionset with No ArgumentsUsing set to Reassign Positional ParametersThe -- OptionOther Options to setThe IFS VariableThe readonly CommandThe unset Command12 Loose EndsThe eval CommandThe wait CommandThe $! VariableThe trap Commandtrap with No ArgumentsIgnoring SignalsResetting TrapsMore on I/O<&- and >&-In-line Input RedirectionShell ArchivesFunctionsRemoving a Function DefinitionThe return CommandThe type Command13 Rolo RevisitedData Formatting ConsiderationsroloaddludisplayremchangelistallSample Output14 Interactive and Nonstandard Shell FeaturesGetting the Right ShellThe ENV FileCommand-Line EditingCommand HistoryThe vi Line Edit ModeAccessing Commands from Your HistoryThe emacs Line Edit ModeAccessing Commands from Your HistoryOther Ways to Access Your HistoryThe history CommandThe fc CommandThe r CommandFunctionsLocal VariablesAutomatically Loaded FunctionsInteger ArithmeticInteger TypesNumbers in Different BasesThe alias CommandRemoving

AliasesArraysJob ControlStopped Jobs and the fg and bg CommandsMiscellaneous FeaturesOther Features of the cd CommandTilde SubstitutionOrder of SearchCompatibility SummaryA Shell SummaryStartupCommandsCommentsParameters and VariablesShell VariablesPositional ParametersSpecial ParametersParameter SubstitutionCommand Re-entryThe fc Commandvi Line Edit ModeQuotingTilde SubstitutionArithmetic ExpressionsFilename SubstitutionI/O RedirectionExported Variables and Subshell ExecutionThe (...) ConstructThe { ...; } ConstructMore on Shell VariablesFunctionsJob ControlShell JobsStopping JobsCommand SummaryThe : CommandThe . CommandThe alias CommandThe bg CommandThe break CommandThe case CommandThe cd CommandThe continue CommandThe echo CommandThe eval CommandThe exec CommandThe exit CommandThe export CommandThe false CommandThe fc CommandThe fg CommandThe for CommandThe getopts CommandThe hash CommandThe if CommandThe jobs CommandThe kill CommandThe newgrp CommandThe pwd CommandThe read CommandThe readonly CommandThe return CommandThe set CommandThe shift CommandThe test CommandThe times CommandThe trap CommandThe true CommandThe type CommandThe umask CommandThe unalias CommandThe unset CommandThe until CommandThe wait CommandThe while CommandB For More InformationOnline DocumentationDocumentation on the WebBooksO’Reilly & AssociatesPearsonIndexAbout the AuthorsStephen Kochan is the author or co-author of several best-selling titles on Unix and the C language, including Programming in C, Programming in Objective-C, Topics in C Programming, and Exploring the Unix System. He is a former software consultant for AT&T Bell Laboratories, where he developed and taught classes on Unix and C programming.Patrick Wood is the CTO of the New Jersey location of Electronics for Imaging. He was a member of the technical staff at Bell Laboratories when he met Mr. Kochan in 1985. Together they founded Pipeline Associates, Inc., a Unix consulting firm, where he was vice president. They co-authored Exploring the Unix System, Unix System Security, Topics in C Programming, and Unix Shell Programming.About the AuthorsStephen Kochan is the author or co-author of several best-selling titles on Unix and the C language, including Programming in C, Programming in Objective-C, Topics in C Programming, and Exploring the Unix System. He is a former software consultant for AT&T Bell Laboratories, where he developed and taught classes on Unix and C programming.Patrick Wood is the CTO of the New Jersey location of Electronics for Imaging. He was a member of the technical staff at Bell Laboratories when he met Mr. Kochan in 1985. Together they founded Pipeline Associates, Inc., a Unix consulting firm, where he was vice president. They co-authored Exploring the Unix System, Unix System Security, Topics in C Programming, and Unix Shell Programming.We Want to Hear from You!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.We welcome your comments. You can email or write directly to let us know what you did or didn’t like about this book—as well as what we can do to make our books

better.Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we 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 email address.Email: feedback@developerslibrary.infoMail: Reader Feedback Addison-Wesley Developer's Library 800 East 96th Street Indianapolis, IN 46240 USAWe Want to Hear from You!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.We welcome your comments. You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books better.Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we 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 email address.Email: feedback@developerslibrary.infoMail: Reader Feedback Addison-Wesley Developer's Library 800 East 96th Street Indianapolis, IN 46240 USAReader ServicesVisit our website and register this book at for convenient access to any updates, downloads, or errata that might be available for this book.Reader ServicesVisit our website and register this book at for convenient access to any updates, downloads, or errata that might be available for this book.IntroductionIt's no secret that the family of Unix and Unix-like operating systems has emerged over the last few decades as the most pervasive, most widely used group of operating systems in computing today. For programmers who have been using Unix for many years, this came as no surprise: The Unix system provides an elegant and efficient environment for program development. That's exactly what Dennis Ritchie and Ken Thompson sought to create when they developed Unix at Bell Laboratories way back in the late 1960s.NoteThroughout this book we'll use the term Unix to refer generically to the broad family of Unix-based operating systems, including true Unix operating systems such as Solaris as well as Unix-like operating systems such as Linux and Mac OS X.One of the strongest features of the Unix system is its wide collection of programs. More than 200 basic commands are distributed with the standard operating system and Linux adds to it, often shipping with 700–1000 standard commands! These commands (also known as tools) do everything from counting the number of lines in a file, to sending electronic mail, to displaying a calendar for any desired year.But the real strength of the Unix system comes not from its large collection of commands but from the elegance and ease with which these commands can be combined to perform far more sophisticated tasks.The standard user interface to Unix is the command line, which actually turns out to be a shell, a program that acts as a buffer between the user and the lowest levels of the system itself (the kernel). The shell is simply a program that reads in the commands you type and converts them into a form more readily understood by the system. It also includes core programming constructs that let you make decisions, loop, and store values in variables.The standard shell distributed

with Unix systems derives from AT&T's distribution, which evolved from a version originally written by Stephen Bourne at Bell Labs. Since then, the IEEE has created standards based on the Bourne shell and the other more recent shells. The current version of this standard, as of this writing, is the Shell and Utilities volume of IEEE Std 1003.1-2001, also known as the POSIX standard. This shell is what we use as the basis for the rest of this book.The examples in this book were tested on a Mac running Mac OS X 10.11, Ubuntu Linux 14.0, and an old version of SunOS 5.7 running on a Sparcstation Ultra-30. All examples, with the exception of some Bash examples in Chapter 14, were run using the Korn shell, although all of them also work fine with Bash.Because the shell offers an interpreted programming language, programs can be written, modified, and debugged quickly and easily. We turn to the shell as our first choice of programming language and after you become adept at shell programming, you will too.How This Book Is OrganizedThis book assumes that you are familiar with the fundamentals of the system and command line; that is, that you know how to log in; how to create files, edit them, and remove them; and how to work with directories. In case you haven't used the Linux or Unix system for a while, we'll examine the basics in Chapter 1, "A Quick Review of the Basics." In addition, filename substitution, I/O redirection, and pipes are also reviewed in the first chapter.Chapter 2, "What Is the Shell?," reveals what the shell really is, how it works, and how it ends up being your primary method of interacting with the operating system itself. You'll learn about what happens every time you log in to the system, how the shell program gets started, how it parses the command line, and how it executes other programs for you. A key point made in Chapter 2 is that the shell is just another program; nothing more, nothing less.Chapter 3, "Tools of the Trade," provides tutorials on tools useful in writing shell programs. Covered in this chapter are cut, paste, sed, grep, sort, tr, and uniq. Admittedly, the selection is subjective, but it does set the stage for programs that we'll develop throughout the remainder of the book. Also in Chapter 3 is a detailed discussion of regular expressions, which are used by many Unix commands, such as sed, grep, and ed.Chapters 4 through 9 teach you how to put the shell to work for writing programs. You'll learn how to write your own commands; use variables; write programs that accept arguments; make decisions; use the shell's for, while, and until looping commands; and use the read command to read data from the terminal or from a file. Chapter 5, "Can I Quote you on That?", is devoted entirely to a discussion of one of the most intriguing (and often confusing) aspects of the shell: the way it interprets quotes.By that point in the book, all the basic programming constructs in the shell will have been covered, and you will be able to write shell programs to solve your particular problems.Chapter 10, "Your Environment," covers a topic of great importance for a real understanding of the way the shell operates: the environment. You'll learn about local and exported variables; subshells; special shell variables, such as HOME, PATH, and CDPATH; and how to set up your .profile file.Chapter 11, "More on Parameters," and Chapter 12, "Loose Ends," tie up some loose ends, and Chapter 13, "Rolo Revisited," presents a final version of a phone directory program called rolo that is developed throughout the book.Chapter 14, "Interactive and Nonstandard Shell Features," discusses

features of the shell that either are not formally part of the IEEE POSIX standard shell (but are available in most Unix and Linux shells) or are mainly used interactively instead of in programs.Appendix A, “Shell Summary,” summarizes the features of the IEEE POSIX standard shell.Appendix B, “For More Information,” lists references and resources, including the Web sites where different shells can be downloaded.The philosophy this book uses is to teach by example. We believe that properly chosen examples do a far better job of illustrating how a particular feature is used than ten times as many words. The old “A picture is worth ...” adage seems to apply just as well to coding.We encourage you to type in each example and test it on your own system, for only by doing can you become adept at shell programming. Don’t be afraid to experiment. Try changing commands in the program examples to see the effect, or add different options or features to make the programs more useful or robust.Accessing the Free Web EditionYour purchase of this book in any format includes access to the corresponding Web Edition, which provides several special features to help you learn: The complete text of the book online Interactive quizzes and exercises to test your understanding of the material Updates and corrections as they become availableThe Web Edition can be viewed on all types of computers and mobile devices with any modern web browser that supports HTML5.To get access to the Web Edition of Shell Programming with Unix, Linux, and OS X all you need to do is register this book:1. Go to .2. Sign in or create a new account.4. Answer the questions as proof of purchase.The Web Edition will appear under the Digital Purchases tab on your Account page. Click the Launch link to access the product.IntroductionIt’s no secret that the family of Unix and Unix-like operating systems has emerged over the last few decades as the most pervasive, most widely used group of operating systems in computing today. For programmers who have been using Unix for many years, this came as no surprise: The Unix system provides an elegant and efficient environment for program development. That’s exactly what Dennis Ritchie and Ken Thompson sought to create when they developed Unix at Bell Laboratories way back in the late 1960s.NoteThroughout this book we’ll use the term Unix to refer generically to the broad family of Unix-based operating systems, including true Unix operating systems such as Solaris as well as Unix-like operating systems such as Linux and Mac OS X.NoteThroughout this book we’ll use the term Unix to refer generically to the broad family of Unix-based operating systems, including true Unix operating systems such as Solaris as well as Unix-like operating systems such as Linux and Mac OS X.One of the strongest features of the Unix system is its wide collection of programs. More than 200 basic commands are distributed with the standard operating system and Linux adds to it, often shipping with 700–1000 standard commands! These commands (also known as tools) do everything from counting the number of lines in a file, to sending electronic mail, to displaying a calendar for any desired year.But the real strength of the Unix system comes not from its large collection of commands but from the elegance and ease with which these commands can be combined to perform far more sophisticated tasks.The standard user interface to Unix is the command line, which actually turns out to be a shell, a program that acts as a buffer between the user and the lowest levels of the system itself (the kernel). The shell

is simply a program that reads in the commands you type and converts them into a form more readily understood by the system. It also includes core programming constructs that let you make decisions, loop, and store values in variables.The standard shell distributed with Unix systems derives from AT&T's distribution, which evolved from a version originally written by Stephen Bourne at Bell Labs. Since then, the IEEE has created standards based on the Bourne shell and the other more recent shells. The current version of this standard, as of this writing, is the Shell and Utilities volume of IEEE Std 1003.1-2001, also known as the POSIX standard. This shell is what we use as the basis for the rest of this book.The examples in this book were tested on a Mac running Mac OS X 10.11, Ubuntu Linux 14.0, and an old version of SunOS 5.7 running on a Sparcstation Ultra-30. All examples, with the exception of some Bash examples in Chapter 14, were run using the Korn shell, although all of them also work fine with Bash.Because the shell offers an interpreted programming language, programs can be written, modified, and debugged quickly and easily. We turn to the shell as our first choice of programming language and after you become adept at shell programming, you will too.How This Book Is OrganizedThis book assumes that you are familiar with the fundamentals of the system and command line; that is, that you know how to log in; how to create files, edit them, and remove them; and how to work with directories. In case you haven't used the Linux or Unix system for a while, we'll examine the basics in Chapter 1, "A Quick Review of the Basics." In addition, filename substitution, I/O redirection, and pipes are also reviewed in the first chapter.How This Book Is OrganizedThis book assumes that you are familiar with the fundamentals of the system and command line; that is, that you know how to log in; how to create files, edit them, and remove them; and how to work with directories. In case you haven't used the Linux or Unix system for a while, we'll examine the basics in Chapter 1, "A Quick Review of the Basics." In addition, filename substitution, I/O redirection, and pipes are also reviewed in the first chapter.Chapter 2, "What Is the Shell?," reveals what the shell really is, how it works, and how it ends up being your primary method of interacting with the operating system itself. You'll learn about what happens every time you log in to the system, how the shell program gets started, how it parses the command line, and how it executes other programs for you. A key point made in Chapter 2 is that the shell is just another program; nothing more, nothing less.Chapter 3, "Tools of the Trade," provides tutorials on tools useful in writing shell programs. Covered in this chapter are cut, paste, sed, grep, sort, tr, and uniq. Admittedly, the selection is subjective, but it does set the stage for programs that we'll develop throughout the remainder of the book. Also in Chapter 3 is a detailed discussion of regular expressions, which are used by many Unix commands, such as sed, grep, and ed.Chapters 4 through 9 teach you how to put the shell to work for writing programs. You'll learn how to write your own commands; use variables; write programs that accept arguments; make decisions; use the shell's for, while, and until looping commands; and use the read command to read data from the terminal or from a file. Chapter 5, "Can I Quote you on That?", is devoted entirely to a discussion of one of the most intriguing (and often confusing) aspects of the shell: the way it interprets quotes.By that point in the book, all the basic programming constructs in the

shell will have been covered, and you will be able to write shell programs to solve your particular problems.Chapter 10, “Your Environment,” covers a topic of great importance for a real understanding of the way the shell operates: the environment. You’ll learn about local and exported variables; subshells; special shell variables, such as HOME, PATH, and CDPATH; and how to set up your .profile file.Chapter 11, “More on Parameters,” and Chapter 12, “Loose Ends,” tie up some loose ends, and Chapter 13, “Rolo Revisited,” presents a final version of a phone directory program called rolo that is developed throughout the book.Chapter 14, “Interactive and Nonstandard Shell Features,” discusses features of the shell that either are not formally part of the IEEE POSIX standard shell (but are available in most Unix and Linux shells) or are mainly used interactively instead of in programs.Appendix A, “Shell Summary,” summarizes the features of the IEEE POSIX standard shell.Appendix B, “For More Information,” lists references and resources, including the Web sites where different shells can be downloaded.The philosophy this book uses is to teach by example. We believe that properly chosen examples do a far better job of illustrating how a particular feature is used than ten times as many words. The old “A picture is worth ...” adage seems to apply just as well to coding.We encourage you to type in each example and test it on your own system, for only by doing can you become adept at shell programming. Don’t be afraid to experiment. Try changing commands in the program examples to see the effect, or add different options or features to make the programs more useful or robust.Accessing the Free Web EditionYour purchase of this book in any format includes access to the corresponding Web Edition, which provides several special features to help you learn:Accessing the Free Web EditionYour purchase of this book in any format includes access to the corresponding Web Edition, which provides several special features to help you learn: The complete text of the book online Interactive quizzes and exercises to test your understanding of the material Updates and corrections as they become availableThe Web Edition can be viewed on all types of computers and mobile devices with any modern web browser that supports HTML5.To get access to the Web Edition of Shell Programming with Unix, Linux, and OS X all you need to do is register this book:1. Go to .2. Sign in or create a new account.3. Enter ISBN: 9780134496009.4. Answer the questions as proof of purchase.The Web Edition will appear under the Digital Purchases tab on your Account page. Click the Launch link to access the product.1. A Quick Review of the BasicsThis chapter provides a review of the Unix system, including the file system, basic commands, filename substitution, I/O redirection, and pipes.Some Basic CommandsDisplaying the Date and Time: The date CommandThe date command tells the system to print the date and time:$ dateThu Dec 3 11:04:09 MST 2015$date prints the day of the week, month, day, time (24-hour clock, the system’s time zone), and year. Throughout the code examples in this book, whenever we use boldface type like this, it’s to indicate what you, the user, type in. Normal face type like this is used to indicate what the Unix system prints. Italic type is used for comments in interactive sequences.Every Unix command is submitted to the system with the pressing of the Enter key. Enter says that you are finished typing things in and are ready for the Unix system to do its thing.Finding Out Who’s Logged In:

The who CommandThe who command can be used to get information about all users currently logged in to the system:Click here to view code image$ whopat tty29 Jul 19 14:40ruth tty37 Jul 19 10:54steve tty25 Jul 19 15:52$Here, three users are logged in: pat, ruth, and steve. Along with each user ID is listed the tty number of that user and the day and time that user logged in. The tty number is a unique identification number the Unix system gives to each terminal or network device that a user is on when they log into the system.The who command also can be used to get information about yourself:Click here to view code image$ who am ipat tty29 Jul 19 14:40$who and who am i are actually the same command: who. In the latter case, the am and i are arguments to the who command. (This isn't a good example of how command arguments work; it's just a curiosity of the who command.)Echoing Characters: The echo CommandThe echo command prints (or echoes) at the terminal whatever else you happen to type on the line (there are some exceptions to this that you'll learn about later):Click here to view code image$ echo this is a testthis is a test$ echo why not print out a longer line with echo? why not print out a longer line with echo?$ echo A blank line is displayed$ echo one two three four fiveone two three four five$You will notice from the preceding example that echo squeezes out extra blanks between words. That's because on a Unix system, the words are important while the blanks are only there to separate the words. Generally, the Unix system ignores extra blanks (you'll learn more about this in the next chapter).Working with FilesThe Unix system recognizes only three basic types of files: ordinary files, directory files, and special files. An ordinary file is just that: any file on the system that contains data, text, program instructions, or just about anything else. Directories, or folders, are described later in this chapter. Finally, as its name implies, a special file has a special meaning to the Unix system and is typically associated with some form of I/O.A filename can be composed of just about any character directly available from the keyboard (and even some that aren't) provided that the total number of characters contained in the name is not greater than 255. If more than 255 characters are specified, the Unix system simply ignores the extra characters.The Unix system provides many tools that make working with files easy. Here we'll review some of the basic file manipulation commands.Listing Files: The ls CommandTo see what files you have stored in your directory, you can type the ls command:$ lsREAD_MEnamestmp$This output indicates that three files called READ_ME, names, and tmp are contained in the current directory. (Note that the output of ls may vary from system to system. For example, on many Unix systems ls produces multicolumn output when sending its output to a terminal; on others, different colors may be used for different types of files. You can always force single-column output with the –1 option—that's the number one.)Displaying the Contents of a File: The cat CommandYou can examine the contents of a file by using the cat command. (That's short for "concatenate," if you're thinking feline thoughts.) The argument to cat is the name of the file whose contents you want to examine.$ cat namesSusanJeffHenryAllanKen $Counting the Number of Words in a File: The wc CommandWith the wc command, you can get a count of the total number of lines, words, and characters contained in a file. Once again, the

name of the file is expected to be specified as the argument to this command:Click here to view code image$ wc names 5 7 27 names$The wc command lists three numbers followed by the filename. The first number represents the number of lines in the file (5), the second the number of words (7), and the third the number of characters (27).Command OptionsMost Unix commands allow the specification of options at the time a command is executed. These options generally follow the same format:-letterThat is, a command option is a minus sign followed immediately by a single letter. For example, to count just the number of lines contained in a file, the option -l (that's the letter l) is given to the wc command:$ wc -l names 5 names$To count just the number of characters in a file, the -c option is specified:$ wc -c names 27 names$Finally, the -w option can be used to count the number of words contained in the file:$ wc -w names 7 names$Some commands require that the options be listed before the filename arguments. For example, sort names -r is acceptable, whereas wc names -l is not. Still, the former is unusual and most Unix commands are designed for you to specify command options first, as exemplified by wc -l names.Making a Copy of a File: The cp CommandTo make a copy of a file, use the cp command. The first argument to the command is the name of the file to be copied (known as the source file), and the second argument is the name of the file to place the copy into (known as the destination file). You can make a copy of the file names and call it saved_names as follows:$ cp names saved_names$Execution of this command causes the contents of the file names to be copied into a new file named saved_names. As with many Unix commands, the fact that no output other than a command prompt was displayed after the cp command was typed indicates that the command executed successfully.Renaming a File: The mv CommandA file can be renamed with the mv ("move") command. The arguments to the mv command follow the same format as the cp command. The first argument is the name of the file to be renamed, and the second argument is the new name. So, to change the name of the file saved_names to hold_it, for example, the following command would do the trick:$ mv saved_names hold_it$Be careful! When executing an mv or cp command, the Unix system does not care whether the file specified as the second argument already exists. If it does, the contents of the file will be lost. For example, if a file called old_names exists, executing the commandcp names old_nameswould copy the filenames to old_names, destroying the previous contents of old_names in the process. Similarly, the commandmv names old_nameswould rename names to old_names, even if the file old_names existed prior to execution of the command.Removing a File: The rm CommandUse the rm command to remove a file from the system. The argument to rm is simply the name of the file to be removed:$ rm hold_it$You can remove more than one file at a time with the rm command by simply specifying all such files on the command line. For example, the following would remove the three files wb, collect, and mon:$ rm wb collect mon$Working with DirectoriesSuppose that you had a set of files consisting of various memos, proposals, and letters. Further suppose that you had another set of files that were computer programs. It would seem logical to group this first set into a directory called documents and the latter into a directory called programs. Figure 1.1

illustrates such a directory organization.Figure 1.1 Example directory structureThe file directory documents contains the files plan, dact, sys.A, new.hire, no.JSK, and AMG.reply. The directory programs contains the files wb, collect, and mon. At some point, you may decide to further categorize the files in a directory. This can be done by creating subdirectories and then placing each file into the appropriate subdirectory. For example, you might want to create subdirectories called memos, proposals, and letters inside your documents directory, as shown in Figure 1.2.Figure 1.2 Directories containing subdirectoriesdocuments contains the subdirectories memos, proposals, and letters. Each of these subdirectories in turn contains two files: memos contains plan and dact; proposals contains sys.A and new.hire; and letters contains no.JSK and AMG.reply.Although each file in a given directory must have a unique name, files contained in different directories do not. So you could have a file in your programs directory called dact, even though a file by that name also exists in the memos subdirectory.The Home Directory and PathnamesThe Unix system always associates each user of the system with a particular directory. When you log in to the system, you are placed automatically into your own directory (called your home directory).Although the location of users' home directories can vary from one system to the next, let's assume that your home directory is called steve and that this directory is actually a subdirectory of a directory called users. Therefore, if you had the directories documents and programs, the overall directory structure would actually look something like Figure 1.3. A special directory named / (pronounced "slash") is shown at the top of the directory tree. This directory is known as the root.Figure 1.3 Hierarchical directory structureWhenever you are "inside" a particular directory (called your current working directory), the files contained within that directory are immediately accessible, without specifying any path information. If you want to access a file from another directory, you can either first issue a command to "change" to the appropriate directory and then access the particular file, or you can specify the particular file by its pathname.A pathname enables you to uniquely identify a particular file to the Unix system. In the specification of a pathname, successive directories along the path are separated by the slash character /. A pathname that begins with a slash character is known as a full or absolute pathname because it specifies a complete path from the root. For example, the pathname /users/steve identifies the directory steve contained within the directory users. Similarly, the pathname /users/steve/documents references the directory documents as contained in the directory steve within users. As a final example, the pathname /users/steve/documents/letters/AMG.reply identifies the file AMG.reply contained along the appropriate directory path.To help reduce the typing that would otherwise be required, Unix provides certain notational conveniences. A pathname that does not begin with a slash is known as a relative pathname: the path is relative to your current working directory. For example, if you just logged in to the system and were placed into your home directory /users/steve, you could directly reference the directory documents simply by typing documents. Similarly, the relative pathname programs/mon could be typed to access the file mon contained inside your programs directory.By convention, .. always references the directory that is one level higher than the current directory, known as the

parent directory. For example, if you were in your home directory /users/steve, the pathname .. would reference the directory users. If you had issued the appropriate command to change your working directory to documents/letters, the pathname .. would reference the documents directory, ../.. would reference the directory steve, and ../proposals/new.hire would reference the file new.hire contained in the proposals directory. There is usually more than one way to specify a path to a particular file, a very Unix-y characteristic.Another notational convention is the single period., which always refers to the current directory. That'll become more important later in the book when you want to specify a shell script in the current directory, not one in the PATH. We'll explain this in more detail soon.Displaying Your Working Directory: The pwd CommandThe pwd command is used to help you "get your bearings" by telling you the name of your current working directory.Recall the directory structure from Figure 1.3. The directory that you are placed in after you log in to the system is called your home directory. You can assume from Figure 1.3 that the home directory for the user steve is /users/steve. Therefore, whenever steve logs in to the system, he will automatically be placed inside this directory. To verify that this is the case, the pwd (print working directory) command can be issued:$ pwd/users/steve$The output from the command verifies that steve's current working directory is /users/steve.Changing Directories: The cd CommandYou can change your current working directory by using the cd command. This command takes as its argument the name of the target or destination directory.Let's assume that you just logged in to the system and were placed in your home directory, /users/steve. This is depicted by the arrow in Figure 1.4.Figure 1.4 Current working directory is steveYou know that two directories are directly "below" steve's home directory: documents and programs. This can be easily verified at the terminal by issuing the ls command:$ lsdocumentsprograms$The ls command lists the two directories documents and programs the same way it listed other ordinary files in previous examples.To change your current working directory, issue the cd command, followed by the name of the new directory:$ cd documents$After executing this command, you will be placed inside the documents directory, as depicted in Figure 1.5.Figure 1.5 cd documentsYou can verify at the terminal that the working directory has been changed by using the pwd command:$ pwd/users/steve/documents$The easiest way to move up one level in a directory is to reference the .. shortcut with the commandcd ..because by convention .. always refers to the directory one level up (see Figure 1.6).$ cd ..$ pwd/users/steve$Figure 1.6 cd ..If you wanted to change to the letters directory, you could get there with a single cd command by specifying the relative path documents/letters (see Figure 1.7):Click here to view code image$ cd documents/letters$ pwd/users/steve/documents/letters$Figure 1.7 cd documents/lettersYou can get back up to your home directory by using a single cd command to go up two directories as shown:$ cd ../..$ pwd/users/steve$Or you can get back to the home directory using a full pathname rather than a relative one:$ cd /users/steve$ pwd/users/steve$Finally, there is a third way to get back to the home directory that is also the easiest. Typing the command cd without an argument always moves you back to your home directory, no matter where you are in the file system:$ cd$ pwd/users/steve$More on the ls CommandWhen you type the ls command, the

files contained in the current working directory are listed. But you can also use ls to obtain a list of files in other directories by supplying an argument to the command. First let's get back to your home directory:$ cd$ pwd/users/steve$Now let's take a look at the files in the current working directory:$ lsdocumentsprograms$If you supply the name of one of these directories to the ls command, you can get a list of the contents of that directory. So you can find out what's contained in the documents directory by typing the command ls documents:$ ls documentslettersmemosproposals$To take a look at the subdirectory memos, you can follow a similar procedure:$ ls documents/memosdactplan$If you specify a nondirectory file argument to the ls command, you simply get that filename echoed back at the terminal:$ ls documents/memos/plandocuments/memos/plan$Confused? There's an option to the ls command that lets you determine whether a particular file is a directory, among other things. The -l option (the letter l) provides a more detailed description of the files in a directory. If you were currently in steve's home directory, here's what the -l option to the ls command produces:Click here to view code image$ ls –ltotal 2drwxr-xr-x 5 steve DP3725 80 Jun 25 13:27 documentsdrwxr-xr-x 2 steve DP3725 96 Jun 25 13:31 programs$The first line of the display is a count of the total number of blocks (1,024 bytes) of storage that the listed files use. Each successive line displayed by the ls -l command contains detailed information about a file in the directory. The first character on each line indicates what type of file it is: d for a directory, - for a file, b, c, l, or p for a special file.The next nine characters on the line define the access permissions of that particular file or directory. These access modes apply to the file's owner (the first three characters), other users in the same group as the file's owner (the next three characters), and finally all other users on the system (the last three characters). Generally, they indicate whether the specified class of user can read the file, write to the file, or execute the contents of the file (in the case of a program or shell script).The ls -l command then shows the link count (see "Linking Files: The ln Command," later in this chapter), the owner of the file, the group owner of the file, how large the file is (that is, how many characters are contained in it), and when the file was last modified. The information displayed last on the line is the filename itself.NoteMany modern Unix systems have gone away from using groups, so while those permissions are still shown, the group owner for a specific file or directory is often omitted in the output of the ls command.You should now be able to glean a lot of information from the ls -l output for a directory full of files:Click here to view code image$ ls -l programstotal 4-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 1 steve DP3725 89 Jun 25 13:30 wb$The dash in the first column of each line indicates that the three files collect, mon, and wb are ordinary files and not directories. Now, can you figure out how big are they?Creating a Directory: The mkdir CommandUse mkdir to create directories. The argument to this command is simply the name of the directory you want to create. For example, assume that you are still working with the directory structure depicted in Figure 1.7 and that you want to create a new directory called misc at the same level as the directories documents and programs. If you were currently in your home directory, typing the command mkdir misc would

achieve the desired effect:$ mkdir misc$Now if you run ls, you will have the new directory listed: $ lsdocumentsmiscprograms$The directory structure now appears as shown in Figure 1.8.Figure 1.8 Directory structure with newly created misc directoryCopying a File from One Directory to AnotherThe cp command can be used to copy a file from one directory into another. For example, you can copy the file wb from the programs directory into a file called wbx in the misc directory as follows:$ cp programs/wb misc/wbx$Because the two files are in different directories, they can safely have the exact same name:$ cp programs/wb misc/wb$When the destination file is going to have the same name as the source file (in a different directory, of course), it is necessary to specify only the destination directory as the second argument:$ cp programs/wb misc$When this command gets executed, the Unix system recognizes that the second argument is a directory and copies the source file into that directory. The new file is given the same name as the source file.You can copy more than one file into a directory by listing the files to be copied prior to the name of the destination directory. If you were currently in the programs directory, the command$ cp wb collect mon ../misc$would copy the three files wb, collect, and mon into the misc directory, with the same filenames.To copy a file from another directory into your current location in the file system and give the file the same name, use the handy "." shortcut for the current directory:$ pwd/users/steve/misc$ cp ../programs/collect .$The preceding command copies the file collect from the directory ../programs into the current directory (/users/steve/misc).Moving Files Between DirectoriesYou recall that the mv command can be used to rename a file. Indeed, there is no "rename" command in Unix. However, when the two arguments reference different directories, the file is actually moved from the first directory into the second.To demonstrate, go from the home directory to the documents directory:$ cd documents$Suppose that now you decide that the file plan contained in the memos directory is really a proposal so you want to move it from the memos directory into the proposals directory. The following would do the trick:Click here to view code image$ mv memos/plan proposals/plan $As with the cp command, if the source file and destination file have the same name, only the name of the destination directory need be supplied, so there's an easier way to move this file:$ mv memos/plan proposals$Also like the cp command, a group of files can be simultaneously moved into a directory by simply listing all files to be moved before the name of the destination directory:$ pwd/users/steve/programs$ mv wb collect mon ../misc$This would move the three files wb, collect, and mon into the directory misc.You can also use the mv command to change the name of a directory, as it happens. For example, the following renames the programs directory to bin.$ mv programs bin$Linking Files: The ln CommandSo far everything we've talked about with file management has assumed that a given collection of data has one and only one filename, wherever it may be located in the file system. It turns out that Unix is more sophisticated than that and can assign multiple filenames to the same collection of data.The main command for creating these duplicate names for a given file is the ln command.The general form of the command isln from toThis links the file from to the file to.Recall the structure of steve's programs directory from Figure 1.8. In that directory, he has stored a program called

wb. Suppose that he decides that he'd also like to call the program writeback. The most obvious thing to do would be to simply create a copy of wb called writeback:$ cp wb writeback$The drawback with this approach is that now twice as much disk space is being consumed by the program. Furthermore, if steve ever changes wb, he may forget to duplicate the change in writeback, resulting in two different, out of sync copies of what he thinks is the same program. Not so good, Steve!By linking the file wb to the new name, these problems are avoided:$ ln wb writeback$Now instead of two copies of the file existing, only one exists with two different names: wb and writeback. The two files have been logically linked by the Unix system.As far as you're concerned, it appears as though you have two different files. Executing an ls command shows the two files separately:$ lscollectmonwbwriteback$Where it gets interesting is when you use ls -l:Click here to view code image$ ls -ltotal 5-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 writeback $Look closely at the second column of the output: The number shown is 1 for collect and mon and 2 for wb and writeback. This is the number of links to a file, normally 1 for nonlinked, nondirectory files. Because wb and writeback are linked, however, this number is 2 for these files (or, more correctly, this file with two names).You can remove either of the two linked files at any time, and the other will not be removed:Click here to view code image$ rm writeback$ ls -ltotal 4-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 1 steve DP3725 89 Jun 25 13:30 wb$Note that the number of links on wb went from 2 to 1 because one of its links was removed.Most often, ln is used to allow a file to appear in more than one directory simultaneously. For example, suppose that pat wanted to have access to steve's wb program. Instead of making a copy for himself (subject to the same data sync problems described previously) or including steve's programs directory in his PATH (which has security risks as described in Chapter 10, "Your Environment"), he can simply link to the file from his own program directory:Click here to view code image$ pwd/users/pat/bin pat's program directory$ ls -ltotal 4-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ ln /users/steve/wb . link wb to pat's bin$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr $Note that steve is still listed as the owner of wb, even when viewing the contents of pat's directory. This makes sense, because there's really only one copy of the file and it's owned by steve.The only stipulation on linking files is that for ordinary links the files to be linked together must reside on the same file system. If they don't, you'll get an error from ln when you try to link them. (To determine the different file systems on your system, execute the df command. The first field on each line of output is the name of a file system.)To create links to files on different file systems (or on different networked systems), you can use the -s option to the ln command. This creates a symbolic link. Symbolic links behave a lot like regular links, except that the symbolic link points to the original file; if the original file is removed, the symbolic link no longer works.Let's

see how symbolic links work with the previous example:Click here to view code image$ rm wb$ ls -ltotal 4-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ ln -s /users/steve/wb ./symwb Symbolic link to wb$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcatlrwxr-xr-x 1 pat DP3822 15 Jul 20 15:22 symwb -> /users/steve/wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$Note that pat is listed as the owner of symwb, and the file type shown as the very first character in the ls output is l, which indicates a symbolic link. The size of the symbolic link is 15 (the file actually contains the string /users/steve/wb), but if we attempt to access the contents of the file, we are presented with the contents of the file it's linked to, /users/steve/wb:Click here to view code image$ wc symwb 5 9 89 symwb$The -L option to the ls command can be used with the -l option to get a detailed list of information on the file the symbolic link points to:Click here to view code image$ ls -Lltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr $Removing the file that a symbolic link points to invalidates the symbolic link (because symbolic links are maintained as filenames), but it doesn't remove it:Click here to view code image$ rm /users/steve/wb Assume pat can remove this file$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcatlrwxr-xr-x 1 pat DP3822 15 Jul 20 15:22 wb -> /users/steve/wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ wc wbCannot open wb: No such file or directory$This type of file is called a dangling symbolic link and should be removed unless you have a specific reason to keep it around (for example, if you intend to replace the removed file).One last note before leaving this discussion: The ln command follows the same general format as cp and mv, meaning that you can create links to a bunch of files within a specific target directory using the formatln files directoryRemoving a Directory: The rmdir CommandYou can remove a directory with the rmdir command. Rather than let you accidentally remove dozens or hundreds of files, however, rmdir won't let you proceed unless the specified directory is completely empty of files and subdirectories.To remove the directory /users/pat, we could use the following:Click here to view code image$ rmdir /users/patrmdir: pat: Directory not empty $Phew! That would have been a mistake! Instead, let's remove the misc directory that you created earlier:$ rmdir /users/steve/misc$Once again, the preceding command works only if no files or directories are contained in the misc directory; otherwise, the following happens, as also shown earlier:Click here to view code image$ rmdir /users/steve/miscrmdir: /users/steve/misc: Directory not empty$If you still want to remove the misc directory, you would first have to remove all the files contained in that directory before reissuing the rmdir command.As an alternative method for removing a directory and its contents, you can use the -r option to the rm command. The format is simple:rm -r dirwhere dir is the name of the directory that you want to remove. rm removes the indicated directory and all files (including directories) in it, so be careful with this powerhouse command.Want to go full turbo? Add the -f flag and it forces the action without prompting you on a command-by-command basis. It can completely trash your system if you're not careful, however, so many admins simply avoid rm -rf entirely!Filename SubstitutionThe

AsteriskOne powerful feature of the Unix system that is handled by the shell is filename substitution. Let's say that your current directory has these files in it:$ lschaptlchapt2chapt3chapt4$Suppose that you want to display their contents en masse. Easy: cat allows you to display the contents of as many files as you specify on the command line. Like this:Click here to view code image$ cat chaptl chapt2 chapt3 chapt4 ...$But that's tedious. Instead, you can take advantage of filename substitution by simply typing:$ cat * ...$The shell automatically substitutes the names of all the files in the current directory that match the pattern *. The same substitution occurs if you use * with another command too, of course. How about echo?$ echo *chaptl chapt2 chapt3 chapt4$Here the * is again replaced with the names of all the files contained in the current directory, and the echo command simply displays that list to you.Any place that * appears on the command line, the shell performs its substitution:Click here to view code image$ echo * : *chaptl chapt2 chapt3 chapt4 : chaptl chapt2 chapt3 chapt4$The * is part of a rich file substitution language, actually, and it can also be used in combination with other characters to limit which filenames are matched.For example, let's say that in your current directory you have not only chapt1 through chapt4 but also files a, b, and c:$ lsabcchaptlchapt2chapt3chapt4$To display the contents of just the files beginning with chap, you can type in$ cat chap* . . .$The chap* matches any filename that begins with chap. All such filenames matched are substituted on the command line before the specified command is even invoked.The * is not limited to the end of a filename; it can be used at the beginning or in the middle as well:$ echo *t1chaptl$ echo *t*chaptl chapt2 chapt3 chapt4$ echo *x*x$In the first echo, the *t1 specifies all filenames that end in the characters t1. In the second echo, the first * matches everything up to a t and the second everything after; thus, all filenames containing a t are printed. Because there are no files ending with x, no substitution occurs in the last case. Therefore, the echo command simply displays *x.Matching Single CharactersThe asterisk (*) matches zero or more characters, meaning that x* matches the file x as well as x1, x2, xabc, and so on. The question mark (?) matches exactly one character. So cat? will display all files that have filenames of exactly one character, just as cat x? prints all files with two-character names beginning with x. Here we see this behavior illustrated again with echo:Click here to view code image$ lsaaaaaxalicebbbcccreport1report2report3$ echo ?a b c$ echo a?aa$ echo ??aa bb cc $ echo ??*aa aax alice bb cc report1 report2 report3$In the preceding example, the ?? matches two characters, and the * matches zero or more characters up to the end. The net effect is to match all filenames of two or more characters.Another way to match a single character is to give a list of characters to match within square brackets [ ]. For example, [abc] matches the letter a, b, or c. It's similar to the ?, but it allows you to choose which characters are valid matches.You can also specify a logical range of characters with a dash, a huge convenience! For example, [0–9] matches the characters 0 through 9. The only restriction in specifying a range of characters is that the first character must be alphabetically less than the last character, so that [z–f] is not a valid range specification, while [f–z] is.By mixing and matching ranges and characters in the list, you can perform complicated substitutions. For example, [a–np–z]* matches all files that start

with the letters a through n or p through z (or more simply stated, any filename that doesn't start with the lowercase letter o).If the first character following the [is a !, the sense of the match is inverted. That is, any character is matched except those enclosed in the brackets. So[!a–z]matches any character except a lowercase letter, and*[!o]matches any file that doesn't end with the lowercase letter o.Table 1.1 gives a few more examples of filename substitution.Table 1.1 Filename Substitution ExamplesFilename NuancesSpaces in FilenamesA discussion of command lines and filenames wouldn't be complete without talking about the bane of old-school Unix people and very much the day-to-day reality of Linux, Windows, and Mac users: spaces in filenames.The problem arises from the fact that the shell uses spaces as delimiters between words. In other words the phrase echo hi mom is properly parsed as an invocation to the command echo, with two arguments hi and mom.Now imagine you have a file called my test document. How do you reference it from the command line? How do you view it or display it using the cat command?Click here to view code image$ cat my test documentcat: my: No such file or directorycat: test: No such file or directorycat: document: No such file or directoryThat definitely doesn't work. Why? Because cat wants a filename to be specified and instead of seeing one, it sees three: my, test, and document.There are two standard solutions for this: Either escape every space by using a backslash, or wrap the entire filename in quotes so that the shell understands that it's a single word with spaces, rather than multiple words.Click here to view code image$ cat "my test document"This is a test document and is fullof scintillating information to edifyand amaze.$ cat my\ test\ documentThis is a test document and is fullof scintillating information to edifyand amaze.That solves the problem and is critical to know as you proceed with file systems that quite likely have lots of directories and files that contain spaces as part of their filenames.Other Weird CharactersWhile the space might be the most difficult and annoying of special characters that can appear in filenames, occasionally you'll find others show up that can throw a proverbial monkey-wrench into your command line efforts.For example, how would you deal with a filename that contains a question mark? In the next section, you'll learn that the character “?” has a specific meaning to the shell. Most modern shells are smart enough to sidestep the duplication of meaning, but, again, quoting the filename or using backslashes to denote that the special character is part of the filename is required:Click here to view code image $ ls -l who\ me\?-rw-r--r-- 1 taylor staff 0 Dec 4 10:18 who me?Where this really gets interesting is if you have a backslash or quote as part of the filename, something that can happen inadvertently, particularly for files created by graphically oriented programs on a Linux or Mac system. The trick? Use single quotes to escape a filename that includes a double quote, and vice versa. Like this:Click here to view code image$ ls -l "don't quote me" 'She said "yes"'-rw-r--r-- 1 taylor staff 0 Dec 4 10:18 don't quote me-rw-r--r-- 1 taylor staff 0 Dec 4 10:19 She said "yes"This topic will come up again as we proceed, but now you know how to side-step problems with directories or files that contain spaces or other non-standard characters.Standard Input/Output, and I/O RedirectionStandard Input and Standard OutputMost Unix system commands take input from your screen and send the resulting output back to your screen. In

Unix nomenclature, the screen is generally called the terminal, a reference that harkens back to the earliest days of computing. Nowadays it's more likely to be a terminal program you're running within a graphical environment, whether it's a Linux window manager, a Windows computer, or a Mac system.A command normally reads its input from standard input, which is your computer keyboard by default. It's a fancy way of clarifying that you "type in" your information. Similarly, a command normally writes its output to standard output, which is also your terminal or terminal app by default. This concept is depicted in Figure 1.9.Figure 1.9 Typical Unix commandAs an example, recall that executing the who command results in the display of all users that are currently logged-in. More formally, the who command writes a list of the logged-in users to standard output. This is depicted in Figure 1.10.Figure 1.10 who commandIt turns out that just about every single Unix command can take the output of a previous command or file as its input too, and can even send its output to another command or program. This concept is hugely important to understanding the power of the command line and why it's so helpful to know all of these commands even when a graphical interface might be also available for your use.Before we get there, however, consider this: if the sort command is invoked without a filename argument, the command takes its input from standard input. As with standard output, this is your terminal (or keyboard) by default.When input is entered this way, an end-of-file sequence must be specified after the last line is typed, and, by Unix convention, that's Ctrl+d; that is, the sequence produced by simultaneously pressing the Control (or Ctrl, depending on your keyboard) key and the d key.As an example, let's use the sort command to sort the following four names: Tony, Barbara, Harry, Dirk. Instead of first entering the names into a file, we'll enter them directly from the terminal:$ sortTonyBarbaraHarryDirkCtrl+dBarbaraDirkHarryTony$Because no filename was specified to the sort command, the input was taken from standard input, the terminal. After the fourth name was typed in, the Ctrl and d keys were pressed to signal the end of the data. At that point, the sort command sorted the four names and displayed the results on the standard output device, which is also the terminal. This is depicted in Figure 1.11.Figure 1.11 sort commandThe wc command is another example of a command that takes its input from standard input if no filename is specified on the command line. The following shows an example of this command used to count the number of lines of text entered from the terminal:$ wc -lThis is text thatis typed on thestandard input device.Ctrl+d 3$Note that the Ctrl+d that is used to terminate the input is not counted as a separate line by the wc command because it's interpreted by the shell, not handed to the command. Furthermore, because the -l flag was specified to the wc command, only the count of the number of lines (3) is presented as the output of the command.Output RedirectionThe output from a command normally intended for standard output can be easily "diverted" to a file instead. This capability is known as output redirection and is also essential to understanding the power of Unix.If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to the file file instead:$ who > users$This command causes the who command to be executed and its output to be written into the file users. Notice that no output

appears. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. We can check this, of course:$ cat usersoko tty01 Sep 12 07:30ai tty15 Sep 12 13:32ruth tty21 Sep 12 10:10pat tty24 Sep 12 13:07steve tty25 Sep 12 13:03$If a command has its output redirected to a file and the file already contains some data, that data will be overwritten and lost.$ echo line 1 > users$ cat usersline 1$But now consider this example, remembering that users already contains the output of the earlier who command:$ echo line 2 >> users$ cat usersline 1line 2$If you're paying close attention you'll notice that this echo command uses a different type of output redirection, indicated by the characters >>. This character pair causes the standard output from the command to be appended to the contents of the specified file. The previous contents are not lost; the new output simply gets added to the end.By using the redirection append characters >>, you can use cat to append the contents of one file onto the end of another:Click here to view code image$ cat file1This is in file1.$ cat file2This is in file2.$ cat file1 >> file2 Append file1 to file2$ cat file2This is in file2.This is in file1.$Recall that specifying more than one filename to cat results in the display of the first file followed immediately by the second file, and so on. This means there's a second way to accomplish the same result:Click here to view code image$ cat file1This is in file1.$ cat file2This is in file2.$ cat file1 file2This is in file1.This is in file2.$ cat file1 file2 > file3 Redirect it instead$ cat file3This is in filel.This is in file2.$In fact, that's where the cat command gets its name: When used with more than one file, its effect is to concatenate the files together.Input RedirectionJust as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. And as the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command. Of course, only commands that normally take their input from standard input can have their input redirected from a file in this manner.To redirect input, type the < character followed by the name of the file that the input is to be read from. To count the number of lines in the file users, for example, you already know that you can execute the command wc -l users:$ wc -l users 2 users$It turns out that you can also count the number of lines in the file by redirecting standard input for the wc command:$ wc -l < users 2$Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.This points out a subtle distinction between the execution of the two commands. In the first case, wc knows that it is reading its input from the file users. In the second case, it only sees the raw data which is being fed to it via standard input. The shell redirects the input so that it comes from the file users and not the terminal (more about this in the next chapter). As far as wc is concerned, it doesn't know whether its input is coming from the terminal or from a file, so it can't report the filename!PipesAs you will recall, the file users that was created previously contains a list of all the users currently logged in to the system. Because you know that there will be one line in the file for each user logged in to the system, you can easily determine the number of login sessions by counting the number of lines in the users file:$ who > users$ wc -l < users 5$This output indicates that currently five users

are logged in or that there are five login sessions, the difference being that users, particularly administrators, often log in more than once. Now you have a command sequence you can use whenever you want to know how many users are logged in.Another approach to determine the number of logged-in users bypasses the intermediate file. As referenced earlier, Unix lets you “connect” two commands together. This connection is known as a pipe, and it enables you to take the output from one command and feed it directly into the input of another. A pipe is denoted by the character |, which is placed between the two commands. To create a pipe between the who and wc -l commands, you type who | wc -l:$ who | wc -l 5$The pipe that is created between these two commands is depicted in Figure 1.12.Figure 1.12 Pipeline process: who | wc -lWhen a pipe is established between two commands, the standard output from the first command is connected directly to the standard input of the second command. You know that the who command writes its list of logged-in users to standard output. Furthermore, you know that if no filename argument is specified to the wc command, it takes its input from standard input. Therefore, the list of logged-in users that is output from the who command automatically becomes the input to the wc command. Note that you never see the output of the who command at the terminal because it is piped directly into the wc command. This is depicted in Figure 1.13.Figure 1.13 Pipeline processA pipe can be made between any two programs, provided that the first program writes its output to standard output, and the second program reads its input from standard input.As another example, suppose you wanted to count the number of files contained in your directory. Knowledge of the fact that the ls command displays one line of output per file enables you to use the same type of approach as before:$ ls | wc -l 10$The output indicates that the current directory contains 10 files.It is also possible to create a more complicated pipeline that consists of more than two programs, with the output of one program feeding into the input of the next. As you become a more sophisticated command line user, you’ll find many situations where pipelines can be tremendously powerful.FiltersThe term filter is often used in Unix terminology to refer to any program that can take input from standard input, perform some operation on that input, and write the results to standard output. More succinctly, a filter is any program that can be used to modify the output of other programs in a pipeline. So in the pipeline in the previous example, wc is considered a filter. ls is not because it does not read its input from standard input. As other examples, cat and sort are filters, whereas who, date, cd, pwd, echo, rm, mv, and cp are not.Standard ErrorIn addition to standard input and standard output, there is a third virtual device known as standard error. This is where most Unix commands write their error messages. And as with the other two “standard” places, standard error is associated with your terminal or terminal app by default. In most cases, you never know the difference between standard output and standard error:Click here to view code image$ ls n* List all files beginning with nn* not found$Here the “not found” message is actually being written to standard error by the ls command. You can verify that this message is not being written to standard output by redirecting the ls command’s output:$ ls n* > foon* not found$As you can see, the message is still printed out at the terminal and was not added to the

file foo, even though you redirected standard output.The preceding example shows the raison d'être for standard error: so that error messages will still get displayed at the terminal even if standard output is redirected to a file or piped to another command.You can also redirect standard error to a file (for instance, if you're logging a program's potential errors during long-term operation) by using the slightly more complex notationcommand 2> fileNote that no space is permitted between the 2 and the >. Any error messages normally intended for standard error will be diverted into the specified file, similar to the way standard output gets redirected.$ ls n* 2> errors$ cat errorsn* not found$More on CommandsTyping More Than One Command on a LineYou can type more than one command on a line provided that you separate them with a semicolon. For example, you can find out the current time and your current working directory by typing in the date and pwd commands on the same line:$ date; pwdSat Jul 20 14:43:25 EDT 2002/users/pat/bin$You can string out as many commands as you want on the line, as long as each command is delimited by a semicolon.Sending a Command to the BackgroundNormally, you type in a command and then wait for the results of the command to be displayed at the terminal. For all the examples you have seen thus far, this waiting time is typically short—a fraction of a second.Sometimes, however, you may have to run commands that require a few minutes or longer to complete. In those cases, you'll have to wait for the command to finish executing before you can proceed further, unless you execute the command in the background.It turns out that while your Unix or Linux system seems like it's focused completely on what you're doing, all systems are actually multitasking, running multiple commands simultaneously at any given time. If you're on an Ubuntu system, for example, it might have the window manager, a clock, a status monitor and your terminal window all running simultaneously. You too can run multiple commands simultaneously from the command line. That's the idea of putting a command "into background," letting you work on other tasks while it completes.The notational convention for pushing a command or command sequence into background is to append the ampersand character &. This means that the command will no longer tie up your terminal, and you can then proceed with other work. The standard output from the command will still be directed to your terminal, though in most cases the standard input will be dissociated from your terminal. If the command does try to read from standard input, it will stop and wait for you to bring it to the foreground (we'll discuss this in more detail in Chapter 14, "Interactive and Nonstandard Shell Features").Here's an example:Click here to view code image$ sort bigdata > out & Send the sort to the background[1] 1258 Process id$ date Your terminal is immediately available to do other workSat Jul 20 14:45:09 EDT 2002$When a command is sent to the background, the Unix system automatically displays two numbers. The first is called the command's job number and the second the process ID, or PID. In the preceding example, 1 is the job number and 1258 the process ID. The job number is used as a shortcut for referring to a specific background job by some shell commands. (You'll learn more about this in Chapter 14.) The process ID uniquely identifies the command that you sent to the background and can be used to obtain status information about the command. This is done with the

processor status—ps—command.The ps CommandThe ps command gives you information about the processes running on the system. Without any options, it prints the status of just your processes. If you type in ps at your terminal, you'll get a few lines back describing the processes you have running:$ ps PID TTY TIME CMD13463 pts/16 00:00:09 bash19880 pts/16 00:00:00 ps$The ps command (typically; your system might vary) prints out four columns of information: PID, the process ID; TTY, the terminal number that the process was run from; TIME, the amount of computer time in minutes and seconds that process has used; and CMD, the name of the process. (The bash process in the preceding example is the shell that was started when we logged in, and it's used 9 seconds of computer time.) Until the command is finished, it shows up in the output of the ps command as a running process, so process 19880 in the preceding example is the ps command itself.When used with the -f option, ps prints out more information about your processes, including the parent process ID (PPID), the time the process started (STIME), and the command arguments:Click here to view code image$ ps -fUID PID PPID C STIME TTY TIME CMDsteve 13463 13355 0 12:12 pts/16 00:00:09 bashsteve 19884 13463 0 13:39 pts/16 00:00:00 ps -f$Command SummaryTable 1.2 summarizes the commands reviewed in this chapter. In this table, file refers to a file, file(s) to one or more files, dir to a directory, and dir(s) to one or more directories.Table 1.2 Command Summary1. A Quick Review of the BasicsThis chapter provides a review of the Unix system, including the file system, basic commands, filename substitution, I/O redirection, and pipes.Some Basic CommandsDisplaying the Date and Time: The date CommandThe date command tells the system to print the date and time:Some Basic CommandsDisplaying the Date and Time: The date CommandThe date command tells the system to print the date and time:Displaying the Date and Time: The date CommandThe date command tells the system to print the date and time:$ dateThu Dec 3 11:04:09 MST 2015$date prints the day of the week, month, day, time (24-hour clock, the system's time zone), and year. Throughout the code examples in this book, whenever we use boldface type like this, it's to indicate what you, the user, type in. Normal face type like this is used to indicate what the Unix system prints. Italic type is used for comments in interactive sequences.Every Unix command is submitted to the system with the pressing of the Enter key. Enter says that you are finished typing things in and are ready for the Unix system to do its thing.Finding Out Who's Logged In: The who CommandThe who command can be used to get information about all users currently logged in to the system:Finding Out Who's Logged In: The who CommandThe who command can be used to get information about all users currently logged in to the system:Click here to view code image$ whopat tty29 Jul 19 14:40ruth tty37 Jul 19 10:54steve tty25 Jul 19 15:52$Here, three users are logged in: pat, ruth, and steve. Along with each user ID is listed the tty number of that user and the day and time that user logged in. The tty number is a unique identification number the Unix system gives to each terminal or network device that a user is on when they log into the system.The who command also can be used to get information about yourself:Click here to view code image$ who am ipat tty29 Jul 19 14:40$who and who am i are actually the

same command: who. In the latter case, the am and i are arguments to the who command. (This isn't a good example of how command arguments work; it's just a curiosity of the who command.)Echoing Characters: The echo CommandThe echo command prints (or echoes) at the terminal whatever else you happen to type on the line (there are some exceptions to this that you'll learn about later):Echoing Characters: The echo CommandThe echo command prints (or echoes) at the terminal whatever else you happen to type on the line (there are some exceptions to this that you'll learn about later):Click here to view code image$ echo this is a testthis is a test $ echo why not print out a longer line with echo?why not print out a longer line with echo?$ echo A blank line is displayed$ echo one two three four fiveone two three four five$You will notice from the preceding example that echo squeezes out extra blanks between words. That's because on a Unix system, the words are important while the blanks are only there to separate the words. Generally, the Unix system ignores extra blanks (you'll learn more about this in the next chapter).Working with FilesThe Unix system recognizes only three basic types of files: ordinary files, directory files, and special files. An ordinary file is just that: any file on the system that contains data, text, program instructions, or just about anything else. Directories, or folders, are described later in this chapter. Finally, as its name implies, a special file has a special meaning to the Unix system and is typically associated with some form of I/O.Working with FilesThe Unix system recognizes only three basic types of files: ordinary files, directory files, and special files. An ordinary file is just that: any file on the system that contains data, text, program instructions, or just about anything else. Directories, or folders, are described later in this chapter. Finally, as its name implies, a special file has a special meaning to the Unix system and is typically associated with some form of I/O.A filename can be composed of just about any character directly available from the keyboard (and even some that aren't) provided that the total number of characters contained in the name is not greater than 255. If more than 255 characters are specified, the Unix system simply ignores the extra characters.The Unix system provides many tools that make working with files easy. Here we'll review some of the basic file manipulation commands.Listing Files: The ls CommandTo see what files you have stored in your directory, you can type the ls command:Listing Files: The ls CommandTo see what files you have stored in your directory, you can type the ls command:$ lsREAD_MEnamestmp $This output indicates that three files called READ_ME, names, and tmp are contained in the current directory. (Note that the output of ls may vary from system to system. For example, on many Unix systems ls produces multicolumn output when sending its output to a terminal; on others, different colors may be used for different types of files. You can always force single-column output with the –1 option—that's the number one.)Displaying the Contents of a File: The cat CommandYou can examine the contents of a file by using the cat command. (That's short for "concatenate," if you're thinking feline thoughts.) The argument to cat is the name of the file whose contents you want to examine.Displaying the Contents of a File: The cat CommandYou can examine the contents of a file by using the cat command. (That's short for "concatenate," if you're thinking feline thoughts.) The argument to cat is the name of the file whose contents you

want to examine.$ cat namesSusanJeffHenryAllanKen$Counting the Number of Words in a File: The wc CommandWith the wc command, you can get a count of the total number of lines, words, and characters contained in a file. Once again, the name of the file is expected to be specified as the argument to this command:Counting the Number of Words in a File: The wc CommandWith the wc command, you can get a count of the total number of lines, words, and characters contained in a file. Once again, the name of the file is expected to be specified as the argument to this command:Click here to view code image$ wc names 5 7 27 names $The wc command lists three numbers followed by the filename. The first number represents the number of lines in the file (5), the second the number of words (7), and the third the number of characters (27).Command OptionsMost Unix commands allow the specification of options at the time a command is executed. These options generally follow the same format:Command OptionsMost Unix commands allow the specification of options at the time a command is executed. These options generally follow the same format:-letterThat is, a command option is a minus sign followed immediately by a single letter. For example, to count just the number of lines contained in a file, the option -l (that's the letter l) is given to the wc command:$ wc -l names 5 names$To count just the number of characters in a file, the -c option is specified:$ wc -c names 27 names$Finally, the -w option can be used to count the number of words contained in the file:$ wc -w names 7 names$Some commands require that the options be listed before the filename arguments. For example, sort names -r is acceptable, whereas wc names -l is not. Still, the former is unusual and most Unix commands are designed for you to specify command options first, as exemplified by wc -l names.Making a Copy of a File: The cp CommandTo make a copy of a file, use the cp command. The first argument to the command is the name of the file to be copied (known as the source file), and the second argument is the name of the file to place the copy into (known as the destination file). You can make a copy of the file names and call it saved_names as follows:Making a Copy of a File: The cp CommandTo make a copy of a file, use the cp command. The first argument to the command is the name of the file to be copied (known as the source file), and the second argument is the name of the file to place the copy into (known as the destination file). You can make a copy of the file names and call it saved_names as follows:$ cp names saved_names$Execution of this command causes the contents of the file names to be copied into a new file named saved_names. As with many Unix commands, the fact that no output other than a command prompt was displayed after the cp command was typed indicates that the command executed successfully.Renaming a File: The mv CommandA file can be renamed with the mv ("move") command. The arguments to the mv command follow the same format as the cp command. The first argument is the name of the file to be renamed, and the second argument is the new name. So, to change the name of the file saved_names to hold_it, for example, the following command would do the trick:Renaming a File: The mv CommandA file can be renamed with the mv ("move") command. The arguments to the mv command follow the same format as the cp command. The first argument is the name of the file to be renamed, and the second argument is the new name. So, to change the name of

the file saved_names to hold_it, for example, the following command would do the trick:$ mv saved_names hold_it$Be careful! When executing an mv or cp command, the Unix system does not care whether the file specified as the second argument already exists. If it does, the contents of the file will be lost. For example, if a file called old_names exists, executing the commandcp names old_nameswould copy the filenames to old_names, destroying the previous contents of old_names in the process. Similarly, the commandmv names old_nameswould rename names to old_names, even if the file old_names existed prior to execution of the command.Removing a File: The rm CommandUse the rm command to remove a file from the system. The argument to rm is simply the name of the file to be removed:Removing a File: The rm CommandUse the rm command to remove a file from the system. The argument to rm is simply the name of the file to be removed:$ rm hold_it$You can remove more than one file at a time with the rm command by simply specifying all such files on the command line. For example, the following would remove the three files wb, collect, and mon:$ rm wb collect mon$Working with DirectoriesSuppose that you had a set of files consisting of various memos, proposals, and letters. Further suppose that you had another set of files that were computer programs. It would seem logical to group this first set into a directory called documents and the latter into a directory called programs. Figure 1.1 illustrates such a directory organization.Working with DirectoriesSuppose that you had a set of files consisting of various memos, proposals, and letters. Further suppose that you had another set of files that were computer programs. It would seem logical to group this first set into a directory called documents and the latter into a directory called programs. Figure 1.1 illustrates such a directory organization.Figure 1.1 Example directory structureFigure 1.1 Example directory structureThe file directory documents contains the files plan, dact, sys.A, new.hire, no.JSK, and AMG.reply. The directory programs contains the files wb, collect, and mon. At some point, you may decide to further categorize the files in a directory. This can be done by creating subdirectories and then placing each file into the appropriate subdirectory. For example, you might want to create subdirectories called memos, proposals, and letters inside your documents directory, as shown in Figure 1.2.Figure 1.2 Directories containing subdirectoriesFigure 1.2 Directories containing subdirectoriesdocuments contains the subdirectories memos, proposals, and letters. Each of these subdirectories in turn contains two files: memos contains plan and dact; proposals contains sys.A and new.hire; and letters contains no.JSK and AMG.reply.Although each file in a given directory must have a unique name, files contained in different directories do not. So you could have a file in your programs directory called dact, even though a file by that name also exists in the memos subdirectory.The Home Directory and PathnamesThe Unix system always associates each user of the system with a particular directory. When you log in to the system, you are placed automatically into your own directory (called your home directory).The Home Directory and PathnamesThe Unix system always associates each user of the system with a particular directory. When you log in to the system, you are placed automatically into your own directory (called your home directory).Although the location of users' home directories can vary from one system to the next, let's assume that your

home directory is called steve and that this directory is actually a subdirectory of a directory called users. Therefore, if you had the directories documents and programs, the overall directory structure would actually look something like Figure 1.3. A special directory named / (pronounced “slash”) is shown at the top of the directory tree. This directory is known as the root.Figure 1.3 Hierarchical directory structureFigure 1.3 Hierarchical directory structureWhenever you are “inside” a particular directory (called your current working directory), the files contained within that directory are immediately accessible, without specifying any path information. If you want to access a file from another directory, you can either first issue a command to “change” to the appropriate directory and then access the particular file, or you can specify the particular file by its pathname.A pathname enables you to uniquely identify a particular file to the Unix system. In the specification of a pathname, successive directories along the path are separated by the slash character /. A pathname that begins with a slash character is known as a full or absolute pathname because it specifies a complete path from the root. For example, the pathname /users/ steve identifies the directory steve contained within the directory users. Similarly, the pathname / users/steve/documents references the directory documents as contained in the directory steve within users. As a final example, the pathname /users/steve/documents/letters/AMG.reply identifies the file AMG.reply contained along the appropriate directory path.To help reduce the typing that would otherwise be required, Unix provides certain notational conveniences. A pathname that does not begin with a slash is known as a relative pathname: the path is relative to your current working directory. For example, if you just logged in to the system and were placed into your home directory /users/steve, you could directly reference the directory documents simply by typing documents. Similarly, the relative pathname programs/mon could be typed to access the file mon contained inside your programs directory.By convention, .. always references the directory that is one level higher than the current directory, known as the parent directory. For example, if you were in your home directory /users/steve, the pathname .. would reference the directory users. If you had issued the appropriate command to change your working directory to documents/letters, the pathname .. would reference the documents directory, ../.. would reference the directory steve, and ../proposals/new.hire would reference the file new.hire contained in the proposals directory. There is usually more than one way to specify a path to a particular file, a very Unix-y characteristic.Another notational convention is the single period., which always refers to the current directory. That’ll become more important later in the book when you want to specify a shell script in the current directory, not one in the PATH. We’ll explain this in more detail soon.Displaying Your Working Directory: The pwd CommandThe pwd command is used to help you “get your bearings” by telling you the name of your current working directory.Displaying Your Working Directory: The pwd CommandThe pwd command is used to help you “get your bearings” by telling you the name of your current working directory.Recall the directory structure from Figure 1.3. The directory that you are placed in after you log in to the system is called your home directory. You can assume from Figure 1.3 that the home directory for the user steve is /users/steve. Therefore, whenever steve logs in to the system, he will

automatically be placed inside this directory. To verify that this is the case, the pwd (print working directory) command can be issued:$ pwd/users/steve$The output from the command verifies that steve's current working directory is /users/steve.Changing Directories: The cd CommandYou can change your current working directory by using the cd command. This command takes as its argument the name of the target or destination directory.Changing Directories: The cd CommandYou can change your current working directory by using the cd command. This command takes as its argument the name of the target or destination directory.Let's assume that you just logged in to the system and were placed in your home directory, /users/steve. This is depicted by the arrow in Figure 1.4.Figure 1.4 Current working directory is steveFigure 1.4 Current working directory is steveYou know that two directories are directly "below" steve's home directory: documents and programs. This can be easily verified at the terminal by issuing the ls command:$ lsdocumentsprograms$The ls command lists the two directories documents and programs the same way it listed other ordinary files in previous examples.To change your current working directory, issue the cd command, followed by the name of the new directory:$ cd documents$After executing this command, you will be placed inside the documents directory, as depicted in Figure 1.5.Figure 1.5 cd documentsFigure 1.5 cd documentsYou can verify at the terminal that the working directory has been changed by using the pwd command:$ pwd/users/steve/documents$The easiest way to move up one level in a directory is to reference the .. shortcut with the commandcd ..because by convention .. always refers to the directory one level up (see Figure 1.6).$ cd ..$ pwd/users/steve$Figure 1.6 cd ..Figure 1.6 cd ..If you wanted to change to the letters directory, you could get there with a single cd command by specifying the relative path documents/letters (see Figure 1.7):Click here to view code image$ cd documents/letters$ pwd/users/steve/documents/letters$Figure 1.7 cd documents/lettersFigure 1.7 cd documents/lettersYou can get back up to your home directory by using a single cd command to go up two directories as shown:$ cd ../..$ pwd/users/steve$Or you can get back to the home directory using a full pathname rather than a relative one:$ cd /users/steve$ pwd/users/steve$Finally, there is a third way to get back to the home directory that is also the easiest. Typing the command cd without an argument always moves you back to your home directory, no matter where you are in the file system:$ cd$ pwd/users/steve$More on the ls CommandWhen you type the ls command, the files contained in the current working directory are listed. But you can also use ls to obtain a list of files in other directories by supplying an argument to the command. First let's get back to your home directory:More on the ls CommandWhen you type the ls command, the files contained in the current working directory are listed. But you can also use ls to obtain a list of files in other directories by supplying an argument to the command. First let's get back to your home directory:$ cd$ pwd/users/steve$Now let's take a look at the files in the current working directory:$ lsdocumentsprograms$If you supply the name of one of these directories to the ls command, you can get a list of the contents of that directory. So you can find out what's contained in the documents directory by typing the command ls documents:$ ls documentslettersmemosproposals$To take a look at the

subdirectory memos, you can follow a similar procedure:$ ls documents/memosdactplan$If you specify a nondirectory file argument to the ls command, you simply get that filename echoed back at the terminal:$ ls documents/memos/plandocuments/memos/plan$Confused? There's an option to the ls command that lets you determine whether a particular file is a directory, among other things. The -l option (the letter l) provides a more detailed description of the files in a directory. If you were currently in steve's home directory, here's what the -l option to the ls command produces:Click here to view code image$ ls –ltotal 2drwxr-xr-x 5 steve DP3725 80 Jun 25 13:27 documentsdrwxr-xr-x 2 steve DP3725 96 Jun 25 13:31 programs$The first line of the display is a count of the total number of blocks (1,024 bytes) of storage that the listed files use. Each successive line displayed by the ls -l command contains detailed information about a file in the directory. The first character on each line indicates what type of file it is: d for a directory, - for a file, b, c, l, or p for a special file.The next nine characters on the line define the access permissions of that particular file or directory. These access modes apply to the file's owner (the first three characters), other users in the same group as the file's owner (the next three characters), and finally all other users on the system (the last three characters). Generally, they indicate whether the specified class of user can read the file, write to the file, or execute the contents of the file (in the case of a program or shell script).The ls -l command then shows the link count (see "Linking Files: The ln Command," later in this chapter), the owner of the file, the group owner of the file, how large the file is (that is, how many characters are contained in it), and when the file was last modified. The information displayed last on the line is the filename itself.NoteMany modern Unix systems have gone away from using groups, so while those permissions are still shown, the group owner for a specific file or directory is often omitted in the output of the ls command.NoteMany modern Unix systems have gone away from using groups, so while those permissions are still shown, the group owner for a specific file or directory is often omitted in the output of the ls command.You should now be able to glean a lot of information from the ls -l output for a directory full of files:Click here to view code image$ ls -l programstotal 4-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 1 steve DP3725 89 Jun 25 13:30 wb$The dash in the first column of each line indicates that the three files collect, mon, and wb are ordinary files and not directories. Now, can you figure out how big are they?Creating a Directory: The mkdir CommandUse mkdir to create directories. The argument to this command is simply the name of the directory you want to create. For example, assume that you are still working with the directory structure depicted in Figure 1.7 and that you want to create a new directory called misc at the same level as the directories documents and programs. If you were currently in your home directory, typing the command mkdir misc would achieve the desired effect:Creating a Directory: The mkdir CommandUse mkdir to create directories. The argument to this command is simply the name of the directory you want to create. For example, assume that you are still working with the directory structure depicted in Figure 1.7 and that you want to create a new directory called misc at the same level as the directories documents and programs. If you were

currently in your home directory, typing the command mkdir misc would achieve the desired effect:$ mkdir misc$Now if you run ls, you will have the new directory listed:$ lsdocumentsmiscprograms$The directory structure now appears as shown in Figure 1.8.Figure 1.8 Directory structure with newly created misc directory

Figure 1.8 Directory structure with newly created misc directory

Copying a File from One Directory to AnotherThe cp command can be used to copy a file from one directory into another. For example, you can copy the file wb from the programs directory into a file called wbx in the misc directory as follows:

Copying a File from One Directory to AnotherThe cp command can be used to copy a file from one directory into another. For example, you can copy the file wb from the programs directory into a file called wbx in the misc directory as follows:

$ cp programs/wb misc/wbx$Because the two files are in different directories, they can safely have the exact same name:$ cp programs/wb misc/wb $When the destination file is going to have the same name as the source file (in a different directory, of course), it is necessary to specify only the destination directory as the second argument:$ cp programs/wb misc$When this command gets executed, the Unix system recognizes that the second argument is a directory and copies the source file into that directory. The new file is given the same name as the source file.You can copy more than one file into a directory by listing the files to be copied prior to the name of the destination directory. If you were currently in the programs directory, the command$ cp wb collect mon ../misc$would copy the three files wb, collect, and mon into the misc directory, with the same filenames.To copy a file from another directory into your current location in the file system and give the file the same name, use the handy "." shortcut for the current directory:$ pwd/users/steve/misc$ cp ../ programs/collect .$The preceding command copies the file collect from the directory ../ programs into the current directory (/users/steve/misc).Moving Files Between DirectoriesYou recall that the mv command can be used to rename a file. Indeed, there is no "rename" command in Unix. However, when the two arguments reference different directories, the file is actually moved from the first directory into the second.

Moving Files Between DirectoriesYou recall that the mv command can be used to rename a file. Indeed, there is no "rename" command in Unix. However, when the two arguments reference different directories, the file is actually moved from the first directory into the second.

To demonstrate, go from the home directory to the documents directory:$ cd documents$Suppose that now you decide that the file plan contained in the memos directory is really a proposal so you want to move it from the memos directory into the proposals directory. The following would do the trick:Click here to view code image$ mv memos/plan proposals/plan$As with the cp command, if the source file and destination file have the same name, only the name of the destination directory need be supplied, so there's an easier way to move this file:$ mv memos/plan proposals$Also like the cp command, a group of files can be simultaneously moved into a directory by simply listing all files to be moved before the name of the destination directory:$ pwd/users/steve/programs$ mv wb collect mon ../misc$This would move the three files wb, collect, and mon into the directory misc.You can also use the mv command to change the name of a directory, as it happens. For

example, the following renames the programs directory to bin.$ mv programs bin$Linking Files: The ln CommandSo far everything we've talked about with file management has assumed that a given collection of data has one and only one filename, wherever it may be located in the file system. It turns out that Unix is more sophisticated than that and can assign multiple filenames to the same collection of data.Linking Files: The ln CommandSo far everything we've talked about with file management has assumed that a given collection of data has one and only one filename, wherever it may be located in the file system. It turns out that Unix is more sophisticated than that and can assign multiple filenames to the same collection of data.The main command for creating these duplicate names for a given file is the ln command.The general form of the command isln from toThis links the file from to the file to.Recall the structure of steve's programs directory from Figure 1.8. In that directory, he has stored a program called wb. Suppose that he decides that he'd also like to call the program writeback. The most obvious thing to do would be to simply create a copy of wb called writeback:$ cp wb writeback$The drawback with this approach is that now twice as much disk space is being consumed by the program. Furthermore, if steve ever changes wb, he may forget to duplicate the change in writeback, resulting in two different, out of sync copies of what he thinks is the same program. Not so good, Steve!By linking the file wb to the new name, these problems are avoided:$ ln wb writeback$Now instead of two copies of the file existing, only one exists with two different names: wb and writeback. The two files have been logically linked by the Unix system.As far as you're concerned, it appears as though you have two different files. Executing an ls command shows the two files separately:$ lscollectmonwbwriteback$Where it gets interesting is when you use ls -l:Click here to view code image$ ls -ltotal 5-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 writeback $Look closely at the second column of the output: The number shown is 1 for collect and mon and 2 for wb and writeback. This is the number of links to a file, normally 1 for nonlinked, nondirectory files. Because wb and writeback are linked, however, this number is 2 for these files (or, more correctly, this file with two names).You can remove either of the two linked files at any time, and the other will not be removed:Click here to view code image$ rm writeback$ ls -ltotal 4-rwxr-xr-x 1 steve DP3725 358 Jun 25 13:31 collect-rwxr-xr-x 1 steve DP3725 1219 Jun 25 13:31 mon-rwxr-xr-x 1 steve DP3725 89 Jun 25 13:30 wb$Note that the number of links on wb went from 2 to 1 because one of its links was removed.Most often, ln is used to allow a file to appear in more than one directory simultaneously. For example, suppose that pat wanted to have access to steve's wb program. Instead of making a copy for himself (subject to the same data sync problems described previously) or including steve's programs directory in his PATH (which has security risks as described in Chapter 10, "Your Environment"), he can simply link to the file from his own program directory:Click here to view code image$ pwd/users/pat/bin pat's program directory$ ls -ltotal 4-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ ln /users/steve/wb .

link wb to pat's bin$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr $Note that steve is still listed as the owner of wb, even when viewing the contents of pat's directory. This makes sense, because there's really only one copy of the file and it's owned by steve.The only stipulation on linking files is that for ordinary links the files to be linked together must reside on the same file system. If they don't, you'll get an error from ln when you try to link them. (To determine the different file systems on your system, execute the df command. The first field on each line of output is the name of a file system.)To create links to files on different file systems (or on different networked systems), you can use the -s option to the ln command. This creates a symbolic link. Symbolic links behave a lot like regular links, except that the symbolic link points to the original file; if the original file is removed, the symbolic link no longer works.Let's see how symbolic links work with the previous example:Click here to view code image$ rm wb$ ls -ltotal 4-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ ln -s /users/steve/wb ./symwb Symbolic link to wb$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcatlrwxr-xr-x 1 pat DP3822 15 Jul 20 15:22 symwb -> /users/steve/wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$Note that pat is listed as the owner of symwb, and the file type shown as the very first character in the ls output is l, which indicates a symbolic link. The size of the symbolic link is 15 (the file actually contains the string /users/steve/wb), but if we attempt to access the contents of the file, we are presented with the contents of the file it's linked to, /users/steve/wb:Click here to view code image$ wc symwb 5 9 89 symwb$The -L option to the ls command can be used with the -l option to get a detailed list of information on the file the symbolic link points to:Click here to view code image$ ls -Lltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcat-rwxr-xr-x 2 steve DP3725 89 Jun 25 13:30 wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr $Removing the file that a symbolic link points to invalidates the symbolic link (because symbolic links are maintained as filenames), but it doesn't remove it:Click here to view code image$ rm /users/steve/wb Assume pat can remove this file$ ls -ltotal 5-rwxr-xr-x 1 pat DP3822 1358 Jan 15 11:01 lcatlrwxr-xr-x 1 pat DP3822 15 Jul 20 15:22 wb -> /users/steve/wb-rwxr-xr-x 1 pat DP3822 504 Apr 21 18:30 xtr$ wc wbCannot open wb: No such file or directory$This type of file is called a dangling symbolic link and should be removed unless you have a specific reason to keep it around (for example, if you intend to replace the removed file).One last note before leaving this discussion: The ln command follows the same general format as cp and mv, meaning that you can create links to a bunch of files within a specific target directory using the formatln files directoryRemoving a Directory: The rmdir CommandYou can remove a directory with the rmdir command. Rather than let you accidentally remove dozens or hundreds of files, however, rmdir won't let you proceed unless the specified directory is completely empty of files and subdirectories.Removing a Directory: The rmdir CommandYou can remove a directory with the rmdir command. Rather than let you accidentally remove dozens or hundreds of files, however, rmdir won't let you proceed unless the specified directory is

completely empty of files and subdirectories.To remove the directory /users/pat, we could use the following:Click here to view code image$ rmdir /users/patrmdir: pat: Directory not empty $Phew! That would have been a mistake! Instead, let's remove the misc directory that you created earlier:$ rmdir /users/steve/misc$Once again, the preceding command works only if no files or directories are contained in the misc directory; otherwise, the following happens, as also shown earlier:Click here to view code image$ rmdir /users/steve/miscrmdir: /users/steve/misc: Directory not empty$If you still want to remove the misc directory, you would first have to remove all the files contained in that directory before reissuing the rmdir command.As an alternative method for removing a directory and its contents, you can use the -r option to the rm command. The format is simple:rm -r dirwhere dir is the name of the directory that you want to remove. rm removes the indicated directory and all files (including directories) in it, so be careful with this powerhouse command.Want to go full turbo? Add the -f flag and it forces the action without prompting you on a command-by-command basis. It can completely trash your system if you're not careful, however, so many admins simply avoid rm -rf entirely!Filename SubstitutionThe AsteriskOne powerful feature of the Unix system that is handled by the shell is filename substitution. Let's say that your current directory has these files in it:Filename SubstitutionThe AsteriskOne powerful feature of the Unix system that is handled by the shell is filename substitution. Let's say that your current directory has these files in it:The AsteriskOne powerful feature of the Unix system that is handled by the shell is filename substitution. Let's say that your current directory has these files in it:$ lschaptlchapt2chapt3chapt4$Suppose that you want to display their contents en masse. Easy: cat allows you to display the contents of as many files as you specify on the command line. Like this:Click here to view code image$ cat chaptl chapt2 chapt3 chapt4 ...$But that's tedious. Instead, you can take advantage of filename substitution by simply typing:$ cat * ...$The shell automatically substitutes the names of all the files in the current directory that match the pattern *. The same substitution occurs if you use * with another command too, of course. How about echo?$ echo *chaptl chapt2 chapt3 chapt4$Here the * is again replaced with the names of all the files contained in the current directory, and the echo command simply displays that list to you.Any place that * appears on the command line, the shell performs its substitution:Click here to view code image$ echo * : *chaptl chapt2 chapt3 chapt4 : chaptl chapt2 chapt3 chapt4$The * is part of a rich file substitution language, actually, and it can also be used in combination with other characters to limit which filenames are matched.For example, let's say that in your current directory you have not only chapt1 through chapt4 but also files a, b, and c:$ lsabcchaptlchapt2chapt3chapt4$To display the contents of just the files beginning with chap, you can type in$ cat chap* . . .$The chap* matches any filename that begins with chap. All such filenames matched are substituted on the command line before the specified command is even invoked.The * is not limited to the end of a filename; it can be used at the beginning or in the middle as well:$ echo *t1chaptl$ echo *t*chaptl chapt2 chapt3 chapt4$ echo *x*x$In the first echo, the *t1 specifies all filenames that end in the characters t1. In the second echo, the first * matches everything up to a t and the second everything after; thus,

all filenames containing a t are printed. Because there are no files ending with x, no substitution occurs in the last case. Therefore, the echo command simply displays *x.Matching Single CharactersThe asterisk (*) matches zero or more characters, meaning that x* matches the file x as well as x1, x2, xabc, and so on. The question mark (?) matches exactly one character. So cat? will display all files that have filenames of exactly one character, just as cat x? prints all files with two-character names beginning with x. Here we see this behavior illustrated again with echo:Matching Single CharactersThe asterisk (*) matches zero or more characters, meaning that x* matches the file x as well as x1, x2, xabc, and so on. The question mark (?) matches exactly one character. So cat? will display all files that have filenames of exactly one character, just as cat x? prints all files with two-character names beginning with x. Here we see this behavior illustrated again with echo:Click here to view code image$ lsaaaaaxalicebbbcccreport1report2report3$ echo ?a b c$ echo a?aa$ echo ??aa bb cc$ echo ??*aa aax alice bb cc report1 report2 report3$In the preceding example, the ?? matches two characters, and the * matches zero or more characters up to the end. The net effect is to match all filenames of two or more characters.Another way to match a single character is to give a list of characters to match within square brackets [ ]. For example, [abc] matches the letter a, b, or c. It's similar to the ?, but it allows you to choose which characters are valid matches.You can also specify a logical range of characters with a dash, a huge convenience! For example, [0–9] matches the characters 0 through 9. The only restriction in specifying a range of characters is that the first character must be alphabetically less than the last character, so that [z–f] is not a valid range specification, while [f–z] is.By mixing and matching ranges and characters in the list, you can perform complicated substitutions. For example, [a–np–z]* matches all files that start with the letters a through n or p through z (or more simply stated, any filename that doesn't start with the lowercase letter o).If the first character following the [is a !, the sense of the match is inverted. That is, any character is matched except those enclosed in the brackets. So[!a–z]matches any character except a lowercase letter, and*[!o]matches any file that doesn't end with the lowercase letter o.Table 1.1 gives a few more examples of filename substitution.Table 1.1 Filename Substitution ExamplesTable 1.1 Filename Substitution ExamplesFilename NuancesSpaces in FilenamesA discussion of command lines and filenames wouldn't be complete without talking about the bane of old-school Unix people and very much the day-to-day reality of Linux, Windows, and Mac users: spaces in filenames.Filename NuancesSpaces in FilenamesA discussion of command lines and filenames wouldn't be complete without talking about the bane of old-school Unix people and very much the day-to-day reality of Linux, Windows, and Mac users: spaces in filenames.Spaces in FilenamesA discussion of command lines and filenames wouldn't be complete without talking about the bane of old-school Unix people and very much the day-to-day reality of Linux, Windows, and Mac users: spaces in filenames.The problem arises from the fact that the shell uses spaces as delimiters between words. In other words the phrase echo hi mom is properly parsed as an invocation to the command echo, with two arguments hi and mom.Now imagine you have a file called my test

document. How do you reference it from the command line? How do you view it or display it using the cat command?Click here to view code image$ cat my test documentcat: my: No such file or directorycat: test: No such file or directorycat: document: No such file or directoryThat definitely doesn't work. Why? Because cat wants a filename to be specified and instead of seeing one, it sees three: my, test, and document.There are two standard solutions for this: Either escape every space by using a backslash, or wrap the entire filename in quotes so that the shell understands that it's a single word with spaces, rather than multiple words.Click here to view code image$ cat "my test document"This is a test document and is fullof scintillating information to edifyand amaze.$ cat my\ test\ documentThis is a test document and is fullof scintillating information to edifyand amaze.That solves the problem and is critical to know as you proceed with file systems that quite likely have lots of directories and files that contain spaces as part of their filenames.Other Weird CharactersWhile the space might be the most difficult and annoying of special characters that can appear in filenames, occasionally you'll find others show up that can throw a proverbial monkey-wrench into your command line efforts.Other Weird CharactersWhile the space might be the most difficult and annoying of special characters that can appear in filenames, occasionally you'll find others show up that can throw a proverbial monkey-wrench into your command line efforts.For example, how would you deal with a filename that contains a question mark? In the next section, you'll learn that the character "?" has a specific meaning to the shell. Most modern shells are smart enough to sidestep the duplication of meaning, but, again, quoting the filename or using backslashes to denote that the special character is part of the filename is required:Click here to view code image$ ls -l who\ me \?-rw-r--r-- 1 taylor staff 0 Dec 4 10:18 who me?Where this really gets interesting is if you have a backslash or quote as part of the filename, something that can happen inadvertently, particularly for files created by graphically oriented programs on a Linux or Mac system. The trick? Use single quotes to escape a filename that includes a double quote, and vice versa. Like this:Click here to view code image$ ls -l "don't quote me" 'She said "yes"'-rw-r--r-- 1 taylor staff 0 Dec 4 10:18 don't quote me-rw-r--r-- 1 taylor staff 0 Dec 4 10:19 She said "yes"This topic will come up again as we proceed, but now you know how to side-step problems with directories or files that contain spaces or other non-standard characters.Standard Input/Output, and I/O RedirectionStandard Input and Standard OutputMost Unix system commands take input from your screen and send the resulting output back to your screen. In Unix nomenclature, the screen is generally called the terminal, a reference that harkens back to the earliest days of computing. Nowadays it's more likely to be a terminal program you're running within a graphical environment, whether it's a Linux window manager, a Windows computer, or a Mac system.Standard Input/Output, and I/O RedirectionStandard Input and Standard OutputMost Unix system commands take input from your screen and send the resulting output back to your screen. In Unix nomenclature, the screen is generally called the terminal, a reference that harkens back to the earliest days of computing. Nowadays it's more likely to be a terminal program you're running within a graphical environment, whether it's a Linux window manager, a

Windows computer, or a Mac system.Standard Input and Standard OutputMost Unix system commands take input from your screen and send the resulting output back to your screen. In Unix nomenclature, the screen is generally called the terminal, a reference that harkens back to the earliest days of computing. Nowadays it's more likely to be a terminal program you're running within a graphical environment, whether it's a Linux window manager, a Windows computer, or a Mac system.A command normally reads its input from standard input, which is your computer keyboard by default. It's a fancy way of clarifying that you "type in" your information. Similarly, a command normally writes its output to standard output, which is also your terminal or terminal app by default. This concept is depicted in Figure 1.9.Figure 1.9 Typical Unix commandFigure 1.9 Typical Unix commandAs an example, recall that executing the who command results in the display of all users that are currently logged-in. More formally, the who command writes a list of the logged-in users to standard output. This is depicted in Figure 1.10.Figure 1.10 who commandFigure 1.10 who commandIt turns out that just about every single Unix command can take the output of a previous command or file as its input too, and can even send its output to another command or program. This concept is hugely important to understanding the power of the command line and why it's so helpful to know all of these commands even when a graphical interface might be also available for your use.Before we get there, however, consider this: if the sort command is invoked without a filename argument, the command takes its input from standard input. As with standard output, this is your terminal (or keyboard) by default.When input is entered this way, an end-of-file sequence must be specified after the last line is typed, and, by Unix convention, that's Ctrl+d; that is, the sequence produced by simultaneously pressing the Control (or Ctrl, depending on your keyboard) key and the d key.As an example, let's use the sort command to sort the following four names: Tony, Barbara, Harry, Dirk. Instead of first entering the names into a file, we'll enter them directly from the terminal:$ sortTonyBarbaraHarryDirkCtrl +dBarbaraDirkHarryTony$Because no filename was specified to the sort command, the input was taken from standard input, the terminal. After the fourth name was typed in, the Ctrl and d keys were pressed to signal the end of the data. At that point, the sort command sorted the four names and displayed the results on the standard output device, which is also the terminal. This is depicted in Figure 1.11.Figure 1.11 sort commandFigure 1.11 sort commandThe wc command is another example of a command that takes its input from standard input if no filename is specified on the command line. The following shows an example of this command used to count the number of lines of text entered from the terminal:$ wc -lThis is text thatis typed on thestandard input device.Ctrl+d 3$Note that the Ctrl+d that is used to terminate the input is not counted as a separate line by the wc command because it's interpreted by the shell, not handed to the command. Furthermore, because the -l flag was specified to the wc command, only the count of the number of lines (3) is presented as the output of the command.Output RedirectionThe output from a command normally intended for standard output can be easily "diverted" to a file instead. This capability is known as output redirection and is also essential to understanding the power of Unix.Output RedirectionThe output from a command normally

intended for standard output can be easily “diverted” to a file instead. This capability is known as output redirection and is also essential to understanding the power of Unix.If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to the file file instead:$ who > users$This command causes the who command to be executed and its output to be written into the file users. Notice that no output appears. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. We can check this, of course:$ cat usersoko tty01 Sep 12 07:30ai tty15 Sep 12 13:32ruth tty21 Sep 12 10:10pat tty24 Sep 12 13:07steve tty25 Sep 12 13:03$If a command has its output redirected to a file and the file already contains some data, that data will be overwritten and lost.$ echo line 1 > users$ cat usersline 1$But now consider this example, remembering that users already contains the output of the earlier who command:$ echo line 2 >> users$ cat usersline 1line 2$If you’re paying close attention you’ll notice that this echo command uses a different type of output redirection, indicated by the characters >>. This character pair causes the standard output from the command to be appended to the contents of the specified file. The previous contents are not lost; the new output simply gets added to the end.By using the redirection append characters >>, you can use cat to append the contents of one file onto the end of another:Click here to view code image$ cat file1This is in file1.$ cat file2This is in file2.$ cat file1 >> file2 Append file1 to file2$ cat file2This is in file2.This is in file1.$Recall that specifying more than one filename to cat results in the display of the first file followed immediately by the second file, and so on. This means there’s a second way to accomplish the same result:Click here to view code image$ cat file1This is in file1.$ cat file2This is in file2.$ cat file1 file2This is in file1.This is in file2.$ cat file1 file2 > file3 Redirect it instead$ cat file3This is in filel.This is in file2.$In fact, that’s where the cat command gets its name: When used with more than one file, its effect is to concatenate the files together.Input RedirectionJust as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. And as the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command. Of course, only commands that normally take their input from standard input can have their input redirected from a file in this manner.
Input RedirectionJust as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. And as the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command. Of course, only commands that normally take their input from standard input can have their input redirected from a file in this manner.
To redirect input, type the < character followed by the name of the file that the input is to be read from. To count the number of lines in the file users, for example, you already know that you can execute the command wc -l users:$ wc -l users 2 users$It turns out that you can also count the number of lines in the file by redirecting standard input for the wc command:$ wc -l < users 2$Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.This points out a

subtle distinction between the execution of the two commands. In the first case, wc knows that it is reading its input from the file users. In the second case, it only sees the raw data which is being fed to it via standard input. The shell redirects the input so that it comes from the file users and not the terminal (more about this in the next chapter). As far as wc is concerned, it doesn't know whether its input is coming from the terminal or from a file, so it can't report the filename! PipesAs you will recall, the file users that was created previously contains a list of all the users currently logged in to the system. Because you know that there will be one line in the file for each user logged in to the system, you can easily determine the number of login sessions by counting the number of lines in the users file:PipesAs you will recall, the file users that was created previously contains a list of all the users currently logged in to the system. Because you know that there will be one line in the file for each user logged in to the system, you can easily determine the number of login sessions by counting the number of lines in the users file:$ who > users$ wc -l < users 5$This output indicates that currently five users are logged in or that there are five login sessions, the difference being that users, particularly administrators, often log in more than once. Now you have a command sequence you can use whenever you want to know how many users are logged in.Another approach to determine the number of logged-in users bypasses the intermediate file. As referenced earlier, Unix lets you "connect" two commands together. This connection is known as a pipe, and it enables you to take the output from one command and feed it directly into the input of another. A pipe is denoted by the character |, which is placed between the two commands. To create a pipe between the who and wc -l commands, you type who | wc -l:$ who | wc -l 5$The pipe that is created between these two commands is depicted in Figure 1.12.Figure 1.12 Pipeline process: who | wc -lFigure 1.12 Pipeline process: who | wc -lWhen a pipe is established between two commands, the standard output from the first command is connected directly to the standard input of the second command. You know that the who command writes its list of logged-in users to standard output. Furthermore, you know that if no filename argument is specified to the wc command, it takes its input from standard input. Therefore, the list of logged-in users that is output from the who command automatically becomes the input to the wc command. Note that you never see the output of the who command at the terminal because it is piped directly into the wc command. This is depicted in Figure 1.13.Figure 1.13 Pipeline processFigure 1.13 Pipeline processA pipe can be made between any two programs, provided that the first program writes its output to standard output, and the second program reads its input from standard input.As another example, suppose you wanted to count the number of files contained in your directory. Knowledge of the fact that the ls command displays one line of output per file enables you to use the same type of approach as before:$ ls | wc -l 10$The output indicates that the current directory contains 10 files.It is also possible to create a more complicated pipeline that consists of more than two programs, with the output of one program feeding into the input of the next. As you become a more sophisticated command line user, you'll find many situations where pipelines can be tremendously powerful.FiltersThe term filter is often used in Unix terminology to

refer to any program that can take input from standard input, perform some operation on that input, and write the results to standard output. More succinctly, a filter is any program that can be used to modify the output of other programs in a pipeline. So in the pipeline in the previous example, wc is considered a filter. ls is not because it does not read its input from standard input. As other examples, cat and sort are filters, whereas who, date, cd, pwd, echo, rm, mv, and cp are not.FiltersThe term filter is often used in Unix terminology to refer to any program that can take input from standard input, perform some operation on that input, and write the results to standard output. More succinctly, a filter is any program that can be used to modify the output of other programs in a pipeline. So in the pipeline in the previous example, wc is considered a filter. ls is not because it does not read its input from standard input. As other examples, cat and sort are filters, whereas who, date, cd, pwd, echo, rm, mv, and cp are not.Standard ErrorIn addition to standard input and standard output, there is a third virtual device known as standard error. This is where most Unix commands write their error messages. And as with the other two "standard" places, standard error is associated with your terminal or terminal app by default. In most cases, you never know the difference between standard output and standard error:Standard ErrorIn addition to standard input and standard output, there is a third virtual device known as standard error. This is where most Unix commands write their error messages. And as with the other two "standard" places, standard error is associated with your terminal or terminal app by default. In most cases, you never know the difference between standard output and standard error:Click here to view code image$ ls n* List all files beginning with nn* not found$Here the "not found" message is actually being written to standard error by the ls command. You can verify that this message is not being written to standard output by redirecting the ls command's output:$ ls n* > foon* not found$As you can see, the message is still printed out at the terminal and was not added to the file foo, even though you redirected standard output.The preceding example shows the raison d'être for standard error: so that error messages will still get displayed at the terminal even if standard output is redirected to a file or piped to another command.You can also redirect standard error to a file (for instance, if you're logging a program's potential errors during long-term operation) by using the slightly more complex notationcommand 2> fileNote that no space is permitted between the 2 and the >. Any error messages normally intended for standard error will be diverted into the specified file, similar to the way standard output gets redirected.$ ls n* 2> errors$ cat errorsn* not found$More on CommandsTyping More Than One Command on a LineYou can type more than one command on a line provided that you separate them with a semicolon. For example, you can find out the current time and your current working directory by typing in the date and pwd commands on the same line:More on CommandsTyping More Than One Command on a LineYou can type more than one command on a line provided that you separate them with a semicolon. For example, you can find out the current time and your current working directory by typing in the date and pwd commands on the same line:Typing More Than One Command on a LineYou can type more than one command on a line provided that you separate them with a semicolon. For example, you can find out the

current time and your current working directory by typing in the date and pwd commands on the same line:$ date; pwdSat Jul 20 14:43:25 EDT 2002/users/pat/bin$You can string out as many commands as you want on the line, as long as each command is delimited by a semicolon.Sending a Command to the BackgroundNormally, you type in a command and then wait for the results of the command to be displayed at the terminal. For all the examples you have seen thus far, this waiting time is typically short—a fraction of a second.Sending a Command to the BackgroundNormally, you type in a command and then wait for the results of the command to be displayed at the terminal. For all the examples you have seen thus far, this waiting time is typically short—a fraction of a second.Sometimes, however, you may have to run commands that require a few minutes or longer to complete. In those cases, you'll have to wait for the command to finish executing before you can proceed further, unless you execute the command in the background.It turns out that while your Unix or Linux system seems like it's focused completely on what you're doing, all systems are actually multitasking, running multiple commands simultaneously at any given time. If you're on an Ubuntu system, for example, it might have the window manager, a clock, a status monitor and your terminal window all running simultaneously. You too can run multiple commands simultaneously from the command line. That's the idea of putting a command "into background," letting you work on other tasks while it completes.The notational convention for pushing a command or command sequence into background is to append the ampersand character &. This means that the command will no longer tie up your terminal, and you can then proceed with other work. The standard output from the command will still be directed to your terminal, though in most cases the standard input will be dissociated from your terminal. If the command does try to read from standard input, it will stop and wait for you to bring it to the foreground (we'll discuss this in more detail in Chapter 14, "Interactive and Nonstandard Shell Features").Here's an example:Click here to view code image $ sort bigdata > out & Send the sort to the background[1] 1258 Process id$ date Your terminal is immediately available to do other workSat Jul 20 14:45:09 EDT 2002$When a command is sent to the background, the Unix system automatically displays two numbers. The first is called the command's job number and the second the process ID, or PID. In the preceding example, 1 is the job number and 1258 the process ID. The job number is used as a shortcut for referring to a specific background job by some shell commands. (You'll learn more about this in Chapter 14.) The process ID uniquely identifies the command that you sent to the background and can be used to obtain status information about the command. This is done with the processor status—ps—command.The ps CommandThe ps command gives you information about the processes running on the system. Without any options, it prints the status of just your processes. If you type in ps at your terminal, you'll get a few lines back describing the processes you have running:The ps CommandThe ps command gives you information about the processes running on the system. Without any options, it prints the status of just your processes. If you type in ps at your terminal, you'll get a few lines back describing the processes you have running:$ ps PID TTY TIME CMD13463 pts/16 00:00:09 bash19880 pts/16 00:00:00 ps$The ps

command (typically; your system might vary) prints out four columns of information: PID, the process ID; TTY, the terminal number that the process was run from; TIME, the amount of computer time in minutes and seconds that process has used; and CMD, the name of the process. (The bash process in the preceding example is the shell that was started when we logged in, and it's used 9 seconds of computer time.) Until the command is finished, it shows up in the output of the ps command as a running process, so process 19880 in the preceding example is the ps command itself.When used with the -f option, ps prints out more information about your processes, including the parent process ID (PPID), the time the process started (STIME), and the command arguments:Click here to view code image$ ps -fUID PID PPID C STIME TTY TIME CMDsteve 13463 13355 0 12:12 pts/16 00:00:09 bashsteve 19884 13463 0 13:39 pts/16 00:00:00 ps -f$Command SummaryTable 1.2 summarizes the commands reviewed in this chapter. In this table, file refers to a file, file(s) to one or more files, dir to a directory, and dir(s) to one or more directories.Command SummaryTable 1.2 summarizes the commands reviewed in this chapter. In this table, file refers to a file, file(s) to one or more files, dir to a directory, and dir(s) to one or more directories.Table 1.2 Command SummaryTable 1.2 Command Summary

Download to continue reading...

What people say about this book

Christopher, “A bit thin on examples. Lots of really great, well-explained content. It was sometimes dense and tended toward the theoretical side, though. I would have liked to see more working examples.”

Sapientlife, “Five Stars. Good read”

Alvaro Pino Nino, “Five Stars. Exceelent”

TheReader, “Very good and easy to follow book into Shell programming. Got that book just a few days ago but couldn't stop reading. Well written and pretty easy to follow even for non-native English speakers. All concepts are well explained and example codes are easy to grasp. Step by step the reader is guided from the simple to more sophisticated application of shell commands and little programs. This book was bought as a refresher for myself already working in IT for 20+ years. I have built-up a nice parallel cluster out of 15 Raspberry Pi B3+ computers - fully maintained via console from my iMac. So I needed a good book on shell programming to automate and get better to handle a lot of tasks in the day-to-day operation of my new little "Raspi Beast" running on Raspbian Linux. And guess what? This book is the one that helped me getting a lot better on doing some nice "console magic". Just a great book - love it and recommend it.”

andrea bardella, “Professionale. Professionale, ben scritto e sufficientemente approfondito negli aspetti che tratta. Ogni capitolo va prima letto senza soffermarsi particolarmente, per poi tornarci sopra in approfondimento.”

Paulo Diaz, “Excelente libro para aprender el Shell basados en Linux. Excelente libro para aprender el Shell basados en Linux”

Sudoer, “The best. The best”

The book by Linda Stevens has a rating of 5 out of 4.5. 35 people have provided feedback.

About This E-Book Title Page Copyright Page Developer’s Library Contents at a Glance Table of Contents About the Authors We Want to Hear from You! Reader Services Introduction 1. A Quick Review of the Basics 2. What Is the Shell? 3. Tools of the Trade 4. And Away We Go 5. Can I Quote You on That? 6. Passing Arguments 7. Decisions, Decisions 8. 'Round and 'Round She Goes 9. Reading and Printing Data 10. Your Environment 11. More on Parameters 12. Loose Ends 13. Rolo Revisited 14. Interactive and Nonstandard Shell Features A. Shell Summary B. For More Information Index Inside Front Cover Inside Back Cover Code Snippets

Book Information

Language: English
Paperback: 52 pages
Item Weight: 5.1 ounces
Dimensions: 8.5 x 0.11 x 11 inches
File size: 19665 KB
Simultaneous device usage: Up to 5 simultaneous devices, per publisher limits
Text-to-Speech: Enabled
Screen Reader: Supported
Enhanced typesetting: Enabled
X-Ray: Not Enabled
Word Wise: Not Enabled
Sticky notes: On Kindle Scribe
Print length: 1189 pages
Reading age: 8 - 10 years
Flexibound: 128 pages

DMCA