Praise for the First Edition of *A Practical Guide to Linux® Commands, Editors, and Shell Programming*

"This book is a very useful tool for anyone who wants to 'look under the hood' so to speak, and really start putting the power of Linux to work. What I find particularly frustrating about man pages is that they never include examples. Sobell, on the other hand, outlines very clearly what the command does and then gives several common, easy-to-understand examples that make it a breeze to start shell programming on one's own. As with Sobell's other works, this is simple, straight-forward, and easy to read. It's a great book and will stay on the shelf at easy arm's reach for a long time."

—*Ray Bartlett*
Travel Writer

"Overall I found this book to be quite excellent, and it has earned a spot on the very front of my bookshelf. It covers the real 'guts' of Linux—the command line and its utilities—and does so very well. Its strongest points are the outstanding use of examples, and the Command Reference section. Highly recommended for Linux users of all skill levels. Well done to Mark Sobell and Prentice Hall for this outstanding book!"

—*Dan Clough*
Electronics Engineer and
Slackware Linux user

"Totally unlike most Linux books, this book avoids discussing everything via GUI and jumps right into making the power of the command line your friend."

—*Bjorn Tipling*
Software Engineer
ask.com

"This book is the best distro-agnostic, foundational Linux reference I've ever seen, out of dozens of Linux-related books I've read. Finding this book was a real stroke of luck. If you want to really understand how to get things done at the command line, where the power and flexibility of

free UNIX-like OSes really live, this book is among the best tools you'll find toward that end."

—*Chad Perrin*
Writer, TechRepublic

Praise for Other Books by Mark G. Sobell

"I keep searching for books that collect everything you want to know about a subject in one place, and keep getting disappointed. Usually the books leave out some important topic, while others go too deep in some areas and must skim lightly over the others. *A Practical Guide to Red Hat® Linux®* is one of those rare books that actually pulls it off. Mark G. Sobell has created a single reference for Red Hat Linux that can't be beat! This marvelous text (with a 4-CD set of Linux Fedora Core 2 included) is well worth the price. This is as close to an 'everything you ever needed to know' book that I've seen. It's just that good and rates 5 out of 5."

—*Ray Lodato*
Slashdot contributor

"Mark Sobell has written a book as approachable as it is authoritative."

—*Jeffrey Bianchine*
Advocate, Author, Journalist

"Excellent reference book, well suited for the sysadmin of a Linux cluster, or the owner of a PC contemplating installing a recent stable Linux. Don't be put off by the daunting heft of the book. Sobell has strived to be as inclusive as possible, in trying to anticipate your system administration needs."

—*Wes Boudville*
Inventor

"*A Practical Guide to Red Hat® Linux®* is a brilliant book. Thank you Mark Sobell."

—*C. Pozrikidis*
University of California
at San Diego

"This book presents the best overview of the Linux operating system that I have found. . . . [It] should be very helpful and understandable no matter what the reader's background: traditional UNIX user, new Linux devotee, or even Windows user. Each topic is presented in a clear, complete fashion, and very few assumptions are made about what the reader knows. . . . The book is extremely useful as a reference, as it contains a 70-page glossary of terms and is very well indexed. It is organized in such a way that the reader can focus on simple tasks without having to wade through more advanced topics until they are ready."

—*Cam Marshall*
Marshall Information Service LLC
Member of Front Range UNIX
Users Group [FRUUG]
Boulder, Colorado

"Conclusively, this is THE book to get if you are a new Linux user and you just got into the RH/Fedora world. There's no other book that discusses so many different topics and in such depth."

—*Eugenia Loli-Queru*
Editor in Chief
OSNews.com

Blank

Excerpts of Chapters From

A Practical Guide to Linux® Commands, Editors, and Shell Programming

SECOND EDITION

Mark G. Sobell

ISBN: 978-0-13-136736-4

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Blank

5

The Shell

This chapter takes a close look at the shell and explains how to use some of its features. It discusses command-line syntax and describes how the shell processes a command line and initiates execution of a program. In addition the chapter explains how to redirect input to and output from a command, construct pipes and filters on the command line, and run a command in the background. The final section covers filename expansion and explains how you can use this feature in your everyday work.

Except as noted, everything in this chapter applies to the Bourne Again (bash) and TC (tcsh) Shells. The exact wording of the shell output differs from shell to shell: What the shell you are using displays may differ slightly from what appears in this book. For shell-specific information, refer to Chapters 8 (bash) and 9 (tcsh). Chapter 10 covers writing and executing bash shell scripts.

In This Chapter

REDIRECTION

The term *redirection* encompasses the various ways you can cause the shell to alter where standard input of a command comes from and where standard output goes to. By default the shell associates standard input and standard output of a command with the keyboard and the screen. You can cause the shell to redirect standard input or standard output of any command by associating the input or output with a command or file other than the device file representing the keyboard and the screen. This section demonstrates how to redirect input from and output to ordinary files.

REDIRECTING STANDARD OUTPUT

The *redirect output symbol* (>) instructs the shell to redirect the output of a command to the specified file instead of to the screen (Figure 5-6). The format of a command line that redirects output is

command [arguments] > filename

where ***command*** is any executable program (such as an application program or a utility), ***arguments*** are optional arguments, and ***filename*** is the name of the ordinary file the shell redirects the output to.

Figure 5-7 uses cat to demonstrate output redirection. This figure contrasts with Figure 5-5, where standard input *and* standard output are associated with the keyboard and screen. The input in Figure 5-7 comes from the keyboard. The redirect output symbol on the command line causes the shell to associate cat's standard output with the **sample.txt** file specified on the command line.

After giving the command and typing the text shown in Figure 5-7, the **sample.txt** file contains the text you entered. You can use cat with an argument of **sample.txt** to display this file. The next section shows another way to use cat to display the file.

Figure 5-7 shows that redirecting standard output from cat is a handy way to create a file without using an editor. The drawback is that once you enter a line

```
$ cat > sample.txt
This text is being entered at the keyboard and
cat is copying it to a file.
Press CONTROL-D to signal the end of file.
CONTROL-D
$
```

Figure 5-7 cat with its output redirected

Redirecting output can destroy a file I

caution Use caution when you redirect output to a file. If the file exists, the shell will overwrite it and destroy its contents. For more information see the tip "Redirecting output can destroy a file II" on page 129.

and press RETURN, you cannot edit the text. While you are entering a line, the erase and kill keys work to delete text. This procedure is useful for creating short, simple files.

Figure 5-8 shows how to use cat and the redirect output symbol to *catenate* (join one after the other—the derivation of the name of the cat utility) several files into one larger file. The first three commands display the contents of three files: **stationery**, **tape**, and **pens**. The next command shows cat with three filenames as arguments. When you call it with more than one filename, cat copies the files, one at a time, to standard output. This command redirects standard output to the file **supply_orders**. The final cat command shows that **supply_orders** holds the contents of all three original files.

```
$ cat stationery
2,000 sheets letterhead ordered:   10/7/08

$ cat tape
1 box masking tape ordered:        10/14/08
5 boxes filament tape ordered:     10/28/08

$ cat pens
12 doz. black pens ordered:        10/4/08

$ cat stationery tape pens > supply_orders

$ cat supply_orders
2,000 sheets letterhead ordered:   10/7/08
1 box masking tape ordered:        10/14/08
5 boxes filament tape ordered:     10/28/08
12 doz. black pens ordered:        10/4/08
$
```

Figure 5-8 Using cat to catenate files

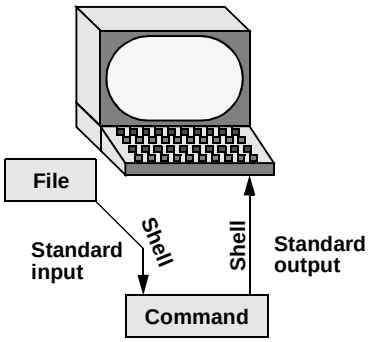

Figure 5-9 Redirecting standard input

REDIRECTING STANDARD INPUT

Just as you can redirect standard output, so you can redirect standard input. The *redirect input symbol* (<) instructs the shell to redirect a command's input to come from the specified file instead of from the keyboard (Figure 5-9). The format of a command line that redirects input is

command [arguments] < filename

where ***command*** is any executable program (such as an application program or a utility), ***arguments*** are optional arguments, and ***filename*** is the name of the ordinary file the shell redirects the input from.

Figure 5-10 shows cat with its input redirected from the **supply_orders** file created in Figure 5-8 and standard output going to the screen. This setup causes cat to display the sample file on the screen. The system automatically supplies an EOF signal at the end of an ordinary file.

Utilities that take input from a file or standard input

Giving a cat command with input redirected from a file yields the same result as giving a cat command with the filename as an argument. The cat utility is a member of a class of Linux utilities that function in this manner. Other members of this class of utilities include lpr, sort, grep, and Perl. These utilities first examine the command line you call them with. If you include a filename on the command line, the utility takes its input from the file you specify. If you do not specify a filename, the utility takes its input from standard input. It is the utility or program—not the shell or operating system—that functions in this manner.

```
$ cat < supply_orders
2,000 sheets letterhead ordered:   10/7/08
1 box masking tape ordered:        10/14/08
5 boxes filament tape ordered:     10/28/08
12 doz. black pens ordered:        10/4/08
```

Figure 5-10 cat with its input redirected

noclobber: AVOIDS OVERWRITING FILES

The shell provides the **noclobber** feature that prevents overwriting a file using redirection. Under bash you can enable this feature by setting **noclobber** using the command **set –o noclobber**. The same command with **+o** unsets **noclobber**. Under tcsh use **set noclobber** and **unset noclobber**. With **noclobber** set, if you redirect output to an existing file, the shell displays an error message and does not execute the command. The following example, run under bash and tcsh, creates a file using touch, sets **noclobber**, attempts to redirect the output from echo to the newly created file, unsets **noclobber**, and performs the redirection again:

bash
```
$ touch tmp
$ set -o noclobber
$ echo "hi there" > tmp
bash: tmp: cannot overwrite existing file
$ set +o noclobber
$ echo "hi there" > tmp
$
```

tcsh
```
tcsh $ touch tmp
tcsh $ set noclobber
tcsh $ echo "hi there" > tmp
tmp: File exists.
tcsh $ unset noclobber
tcsh $ echo "hi there" > tmp
$
```

Redirecting output can destroy a file II

caution Depending on which shell you are using and how the environment is set up, a command such as the following may yield undesired results:

```
$ cat orange pear > orange
cat: orange: input file is output file
```

Although cat displays an error message, the shell destroys the contents of the existing **orange** file. The new **orange** file will have the same contents as **pear** because the first action the shell takes when it sees the redirection symbol (>) is to remove the contents of the original **orange** file. If you want to catenate two files into one, use cat to put the two files into a temporary file and then use mv to rename the temporary file:

```
$ cat orange pear > temp
$ mv temp orange
```

What happens in the next example can be even worse. The user giving the command wants to search through files **a**, **b**, and **c** for the word **apple** and redirect the output from grep (page 52) to the file **a.output**. Unfortunately the user enters the filename as **a output**, omitting the period and inserting a SPACE in its place:

```
$ grep apple a b c > a output
grep: output: No such file or directory
```

The shell obediently removes the contents of **a** and then calls grep. The error message may take a moment to appear, giving you a sense the command is running correctly. Even after you see the error message, it may take a while to realize you have destroyed the contents of **a**.

You can override **noclobber** by putting a pipe symbol (tcsh uses an exclamation point) after the redirect symbol (>|). In the following example, the user creates a file by redirecting the output of date. Next the user sets the **noclobber** variable and redirects output to the same file again. The shell displays an error message. Then the user places a pipe symbol after the redirect symbol and the shell allows the user to overwrite the file.

```
$ date > tmp2
$ set -o noclobber
$ date > tmp2
bash: a: cannot overwrite existing file
$ date >| tmp2
$
```

For more information on using **noclobber** under tcsh, refer to page 377.

APPENDING STANDARD OUTPUT TO A FILE

The *append output symbol* (>>) causes the shell to add new information to the end of a file, leaving existing information intact. This symbol provides a convenient way of catenating two files into one. The following commands demonstrate the action of the append output symbol. The second command accomplishes the catenation described in the preceding caution box:

```
$ cat orange
this is orange
$ cat pear >> orange
$ cat orange
this is orange
this is pear
```

The first command displays the contents of the **orange** file. The second command appends the contents of the **pear** file to the **orange** file. The final cat displays the result.

Do not trust noclobber

caution Appending output is simpler than the two-step procedure described in the preceding caution box but you must be careful to include both greater than signs. If you accidentally use only one and the **noclobber** feature is not set, the shell will overwrite the **orange** file. Even if you have the **noclobber** feature turned on, it is a good idea to keep backup copies of the files you are manipulating in case you make a mistake.

Although it protects you from overwriting a file using redirection, **noclobber** does not stop you from overwriting a file using cp or mv. These utilities include the **–i** (interactive) option that helps protect you from this type of mistake by verifying your intentions when you try to overwrite a file. For more information see the tip "cp can destroy a file" on page 50.

The next example shows how to create a file that contains the date and time (the output from date), followed by a list of who is logged in (the output from who). The first line in Figure 5-11 redirects the output from date to the file named **whoson**. Then cat displays the file. Next the example appends the output from who to the **whoson** file. Finally cat displays the file containing the output of both utilities.

```
$ date > whoson
$ cat whoson
Fri Mar 27 14:31:18 PST 2009
$ who >> whoson
$ cat whoson
Fri Mar 27 14:31:18 PST 2009
sam        console      Mar 27 05:00(:0)
max        pts/4        Mar 27 12:23(:0.0)
max        pts/5        Mar 27 12:33(:0.0)
zach       pts/7        Mar 26 08:45 (bravo.example.com)
```

Figure 5-11 Redirecting and appending output

/dev/null: MAKING DATA DISAPPEAR

The **/dev/null** device is a *data sink*, commonly referred to as a *bit bucket*. You can redirect output that you do not want to keep or see to **/dev/null** and the output will disappear without a trace:

```
$ echo "hi there" > /dev/null
$
```

When you read from **/dev/null**, you get a null string. Give the following cat command to truncate a file named **messages** to zero length while preserving the ownership and permissions of the file:

```
$ ls -l messages
-rw-r--r--   1 max pubs 25315 Oct 24 10:55 messages
$ cat /dev/null > messages
$ ls -l messages
-rw-r--r--   1 max pubs 0 Oct 24 11:02 messages
```

Blank

6

The vim Editor

This chapter begins with a history and description of vi, the original, powerful, sometimes cryptic, interactive, visually oriented text editor. The chapter continues with a tutorial that explains how to use vim (vi improved—a vi clone supplied with or available for most Linux distributions) to create and edit a file. Much of the tutorial and the balance of the chapter apply to vi and other vi clones. Following the tutorial, the chapter delves into the details of many vim commands and explains how to use parameters to customize vim to meet your needs. It concludes with a quick reference/summary of vim commands.

In This Chapter

SEARCHING AND SUBSTITUTING

Searching for and replacing a character, a string of text, or a string that is matched by a regular expression is a key feature of any editor. The vim editor provides simple commands for searching for a character on the current line. It also provides more complex commands for searching for—and optionally substituting for—single and multiple occurrences of strings or regular expressions anywhere in the Work buffer.

SEARCHING FOR A CHARACTER

Find (**f**/**F**) You can search for and move the cursor to the next occurrence of a specified character on the current line using the **f** (Find) command. Refer to "Moving the Cursor to a Specific Character" on page 165.

Find (**t**/**T**) The next two commands are used in the same manner as the Find commands. The **t** command places the cursor on the character before the next occurrence of the specified character. The **T** command places the cursor on the character after the previous occurrence of the specified character.

A semicolon (**;**) repeats the last **f**, **F**, **t**, or **T** command.

You can combine these search commands with other commands. For example, the command **d2fq** deletes the text from the current character to the second occurrence of the letter **q** on the current line.

Command	Result
s	Substitutes one or more characters for current character
S	Substitutes one or more characters for current line
5s	Substitutes one or more characters for five characters, starting with current character

SEARCHING FOR A STRING

Search (/ ?) The vim editor can search backward or forward through the Work buffer to find a string of text or a string that matches a regular expression (Appendix A). To find the next occurrence of a string (forward), press the forward slash (/) key, enter the text you want to find (called the *search string*), and press RETURN. When you press the slash key, vim displays a slash on the status line. As you enter the string of text, it is also displayed on the status line. When you press RETURN, vim searches for the string. If this search is successful, vim positions the cursor on the first character of the string. If you use a question mark (?) in place of the forward slash, vim searches for the previous occurrence of the string. If you need to include a forward slash in a forward search or a question mark in a backward search, you must quote it by preceding it with a backslash (\).

Two distinct ways of quoting characters

tip You use CONTROL-V to quote special characters in text that you are entering into a file (page 169). This section discusses the use of a backslash (\) to quote special characters in a search string. The two techniques of quoting characters are not interchangeable.

Next (n/N) The **N** and **n** keys repeat the last search but do not require you to reenter the search string. The **n** key repeats the original search exactly, and the **N** key repeats the search in the opposite direction of the original search.

If you are searching forward and vim does not find the search string before it gets to the end of the Work buffer, the editor typically *wraps around* and continues the search at the beginning of the Work buffer. During a backward search, vim wraps around from the beginning of the Work buffer to the end. Also, vim normally performs case-sensitive searches. Refer to "Wrap scan" (page 189) and "Ignore case in searches" (page 187) for information about how to change these search parameters.

NORMAL VERSUS INCREMENTAL SEARCHES

When vim performs a normal search (its default behavior), you enter a slash or question mark followed by the search string and press RETURN. The vim editor then moves the cursor to the next or previous occurrence of the string you are searching for.

When vim performs an incremental search, you enter a slash or question mark. As you enter each character of the search string, vim moves the highlight to the next or previous occurrence of the string you have entered so far. When the highlight is on the string you are searching for, you must press RETURN to move the cursor to the highlighted string. If the string you enter does not match any text, vim does not highlight anything.

The type of search that vim performs depends on the **incsearch** parameter (page 187). Give the command **:set incsearch** to turn on incremental searching; use **noincsearch** to turn it off. When you set the **compatible** parameter (page 158), vim turns off incremental searching.

SPECIAL CHARACTERS IN SEARCH STRINGS

Because the search string is a regular expression, some characters take on a special meaning within the search string. The following paragraphs list some of these characters. See also "Extended Regular Expressions" on page 893.

The first two items in the following list (**^** and **$**) always have their special meanings within a search string unless you quote them by preceding them with a backslash (****). You can turn off the special meanings within a search string for the rest of the items in the list by setting the **nomagic** parameter. For more information refer to "Allow special characters in searches" on page 186.

^ BEGINNING-OF-LINE INDICATOR

When the first character in a search string is a caret (also called a circumflex), it matches the beginning of a line. For example, the command **/^the** finds the next line that begins with the string **the**.

$ END-OF-LINE INDICATOR

A dollar sign matches the end of a line. For example, the command **/!$** finds the next line that ends with an exclamation point and **/ $** matches the next line that ends with a SPACE.

. ANY-CHARACTER INDICATOR

A period matches *any* character, anywhere in the search string. For example, the command **/l..e** finds **line**, **followed**, **like**, **included**, **all memory**, or any other word or character string that contains an **l** followed by any two characters and an **e**. To search for a period, use a backslash to quote the period (**\.**).

\> END-OF-WORD INDICATOR

This pair of characters matches the end of a word. For example, the command **/s\>** finds the next word that ends with an **s**. Whereas a backslash (****) is typically used to *turn off* the special meaning of a character, the character sequence **\>** has a special meaning, while **>** alone does not.

\< BEGINNING-OF-WORD INDICATOR

This pair of characters matches the beginning of a word. For example, the command **/\<The** finds the next word that begins with the string **The**. The beginning-of-word indicator uses the backslash in the same, atypical way as the end-of-word indicator.

* ZERO OR MORE OCCURRENCES

This character is a modifier that will match zero or more occurrences of the character immediately preceding it. For example, the command **/dis*m** will match the string **di** followed by zero or more **s** characters followed by an **m**. Examples of successful matches would include **dim**, or **dism**, and **dissm**.

[ ] CHARACTER-CLASS DEFINITION

Brackets surrounding two or more characters match any *single* character located between the brackets. For example, the command **/dis[ck]** finds the next occurrence of *either* **disk** or **disc**.

There are two special characters you can use within a character-class definition. A caret (**^**) as the first character following the left bracket defines the character class to be *any except the following characters*. A hyphen between two characters indicates a range of characters. Refer to the examples in Table 6-4.

Table 6-4 Search examples

Search string	What it finds
/and	Finds the next occurrence of the string **and** **Examples: sand and standard slander andiron**
/\<and\>	Finds the next occurrence of the word **and** **Example: and**
/^The	Finds the next line that starts with **The** **Examples:** **The . . .** **There . . .**
/^[0-9][0-9])	Finds the next line that starts with a two-digit number followed by a right parenthesis **Examples:** **77)…** **01)…** **15)…**
/\<[adr]	Finds the next word that starts with **a**, **d**, or **r** **Examples: apple drive road argument right**
/^[A-Za-z]	Finds the next line that starts with an uppercase or lowercase letter **Examples:** **will not find a line starting with the number 7 . . .** **Dear Mr. Jones . . .** **in the middle of a sentence like this . . .**

SUBSTITUTING ONE STRING FOR ANOTHER

A Substitute command combines the effects of a Search command and a Change command. That is, it searches for a string (regular expression) just as the **/** command

does, allowing the same special characters discussed in the previous section. When it finds the string or matches the regular expression, the Substitute command changes the string or regular expression it matches. The syntax of the Substitute command is

:[g][address]s/search-string/replacement-string[/option]

As with all commands that begin with a colon, vim executes a Substitute command from the status line.

The Substitute Address

If you do not specify an ***address***, Substitute searches only the current line. If you use a single line number as the ***address***, Substitute searches that line. If the ***address*** is two line numbers separated by a comma, Substitute searches those lines and the lines between them. Refer to "Line numbers" on page 187 if you want vim to display line numbers. Wherever a line number is allowed in the ***address***, you may also use an ***address*** string enclosed between slashes. The vim editor operates on the next line that the ***address*** string matches. When you precede the first slash of the ***address*** string with the letter **g** (for global), vim operates on all lines in the file that the ***address*** string matches. (This **g** is not the same as the one that goes at the end of the Substitute command to cause multiple replacements on a single line; see "Searching for and Replacing Strings" on the next page).

Within the ***address***, a period represents the current line, a dollar sign represents the last line in the Work buffer, and a percent sign represents the entire Work buffer. You can perform ***address*** arithmetic using plus and minus signs. Table 6-5 shows some examples of ***address***es.

Table 6-5 Addresses

Address	Portion of Work buffer addressed
5	Line 5
77,100	Lines 77 through 100 inclusive
1,.	Beginning of Work buffer through current line
.,$	Current line through end of Work buffer
1,$	Entire Work buffer
%	Entire Work buffer
/pine/	The next line containing the word **pine**
g/pine/	All lines containing the word **pine**
.,.+10	Current line through tenth following line (11 lines in all)

SEARCHING FOR AND REPLACING STRINGS

An **s** comes after the ***address*** in the command syntax, indicating that this is a Substitute command. A delimiter follows the **s**, marking the beginning of the ***search-string***. Although the examples in this book use a forward slash, you can use as a delimiter any character that is not a letter, number, blank, or backslash. You must use the same delimiter at the end of the ***search-string***.

Next comes the ***search-string***. It has the same format as the search string in the **/** command and can include the same special characters (page 175). (The ***search-string*** is a regular expression; refer to Appendix A for more information.) Another delimiter marks the end of the ***search-string*** and the beginning of the ***replacement-string***.

The ***replacement-string*** replaces the text matched by the ***search-string*** and is typically followed by the delimiter character. You can omit the final delimiter when no option follows the ***replacement-string;*** a final delimiter is required if an option is present.

Several characters have special meanings in the ***search-string***, and other characters have special meanings in the ***replacement-string***. For example, an ampersand (**&**) in the ***replacement-string*** represents the text that was matched by the ***search-string***. A backslash in the ***replacement-string*** quotes the character that follows it. Refer to Table 6-6 and Appendix A.

Table 6-6 Search and replace examples

Command	Result
:s/bigger/biggest/	Replaces the first occurrence of the string **bigger** on the current line with **biggest** **Example:** **bigger ⟶ biggest**
:1,.s/Ch 1/Ch 2/g	Replaces every occurrence of the string **Ch 1**, before or on the current line, with the string **Ch 2** **Examples:** **Ch 1 ⟶ Ch 2** **Ch 12 ⟶ Ch 22**
:1,$s/ten/10/g	Replaces every occurrence of the string **ten** with the string **10** **Examples:** **ten ⟶ 10** **often ⟶ of10** **tenant ⟶ 10ant**
:g/chapter/s/ten/10/	Replaces the first occurrence of the string **ten** with the string **10** on all lines containing the word **chapter** **Examples:** **chapter ten ⟶ chapter 10** **chapters will often ⟶ chapters will of10**

Table 6-6 Search and replace examples (continued)

Command	Result
:%s/\<ten\>/10/g	Replaces every occurrence of the word **ten** with the string **10** **Example:** **ten → 10**
:.,.+10s/every/each/g	Replaces every occurrence of the string **every** with the string **each** on the current line through the tenth following line **Examples:** **every → each** **everything → eachthing**
:s/\<short\>/"&"/	Replaces the word **short** on the current line with **"short"** (enclosed within quotation marks) **Example:** **the shortest of the short → the shortest of the "short"**

Normally, the Substitute command replaces only the first occurrence of any text that matches the ***search-string*** on a line. If you want a global substitution—that is, if you want to replace all matching occurrences of text on a line—append the **g** (global) option after the delimiter that ends the ***replacement-string***. Another useful option, **c** (check), causes vim to ask whether you would like to make the change each time it finds text that matches the ***search-string***. Pressing **y** replaces the ***search-string***, **q** terminates the command, **l** (last) makes the replacement and quits, **a** (all) makes all remaining replacements, and **n** continues the search without making that replacement.

The ***address*** string need not be the same as the ***search-string***. For example,

```
:/candle/s/wick/flame/
```

substitutes **flame** for the first occurrence of **wick** on the next line that contains the string **candle**. Similarly,

```
:g/candle/s/wick/flame/
```

performs the same substitution for the first occurrence of **wick** on each line of the file containing the string **candle** and

```
:g/candle/s/wick/flame/g
```

performs the same substitution for all occurrences of **wick** on each line that contains the string **candle**.

If the ***search-string*** is the same as the ***address,*** you can leave the ***search-string*** blank. For example, the command **:/candle/s//lamp/** is equivalent to the command **:/candle/s/candle/lamp/**.

MISCELLANEOUS COMMANDS

This section describes three commands that do not fit naturally into any other groups.

JOIN

Join (**J**)

The **J** (Join) command joins the line below the current line to the end of the current line, inserting a SPACE between what was previously two lines and leaving the cursor on this SPACE. If the current line ends with a period, vim inserts two SPACEs.

You can always "unjoin" (break) a line into two lines by replacing the SPACE or SPACEs where you want to break the line with a RETURN.

STATUS

Status (CONTROL-G)

The Status command, CONTROL-G, displays the name of the file you are editing, information about whether the file has been modified or is a readonly file, the number of the current line, the total number of lines in the Work buffer, and the percentage of the Work buffer preceding the current line. You can also use **:f** to display status information. Following is a sample status line:

```
"/usr/share/dict/words" [readonly] line 28501 of 98569 --28%-- col 1
```

. (PERIOD)

.

The **.** (period) command repeats the most recent command that made a change. If you had just given a **d2w** command (delete the next two words), for example, the **.** command would delete the next two words. If you had just inserted text, the **.** command would repeat the insertion of the same text. This command is useful if you want to change some occurrences of a word or phrase in the Work buffer. Search for the first occurrence of the word (use **/**) and then make the change you want (use **cw**). You can then use **n** to search for the next occurrence of the word and **.** to make the same change to it. If you do not want to make the change, give the **n** command again to find the next occurrence.

Blank

8

The Bourne Again Shell

In This Chapter

This chapter picks up where Chapter 5 left off by focusing on the Bourne Again Shell (bash). It notes where tcsh implementation of a feature differs from that of bash and, if appropriate, directs you to the page that discusses the alternative implementation. Chapter 10 expands on this chapter, exploring control flow commands and more advanced aspects of programming the Bourne Again Shell (bash). The bash home page is www.gnu.org/software/bash. The bash info page is a complete Bourne Again Shell reference.

The Bourne Again Shell and TC Shell (tcsh) are command interpreters and high-level programming languages. As command interpreters, they process commands you enter on the command line in response to a prompt. When you use the shell as a programming language, it processes commands stored in files called *shell scripts*. Like other languages, shells have variables and control flow commands (for example, **for** loops and **if** statements).

When you use a shell as a command interpreter, you can customize the environment you work in. You can make your prompt display the name of the working directory, create a function or an alias for cp that keeps it from overwriting certain kinds of files, take advantage of keyword variables to change

aspects of how the shell works, and so on. You can also write shell scripts that do your bidding—anything from a one-line script that stores a long, complex command to a longer script that runs a set of reports, prints them, and mails you a reminder when the job is done. More complex shell scripts are themselves programs; they do not just run other programs. Chapter 10 has some examples of these types of scripts.

Most system shell scripts are written to run under bash (or dash; see below). If you will ever work in single-user or recovery mode—when you boot the system or perform system maintenance, administration, or repair work, for example—it is a good idea to become familiar with this shell.

This chapter expands on the interactive features of the shell described in Chapter 5, explains how to create and run simple shell scripts, discusses job control, introduces the basic aspects of shell programming, talks about history and aliases, and describes command-line expansion. Chapter 9 covers interactive use of the TC Shell and TC Shell programming, and Chapter 10 presents some more challenging shell programming problems.

Blank

USER-CREATED VARIABLES

The first line in the following example declares the variable named **person** and initializes it with the value **max** (use **set person = max** in tcsh):

```
$ person=max
$ echo person
person
$ echo $person
max
```

Parameter substitution

Because the echo builtin copies its arguments to standard output, you can use it to display the values of variables. The second line of the preceding example shows that **person** does not represent **max**. Instead, the string **person** is echoed as **person**. The shell substitutes the value of a variable only when you precede the name of the variable with a dollar sign (**$**). Thus the command **echo $person** displays the value of the variable **person**; it does not display **$person** because the shell does not pass **$person** to echo as an argument. Because of the leading **$**, the shell recognizes that **$person** is the name of a variable, *substitutes* the value of the variable, and passes that value to echo. The echo builtin displays the value of the variable—not its name—never "knowing" that you called it with a variable.

Quoting the $

You can prevent the shell from substituting the value of a variable by quoting the leading **$**. Double quotation marks do not prevent the substitution; single quotation marks or a backslash (\) do.

```
$ echo $person
max
$ echo "$person"
max
$ echo '$person'
$person
$ echo \$person
$person
```

SPACEs Because they do not prevent variable substitution but do turn off the special meanings of most other characters, double quotation marks are useful when you assign values to variables and when you use those values. To assign a value that contains SPACEs or TABs to a variable, use double quotation marks around the value. Although double quotation marks are not required in all cases, using them is a good habit.

```
$ person="max and zach"
$ echo $person
max and zach
$ person=max and zach
bash: and: command not found
```

When you reference a variable whose value contains TABs or multiple adjacent SPACEs, you need to use quotation marks to preserve the spacing. If you do not quote the variable, the shell collapses each string of blank characters into a single SPACE before passing the variable to the utility:

```
$ person="max   and   zach"
$ echo $person
max and zach
$ echo "$person"
max   and   zach
```

Pathname expansion in assignments When you execute a command with a variable as an argument, the shell replaces the name of the variable with the value of the variable and passes that value to the program being executed. If the value of the variable contains a special character, such as * or ?, the shell *may* expand that variable.

The first line in the following sequence of commands assigns the string **max*** to the variable **memo**. The Bourne Again Shell does *not expand the string* because bash does not perform pathname expansion (page 136) when it assigns a value to a variable. All shells process a command line in a specific order. Within this order bash (but not tcsh) expands variables before it interprets commands. In the following echo command line, the double quotation marks quote the asterisk (*) in the expanded value of **$memo** and prevent bash from performing pathname expansion on the expanded **memo** variable before passing its value to the echo command:

```
$ memo=max*
$ echo "$memo"
max*
```

All shells interpret special characters as special when you reference a variable that contains an unquoted special character. In the following example, the shell expands the value of the **memo** variable because it is not quoted:

```
$ ls
max.report
max.summary
$ echo $memo
max.report max.summary
```

Here the shell expands the **$memo** variable to **max***, expands **max*** to **max.report** and **max.summary**, and passes these two values to echo.

optional

Braces The *$VARIABLE* syntax is a special case of the more general syntax *${VARIABLE}*, in which the variable name is enclosed by **${}**. The braces insulate the variable name from adjacent characters. Braces are necessary when catenating a variable value with a string:

```
$ PREF=counter
$ WAY=$PREFclockwise
$ FAKE=$PREFfeit
$ echo $WAY $FAKE

$
```

The preceding example does not work as planned. Only a blank line is output because, although the symbols **PREFclockwise** and **PREFfeit** are valid variable names, they are not set. By default the shell evaluates an unset variable as an empty (null) string and displays this value (bash) or generates an error message (tcsh). To achieve the intent of these statements, refer to the **PREF** variable using braces:

```
$ PREF=counter
$ WAY=${PREF}clockwise
$ FAKE=${PREF}feit
$ echo $WAY $FAKE
counterclockwise counterfeit
```

The Bourne Again Shell refers to the arguments on its command line by position, using the special variables **$1**, **$2**, **$3**, and so forth up to **$9**. If you wish to refer to arguments past the ninth argument, you must use braces: **${10}**. The name of the command is held in **$0** (page 441).

unset: REMOVES A VARIABLE

Unless you remove a variable, it exists as long as the shell in which it was created exists. To remove the *value* of a variable but not the variable itself, assign a null value to the variable (use **set person =** in tcsh):

```
$ person=
$ echo $person

$
```

You can remove a variable using the unset builtin. The following command removes the variable **person**:

```
$ unset person
```

VARIABLE ATTRIBUTES

This section discusses attributes and explains how to assign them to variables.

readonly: MAKES THE VALUE OF A VARIABLE PERMANENT

You can use the readonly builtin (not in tcsh) to ensure that the value of a variable cannot be changed. The next example declares the variable **person** to be readonly. You must assign a value to a variable *before* you declare it to be readonly; you cannot change its value after the declaration. When you attempt to unset or change the value of a readonly variable, the shell displays an error message:

```
$ person=zach
$ echo $person
zach
$ readonly person
$ person=helen
bash: person: readonly variable
```

If you use the readonly builtin without an argument, it displays a list of all readonly shell variables. This list includes keyword variables that are automatically set as readonly as well as keyword or user-created variables that you have declared as readonly. See page 296 for an example (**readonly** and **declare –r** produce the same output).

declare AND typeset: ASSIGN ATTRIBUTES TO VARIABLES

The declare and typeset builtins (two names for the same command, neither of which is available in tcsh) set attributes and values for shell variables. Table 8-3 lists five of these attributes.

Table 8-3 Variable attributes (typeset or declare)

Attribute	Meaning
–a	Declares a variable as an array (page 434)
–f	Declares a variable to be a function name (page 327)
–i	Declares a variable to be of type integer (page 296)
–r	Makes a variable readonly; also readonly (page 295)
–x	Exports a variable (makes it global); also export (page 436)

The following commands declare several variables and set some attributes. The first line declares **person1** and assigns it a value of **max**. This command has the same effect with or without the word **declare**.

```
$ declare person1=max
$ declare -r person2=zach
$ declare -rx person3=helen
$ declare -x person4
```

The readonly and export builtins are synonyms for the commands **declare –r** and **declare –x**, respectively. You can declare a variable without assigning a value to it, as the preceding declaration of the variable **person4** illustrates. This declaration makes **person4** available to all subshells (i.e., makes it global). Until an assignment is made to the variable, it has a null value.

You can list the options to declare separately in any order. The following is equivalent to the preceding declaration of **person3**:

```
$ declare -x -r person3=helen
```

Use the **+** character in place of **–** when you want to remove an attribute from a variable. You cannot remove the readonly attribute. After the following command is given, the variable **person3** is no longer exported but it is still readonly.

```
$ declare +x person3
```

You can use typeset instead of declare.

Listing variable attributes

Without any arguments or options, declare lists all shell variables. The same list is output when you run set (page 442) without any arguments.

If you use a declare builtin with options but no variable names as arguments, the command lists all shell variables that have the indicated attributes set. For example, the command **declare –r** displays a list of all readonly shell variables. This list is the same as that produced by the **readonly** command without any arguments. After the declarations in the preceding example have been given, the results are as follows:

```
$ declare -r
declare -ar BASH_VERSINFO='([0]="3" [1]="2" [2]="39" [3]="1" ... )'
declare -ir EUID="500"
declare -ir PPID="936"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:..."
declare -ir UID="500"
declare -r person2="zach"
declare -rx person3="helen"
```

The first five entries are keyword variables that are automatically declared as readonly. Some of these variables are stored as integers (**–i**). The **–a** option indicates that **BASH_VERSINFO** is an array variable; the value of each element of the array is listed to the right of an equal sign.

Integer

By default the values of variables are stored as strings. When you perform arithmetic on a string variable, the shell converts the variable into a number, manipulates it, and then converts it back to a string. A variable with the integer attribute is stored as an integer. Assign the integer attribute as follows:

```
$ declare -i COUNT
```

12

The AWK Pattern Processing Language

In This Chapter

AWK is a pattern-scanning and processing language that searches one or more files for records (usually lines) that match specified patterns. It processes lines by performing actions, such as writing the record to standard output or incrementing a counter, each time it finds a match. Unlike *procedural* languages, AWK is *data driven*: You describe the data you want to work with and tell AWK what to do with the data once it finds it.

You can use AWK to generate reports or filter text. It works equally well with numbers and text; when you mix the two, AWK usually comes up with the right answer. The authors of AWK (Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan) designed the language to be easy to use. To achieve this end they sacrificed execution speed in the original implementation.

ASSOCIATIVE ARRAYS

The *associative array* is one of gawk's most powerful features. These arrays use strings as indexes. Using an associative array, you can mimic a traditional array by using numeric strings as indexes. In Perl, an associative array is called a *hash* (page 500).

You assign a value to an element of an associative array using the following syntax:

array[string] = value

where ***array*** is the name of the array, ***string*** is the index of the element of the array you are assigning a value to, and ***value*** is the value you are assigning to that element.

Using the following syntax, you can use a **for** structure with an associative array:

*for (**elem** in **array**) **action***

where ***elem*** is a variable that takes on the value of each element of the array as the **for** structure loops through them, ***array*** is the name of the array, and ***action*** is the action that gawk takes for each element in the array. You can use the ***elem*** variable in this ***action***.

See page 551 for example programs that use associative arrays.

printf

You can use the **printf** command in place of **print** to control the format of the output gawk generates. The gawk version of **printf** is similar to that found in the C language. A **printf** command has the following syntax:

*printf "**control-string**", **arg1**, **arg2**, ..., **argn***

The ***control-string*** determines how **printf** formats ***arg1, arg2, ..., argn***. These arguments can be variables or other expressions. Within the ***control-string*** you can use **\n** to indicate a NEWLINE and **\t** to indicate a TAB. The ***control-string*** contains conversion specifications, one for each argument. A conversion specification has the following syntax:

*%[–][x[.y]]**conv***

where **–** causes **printf** to left-justify the argument, ***x*** is the minimum field width, and ***.y*** is the number of places to the right of a decimal point in a number. The ***conv*** indicates the type of numeric conversion and can be selected from the letters in Table 12-5. See page 548 for example programs that use **printf**.

Table 12-5 Numeric conversion

conv	**Type of conversion**
d	Decimal
e	Exponential notation
f	Floating-point number

Table 12-5 Numeric conversion (continued)

conv	Type of conversion
g	Use **f** or **e**, whichever is shorter
o	Unsigned octal
s	String of characters
x	Unsigned hexadecimal

CONTROL STRUCTURES

Control (flow) statements alter the order of execution of commands within a gawk program. This section details the **if...else**, **while**, and **for** control structures. In addition, the **break** and **continue** statements work in conjunction with the control structures to alter the order of execution of commands. See page 398 for more information on control structures. You do not need to use braces around ***commands*** when you specify a single, simple command.

if...else

The **if...else** control structure tests the status returned by the ***condition*** and transfers control based on this status. The syntax of an **if...else** structure is shown below. The **else** part is optional.

```
if (condition)
        {commands}
    [else
        {commands}]
```

The simple **if** statement shown here does not use braces:

```
if ($5 <= 5000) print $0
```

Next is a gawk program that uses a simple **if...else** structure. Again, there are no braces.

```
$ cat if1
BEGIN   {
        nam="sam"
        if (nam == "max")
                print "nam is max"
            else
                print "nam is not max, it is", nam
        }
$ gawk -f if1
nam is not max, it is sam
```

while

The **while** structure loops through and executes the ***commands*** as long as the ***condition*** is *true*. The syntax of a **while** structure is

```
while (condition)
    {commands}
```

The next gawk program uses a simple **while** structure to display powers of 2. This example uses braces because the **while** loop contains more than one statement. This program does not accept input; all processing takes place when gawk executes the statements associated with the BEGIN pattern.

```
$ cat while1
BEGIN{
    n = 1
    while (n <= 5)
        {
        print "2^" n, 2**n
        n++
        }
    }

$ gawk -f while1
1^2 2
2^2 4
3^2 8
4^2 16
5^2 32
```

for

The syntax of a **for** control structure is

*for (**init; condition; increment**)*
    *{**commands**}*

A **for** structure starts by executing the ***init*** statement, which usually sets a counter to 0 or 1. It then loops through the ***commands*** as long as the ***condition*** remains *true*. After each loop it executes the ***increment*** statement. The **for1** gawk program does the same thing as the preceding **while1** program except that it uses a **for** statement, which makes the program simpler:

```
$ cat for1
BEGIN   {
        for (n=1; n <= 5; n++)
        print "2^" n, 2**n
        }

$ gawk -f for1
1^2 2
2^2 4
3^2 8
4^2 16
5^2 32
```

The gawk utility supports an alternative **for** syntax for working with associative arrays:

*for (**var in array**)*
    *{**commands**}*

This **for** structure loops through elements of the associative array named *array*, assigning the value of the index of each element of *array* to *var* each time through the loop. The following line of code (from the program on page 551) demonstrates a **for** structure:

```
END    {for (name in manuf) print name, manuf[name]}
```

break

The **break** statement transfers control out of a **for** or **while** loop, terminating execution of the innermost loop it appears in.

continue

The **continue** statement transfers control to the end of a **for** or **while** loop, causing execution of the innermost loop it appears in to continue with the next iteration.

EXAMPLES

cars data file

Many of the examples in this section work with the **cars** data file. From left to right, the columns in the file contain each car's make, model, year of manufacture, mileage in thousands of miles, and price. All whitespace in this file is composed of single TABs (the file does not contain any SPACEs).

```
$ cat cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

Missing pattern

A simple gawk program is

```
{ print }
```

This program consists of one program line that is an *action*. Because the *pattern* is missing, gawk selects all lines of input. When used without any arguments the **print** command displays each selected line in its entirety. This program copies the input to standard output.

```
$ gawk '{ print }' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

Missing action

The next program has a ***pattern*** but no explicit ***action***. The slashes indicate that **chevy** is a regular expression.

```
/chevy/
```

In this case gawk selects from the input just those lines that contain the string **chevy**. When you do not specify an ***action***, gawk assumes the ***action*** is **print**. The following example copies to standard output all lines from the input that contain the string **chevy**:

```
$ gawk '/chevy/' cars
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
chevy   impala  1985    85      1550
```

Single quotation marks

Although neither gawk nor shell syntax requires single quotation marks on the command line, it is still a good idea to use them because they can prevent problems. If the gawk program you create on the command line includes SPACEs or characters that are special to the shell, you must quote them. Always enclosing the program in single quotation marks is the easiest way to make sure you have quoted any characters that need to be quoted.

Fields

The next example selects all lines from the file (it has no ***pattern***). The braces enclose the ***action;*** you must always use braces to delimit the ***action*** so gawk can distinguish it from the ***pattern***. This example displays the third field (**$3**), a SPACE (the output field separator, indicated by the comma), and the first field (**$1**) of each selected line:

```
$ gawk '{print $3, $1}' cars
1970 plym
1999 chevy
1965 ford
1998 volvo
...
```

The next example, which includes both a ***pattern*** and an ***action***, selects all lines that contain the string **chevy** and displays the third and first fields from the selected lines:

```
$ gawk '/chevy/ {print $3, $1}' cars
1999 chevy
2000 chevy
1985 chevy
```

In the following example, gawk selects lines that contain a match for the regular expression **h**. Because there is no explicit ***action***, gawk displays all the lines it selects.

```
$ gawk '/h/' cars
chevy   malibu  1999    60      3000
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
chevy   impala  1985    85      1550
```

~ (matches operator)

The next *pattern* uses the matches operator (~) to select all lines that contain the letter **h** in the first field:

```
$ gawk '$1 ~ /h/' cars
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
chevy   impala  1985    85      1550
```

The caret (^) in a regular expression forces a match at the beginning of the line (page 890) or, in this case, at the beginning of the first field:

```
$ gawk '$1 ~ /^h/' cars
honda   accord  2001    30      6000
```

Brackets surround a character class definition (page 889). In the next example, gawk selects lines that have a second field that begins with **t** or **m** and displays the third and second fields, a dollar sign, and the fifth field. Because there is no comma between the **"$"** and the **$5**, gawk does not put a SPACE between them in the output.

```
$ gawk '$2 ~ /^[tm]/ {print $3, $2, "$"  $5}' cars
1999 malibu $3000
1965 mustang $10000
2003 thundbd $10500
2000 malibu $3500
2004 taurus $17000
```

Dollar signs

The next example shows three roles a dollar sign can play in a gawk program. First, a dollar sign followed by a number names a field. Second, within a regular expression a dollar sign forces a match at the end of a line or field (**5$**). Third, within a string a dollar sign represents itself.

```
$ gawk '$3 ~ /5$/ {print $3, $1, "$"  $5}' cars
1965 ford $10000
1985 bmw $450
1985 chevy $1550
```

In the next example, the equal-to relational operator (**==**) causes gawk to perform a numeric comparison between the third field in each line and the number **1985**. The gawk command takes the default *action,* **print**, on each line where the comparison is *true*.

```
$ gawk '$3 == 1985' cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
```

The next example finds all cars priced at or less than $3,000.

```
$ gawk '$5 <= 3000' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
bmw     325i    1985    115     450
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
```

Textual comparisons

When you use double quotation marks, gawk performs textual comparisons by using the ASCII (or other local) collating sequence as the basis of the comparison. In the following example, gawk shows that the *strings* **450** and **750** fall in the range that lies between the *strings* **2000** and **9000**, which is probably not the intended result.

```
$ gawk '"2000" <= $5 && $5 < "9000"' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
toyota  rav4    2002    180     750
```

When you need to perform a numeric comparison, do not use quotation marks. The next example gives the intended result. It is the same as the previous example except it omits the double quotation marks.

```
$ gawk '2000 <= $5 && $5 < 9000' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
```

, (range operator)

The range operator (**,**) selects a group of lines. The first line it selects is the one specified by the ***pattern*** before the comma. The last line is the one selected by the ***pattern*** after the comma. If no line matches the ***pattern*** after the comma, gawk selects every line through the end of the input. The next example selects all lines, starting with the line that contains **volvo** and ending with the line that contains **bmw**.

```
$ gawk '/volvo/ , /bmw/' cars
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
```

After the range operator finds its first group of lines, it begins the process again, looking for a line that matches the ***pattern*** before the comma. In the following example, gawk finds three groups of lines that fall between **chevy** and **ford**. Although the fifth line of input contains **ford**, gawk does not select it because at the time it is processing the fifth line, it is searching for **chevy**.

```
$ gawk '/chevy/ , /ford/' cars
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
```

```
ford    taurus  2004    10      17000
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

--file option When you are writing a longer gawk program, it is convenient to put the program in a file and reference the file on the command line. Use the **–f** (**––file**) option followed by the name of the file containing the gawk program.

BEGIN The following gawk program, which is stored in a file named **pr_header**, has two ***actions*** and uses the **BEGIN** ***pattern***. The gawk utility performs the ***action*** associated with **BEGIN** before processing any lines of the data file: It displays a header. The second ***action***, {**print**}, has no ***pattern*** part and displays all lines from the input.

```
$ cat pr_header
BEGIN   {print "Make    Model   Year    Miles   Price"}
        {print}

$ gawk -f pr_header cars
Make    Model   Year    Miles   Price
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

The next example expands the ***action*** associated with the **BEGIN** ***pattern***. In the previous and the following examples, the whitespace in the headers is composed of single TABs, so the titles line up with the columns of data.

```
$ cat pr_header2
BEGIN   {
print "Make     Model   Year    Miles   Price"
print "---------------------------------------"
}
        {print}

$ gawk -f pr_header2 cars
Make    Model   Year    Miles   Price
---------------------------------------
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

COPROCESS: TWO-WAY I/O

A *coprocess* is a process that runs in parallel with another process. Starting with version 3.1, gawk can invoke a coprocess to exchange information directly with a background process. A coprocess can be useful when you are working in a client/server environment, setting up an *SQL* (page 980) front end/back end, or exchanging data with a remote system over a network. The gawk syntax identifies a coprocess by preceding the name of the program that starts the background process with a |& operator.

Only gawk supports coprocesses

tip The awk and mawk utilities do not support coprocesses. Only gawk supports coprocesses.

The coprocess command must be a filter (i.e., it reads from standard input and writes to standard output) and must flush its output whenever it has a complete line rather than accumulating lines for subsequent output. When a command is invoked as a coprocess, it is connected via a two-way pipe to a gawk program so you can read from and write to the coprocess.

to_upper When used alone the tr utility (page 864) does not flush its output after each line. The **to_upper** shell script is a wrapper for tr that does flush its output; this filter can be run as a coprocess. For each line read, **to_upper** writes the line, translated to uppercase, to standard output. Remove the **#** before **set –x** if you want **to_upper** to display debugging output.

```
$ cat to_upper
#!/bin/bash
#set -x
while read arg
do
    echo "$arg" | tr '[a-z]' '[A-Z]'
done

$ echo abcdef | ./to_upper
ABCDEF
```

The **g6** program invokes **to_upper** as a coprocess. This gawk program reads standard input or a file specified on the command line, translates the input to uppercase, and writes the translated data to standard output.

```
$ cat g6
    {
    print $0 |& "to_upper"
    "to_upper" |& getline hold
    print hold
    }

$ gawk -f g6 < alpha
AAAAAAAAA
BBBBBBBBB
CCCCCCCCC
DDDDDDDDD
```

The **g6** program has one compound statement, enclosed within braces, comprising three statements. Because there is no ***pattern***, gawk executes the compound statement once for each line of input.

In the first statement, **print $0** sends the current record to standard output. The **|&** operator redirects standard output to the program named **to_upper**, which is running as a coprocess. The quotation marks around the name of the program are required. The second statement redirects standard output from **to_upper** to a **getline** statement, which copies its standard input to the variable named **hold**. The third statement, **print hold**, sends the contents of the **hold** variable to standard output.

GETTING INPUT FROM A NETWORK

Building on the concept of a coprocess, gawk can exchange information with a process on another system via an IP network connection. When you specify one of the special filenames that begins with **/inet/**, gawk processes the request using a network connection. The format of these special filenames is

*/inet/**protocol**/**local-port**/**remote-host**/**remote-port***

where ***protocol*** is usually **tcp** but can be **udp**, ***local-port*** is 0 (zero) if you want gawk to pick a port (otherwise it is the number of the port you want to use), ***remote-host*** is the *IP address* (page 960) or *fully qualified domain name* (page 955) of the remote host, and ***remote-port*** is the port number on the remote host. Instead of a port number in ***local-port*** and ***remote-port***, you can specify a service name such as **http** or **ftp**.

The **g7** program reads the **rfc-retrieval.txt** file from the server at **www.rfc-editor.org**. On **www.rfc-editor.org** the file is located at **/rfc/rfc-retrieval.txt**. The first statement in **g7** assigns the special filename to the **server** variable. The filename specifies a TCP connection, allows the local system to select an appropriate port, and connects to **www.rfc-editor.org** on port 80. You can use **http** in place of **80** to specify the standard HTTP port.

The second statement uses a coprocess to send a **GET** request to the remote server. This request includes the pathname of the file gawk is requesting. A **while** loop uses a coprocess to redirect lines from the server to **getline**. Because **getline** has no variable name as an argument, it saves its input in the current record buffer **$0**. The final **print** statement sends each record to standard output. Experiment with this script, replacing the final **print** statement with gawk statements that process the file.

```
$ cat g7
BEGIN           {
    # set variable named server
    # to special networking filename
    server = "/inet/tcp/0/www.rfc-editor.org/80"

    # use coprocess to send GET request to remote server
    print "GET /rfc/rfc-retrieval.txt" |& server

    # while loop uses coprocess to redirect
    # output from server to getline
    while (server |& getline)
        print $0
    }
```

```
$ gawk -f g7

                   Where and how to get new RFCs
                   =============================

RFCs may be obtained via FTP or HTTP or email from many RFC repositories.
The official repository for RFCs is:

        http://www.rfc-editor.org/
...
```

Blank

PART V
Command Reference

Blank

sample OS X ←*OS X in an oval indicates this utility runs under Mac OS X only.*

Brief description of what the utility does

*sample [**options**] **arguments***

Following the syntax line is a description of the utility. The syntax line shows how to run the utility from the command line. Options and arguments enclosed in brackets (*[]*) are not required. Enter words that appear in *this italic typeface* as is. Words that you must substitute when you type appear in ***this bold italic typeface***. Words listed as arguments to a command identify single arguments (for example, ***source-file***) or groups of similar arguments (for example, ***directory-list***).

Arguments

This section describes the arguments you can use when you run the utility. The argument itself, as shown in the preceding syntax line, is printed in ***this bold italic typeface***.

Options

This section lists some of the options you can use with the command. Unless otherwise specified, you must precede options with one or two hyphens. Most commands accept a single hyphen before multiple options (page 119). Options in this section are ordered alphabetically by short (single-hyphen) options. If an option has only a long version (two hyphens), it is ordered by its long option. Following are some sample options:

––delimiter=*dchar*

–d ***dchar***

This option includes an argument. The argument is set in a ***bold italic typeface*** in both the heading and the description. You substitute another word (filename, string of characters, or other value) for any arguments you see in ***this typeface***. Type characters that are in **bold type** (such as the **––delimiter** and **–d**) as is.

––make-dirs **–m** This option has a long and a short version. You can use either option; they are equivalent. This option description ends with **Linux** in a box, indicating it is available under Linux only. Options *not* followed by **Linux** or **OS X** are available under both operating systems. LINUX

–t (**table of contents**) This simple option is preceded by a single hyphen and not followed by arguments. It has no long version. The **table of contents** appearing in parentheses at the beginning of the description is a cue, suggestive of what the option letter stands for. This option description ends with **OS X** in a box, indicating it is available under OS X only. Options *not* followed by **Linux** or **OS X** are available under both operating systems. OS X

Discussion This optional section describes how to use the utility and identifies any quirks it may have.

Notes This section contains miscellaneous notes—some important and others merely interesting.

Examples This section contains examples illustrating how to use the utility. This section is a tutorial, so it takes a more casual tone than the preceding sections of the description.

bzip2

Compresses or decompresses files

*bzip2 [**options**] [**file-list**]*
*bunzip2 [**options**] [**file-list**]*
*bzcat [**options**] [**file-list**]*
*bzip2recover [**file**]*

The bzip2 utility compresses files; bunzip2 restores files compressed with bzip2; and bzcat displays files compressed with bzip2.

Arguments

The ***file-list*** is a list of one or more files (no directories) that are to be compressed or decompressed. If ***file-list*** is empty or if the special option **–** is present, bzip2 reads from standard input. The **––stdout** option causes bzip2 to write to standard output.

Options

Under Linux, bzip2, bunzip2, and bzcat accept the common options described on page 603.

The Mac OS X version of bzip2 accepts long options

tip Options for bzip2 preceded by a double hyphen (**––**) work under Mac OS X as well as under Linux.

––stdout	**–c**	Writes the results of compression or decompression to standard output.
––decompress	**–d**	Decompresses a file compressed with bzip2. This option with bzip2 is equivalent to the bunzip2 command.
––fast *or* **––best**	***–n***	Sets the block size when compressing a file. The ***n*** is a digit from 1 to 9, where 1 (**––fast**) generates a block size of 100 kilobytes and 9 (**––best**) generates a block size of 900 kilobytes. The default level is 9. The options **––fast** and **––best** are provided for compatibility with gzip and do not necessarily yield the fastest or best compression.
––force	**–f**	Forces compression even if a file already exists, has multiple links, or comes directly from a terminal. The option has a similar effect with bunzip2.
––keep	**–k**	Does not delete input files while compressing or decompressing them.
––quiet	**–q**	Suppresses warning messages; does display critical messages.
––test	**–t**	Verifies the integrity of a compressed file. Displays nothing if the file is OK.
––verbose	**–v**	For each file being compressed, displays the name of the file, the compression ratio, the percentage of space saved, and the sizes of the decompressed and compressed files.

Discussion

The bzip2 and bunzip2 utilities work similarly to gzip and gunzip; see the discussion of gzip (page 725) for more information. Normally bzip2 does not overwrite a file; you must use **––force** to overwrite a file during compression or decompression.

Notes

See page 62 for additional information on and examples of tar.

The bzip2 home page is bzip.org.

The bzip2 utility does a better job of compressing files than does gzip.

Use the **––bzip2** modifier with tar (page 847) to compress archive files with bzip2.

bzcat ***file-list*** Works like cat except it uses bunzip2 to decompress ***file-list*** as it copies files to standard output.

bzip2recover Attempts to recover a damaged file that was compressed with bzip2.

Examples

In the following example, bzip2 compresses a file and gives the resulting file the same name with a **.bz2** filename extension. The **–v** option displays statistics about the compression.

```
$ ls -l
total 728
-rw-r--r--  1 sam sam 737414 Feb 20 19:05 bigfile
$ bzip2 -v bigfile
  bigfile: 3.926:1, 2.037 bits/byte, 74.53% saved, 737414 in, 187806 out
$ ls -l
total 188
-rw-r--r--  1 sam sam 187806 Feb 20 19:05 bigfile.bz2
```

Next touch creates a file with the same name as the original file; bunzip2 refuses to overwrite the file in the process of decompressing **bigfile.bz2**. The **––force** option enables bunzip2 to overwrite the file.

```
$ touch bigfile
$ bunzip2 bigfile.bz2
bunzip2: Output file bigfile already exists.
$ bunzip2 --force bigfile.bz2
$ ls -l
total 728
-rw-r--r--  1 sam sam 737414 Feb 20 19:05 bigfile
```

Blank

cp

Copies files

*cp [**options**] **source-file destination-file***
*cp [**options**] **source-file-list destination-directory***

The cp utility copies one or more files. It can either make a copy of a single file (first format) or copy one or more files to a directory (second format). With the **–R** option, cp can copy directory hierarchies.

Arguments

The ***source-file*** is the pathname of the file that cp makes a copy of. The ***destination-file*** is the pathname that cp assigns to the resulting copy of the file.

The ***source-file-list*** is a list of one or more pathnames of files that cp makes copies of. The ***destination-directory*** is the pathname of the directory in which cp places the copied files. With this format, cp gives each copied file the same simple filename as its ***source-file***.

The **–R** option enables cp to copy directory hierarchies recursively from the ***source-file-list*** into the ***destination-directory***.

Options

Under Linux, cp accepts the common options described on page 603. Options preceded by a double hyphen (––) work under Linux only. Except as noted, options named with a single letter and preceded by a single hyphen work under Linux and OS X.

––archive **–a** Attempts to preserve the owner, group, permissions, access date, and modification date of source file(s) while copying recursively without dereferencing symbolic links. Same as **–dpR**. LINUX

––backup **–b** If copying a file would remove or overwrite an existing file, this option makes a backup copy of the file that would be overwritten. The backup copy has the same name as the ***destination-file*** with a tilde (~) appended to it. When you use both **––backup** and **––force**, cp makes a backup copy when you try to copy a file over itself. For more backup options, search for **Backup options** in the **core utils** info page. LINUX

–d For each file that is a symbolic link, copies the symbolic link, not the file the link points to. Also preserves hard links in ***destination-files*** that exist between corresponding ***source-files***. This option is equivalent to **––no-dereference** and **––preserve=links**. LINUX

––force **–f** When the ***destination-file*** exists and cannot be opened for writing, causes cp to try to remove ***destination-file*** before copying ***source-file***. This option is useful when the user copying a file does not have write permission to an existing ***destination-file*** but does have write permission to the directory containing the

destination-file. Use this option with **–b** to back up a destination file before removing or overwriting it.

–H (**partial dereference**) For each file that is a symbolic link, copies the file the link points to, not the symbolic link itself. This option affects files specified on the command line; it does not affect files found while descending a directory hierarchy. This option treats files that are not symbolic links normally. Under OS X works with **–R** only. See page 623 for an example of the use of the **–H** versus **–L** options.

––interactive **–i** Prompts you whenever cp would overwrite a file. If you respond with a string that starts with **y** or **Y**, cp copies the file. If you enter anything else, cp does not copy the file.

––dereference **–L** (**dereference**) For each file that is a symbolic link, copies the file the link points to, not the symbolic link itself. This option affects all files and treats files that are not symbolic links normally. Under OS X works with **–R** only. See page 623 for an example of the use of the **–H** versus **–L** options.

––no-dereference **–P** (**no dereference**) For each file that is a symbolic link, copies the symbolic link, not the file the link points to. This option affects all files and treats files that are not symbolic links normally. Under OS X works with **–R** only. See page 625 for an example of the use of the **–P** option.

––preserve[**=***attr*] **–p** Creates a ***destination-file*** with the same owner, group, permissions, access date, and modification date as the ***source-file***. The **–p** option does not take an argument.

Without ***attr***, **––preserve** works as described above. The ***attr*** is a comma-separated list that can include **mode** (permissions and ACLs), **ownership** (owner and group), **timestamps** (access and modification dates), **links** (hard links), and **all** (all attributes).

––parents Copies a relative pathname to a directory, creating directories as needed. See the "Examples" section. LINUX

––recursive **–R** *or* **–r**
Recursively copies directory hierarchies including ordinary files. Under Linux, the **––no-dereference** (**–d**) option is implied: With the **–R**, **–r**, or **––recursive** option, cp copies the links (not the files the links point to). The **–r** and **––recursive** options are available under Linux only.

––update **–u** Copies only when the ***destination-file*** does not exist or when it is older than the ***source-file*** (i.e., this option will not overwrite a newer destination file). LINUX

––verbose **–v** Displays the name of each file as cp copies it.

Notes

Under Linux, cp dereferences symbolic links unless you use one or more of the **–R**, **–r**, **––recursive**, **–P**, **–d**, or **––no-dereference** options. As explained on the previous page, under Linux the **–H** option dereferences only symbolic links listed on the command line. Under Mac OS X, without the **–R** option, cp always dereferences symbolic links; with the **–R** option, cp does not dereference symbolic links (**–P** is the default) unless you specify **–H** or **–L**.

Many options are available for cp under Linux. See the **coreutils** info page for a complete list.

If the ***destination-file*** exists before you execute a cp command, cp overwrites the file, destroying the contents but leaving the access privileges, owner, and group associated with the file as they were.

If the ***destination-file*** does not exist, cp uses the access privileges of the ***source-file***. The user who copies the file becomes the owner of the ***destination-file*** and the user's login group becomes the group associated with the ***destination-file***.

Using the **–p** option without any arguments causes cp to attempt to set the owner, group, permissions, access date, and modification date to match those of the ***source-file***.

Unlike with the ln utility (page 740), the ***destination-file*** that cp creates is independent of its ***source-file***.

Under Mac OS X version 10.4 and later, cp copies extended attributes (page 928).

Examples

The first command makes a copy of the file **letter** in the working directory. The name of the copy is **letter.sav**.

```
$ cp letter letter.sav
```

The next command copies all files with filenames ending in **.c** into the **archives** directory, which is a subdirectory of the working directory. Each copied file retains its simple filename but has a new absolute pathname. The **–p** (**––preserve**) option causes the copied files in **archives** to have the same owner, group, permissions, access date, and modification date as the source files.

```
$ cp -p *.c archives
```

The next example copies **memo** from Sam's home directory to the working directory:

```
$ cp ~sam/memo .
```

The next example runs under Linux and uses the **––parents** option to copy the file **memo/thursday/max** to the **dir** directory as **dir/memo/thursday/max**. The find utility shows the newly created directory hierarchy.

```
$ cp --parents memo/thursday/max dir
$ find dir
dir
dir/memo
dir/memo/thursday
dir/memo/thursday/max
```

The following command copies the files named **memo** and **letter** into another directory. The copies have the same simple filenames as the source files (**memo** and **letter**) but have different absolute pathnames. The absolute pathnames of the copied files are **/home/sam/memo** and **/home/sam/letter**, respectively.

```
$ cp memo letter /home/sam
```

The final command demonstrates one use of the **–f** (**––force**) option. Max owns the working directory and tries unsuccessfully to copy **one** over another file (**me**) that he does not have write permission for. Because he has write permission to the directory that holds **me**, Max can remove the file but cannot write to it. The **–f** (**––force**) option unlinks, or removes, **me** and then copies **one** to the new file named **me**.

```
$ ls -ld
drwxrwxr-x    2 max max 4096 Oct 21 22:55 .
$ ls -l
-rw-r--r--    1 root root 3555 Oct 21 22:54 me
-rw-rw-r--    1 max max 1222 Oct 21 22:55 one
$ cp one me
cp: cannot create regular file 'me': Permission denied
$ cp -f one me
$ ls -l
-rw-r--r--    1 max max 1222 Oct 21 22:58 me
-rw-rw-r--    1 max max 1222 Oct 21 22:55 one
```

If Max had used the **–b** (**––backup**) option in addition to **–f** (**––force**), cp would have created a backup of **me** named **me~**. Refer to "Directory Access Permissions" on page 98 for more information.

cut

Selects characters or fields from input lines

*cut [**options**] [**file-list**]*

The cut utility selects characters or fields from lines of input and writes them to standard output. Character and field numbering start with 1.

Arguments

The ***file-list*** is a list of ordinary files. If you do not specify an argument or if you specify a hyphen (**–**) in place of a filename, cut reads from standard input.

Options

Under Linux, cut accepts the common options described on page 603. Options preceded by a double hyphen (**––**) work under Linux only. Options named with a single letter and preceded by a single hyphen work under Linux and OS X.

––characters=*clist*
–c ***clist***
Selects the characters given by the column numbers in ***clist***. The value of ***clist*** is one or more comma-separated column numbers or column ranges. A range is specified by two column numbers separated by a hyphen. A range of ***–n*** means columns **1** through ***n; n–*** means columns ***n*** through the end of the line.

––delimiter=*dchar*
–d ***dchar***
Specifies ***dchar*** as the input field delimiter. Also specifies ***dchar*** as the output field delimiter unless you use the **––output-delimiter** option. The default delimiter is a TAB character. Quote characters as necessary to protect them from shell expansion.

––fields=*flist* **–f** ***flist***
Selects the fields specified in ***flist***. The value of ***flist*** is one or more comma-separated field numbers or field ranges. A range is specified by two field numbers separated by a hyphen. A range of ***–n*** means fields **1** through ***n; n–*** means fields ***n*** through the last field. The field delimiter is a TAB character unless you use the **––delimiter** option to change it.

––output-delimiter=*ochar*
Specifies ***ochar*** as the output field delimiter. The default delimiter is the TAB character. You can specify a different delimiter by using the **––delimiter** option. Quote characters as necessary to protect them from shell expansion.

--only-delimited **–s** Copies only lines containing delimiters. Without this option, cut copies—but does not modify—lines that do not contain delimiters.

Notes

Although limited in functionality, cut is easy to learn and use and is a good choice when columns and fields can be selected without using pattern matching. Sometimes cut is used with paste (page 784).

Examples

For the next two examples, assume that an **ls –l** command produces the following output:

```
$ ls -l
total 2944
-rwxr-xr-x  1 zach pubs     259 Feb  1 00:12 countout
-rw-rw-r--  1 zach pubs    9453 Feb  4 23:17 headers
-rw-rw-r--  1 zach pubs 1474828 Jan 14 14:15 memo
-rw-rw-r--  1 zach pubs 1474828 Jan 14 14:33 memos_save
-rw-rw-r--  1 zach pubs    7134 Feb  4 23:18 tmp1
-rw-rw-r--  1 zach pubs    4770 Feb  4 23:26 tmp2
-rw-rw-r--  1 zach pubs   13580 Nov  7 08:01 typescript
```

The following command outputs the permissions of the files in the working directory. The cut utility with the **–c** option selects characters 2 through 10 from each input line. The characters in this range are written to standard output.

```
$ ls -l | cut -c2-10
otal 2944
rwxr-xr-x
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
```

The next command outputs the size and name of each file in the working directory. The **–f** option selects the fifth and ninth fields from the input lines. The **–d** option tells cut to use SPACEs, not TABs, as delimiters. The tr utility (page 864) with the **–s** option changes sequences of more than one SPACE character into a single SPACE; otherwise, cut counts the extra SPACE characters as separate fields.

```
$ ls -l | tr -s '  ' ' ' | cut -f5,9 -d' '

259 countout
9453 headers
1474828 memo
1474828 memos_save
7134 tmp1
4770 tmp2
13580 typescript
```

The last example displays a list of full names as stored in the fifth field of the **/etc/passwd** file. The **–d** option specifies that the colon character be used as the field

delimiter. Although this example works under Mac OS X, **/etc/passwd** does not contain information about most users; see "Open Directory" on page 926 for more information.

```
$ cat /etc/passwd
root:x:0:0:Root:/:/bin/sh
sam:x:401:50:Sam the Great:/home/sam:/bin/zsh
max:x:402:50:Max Wild:/home/max:/bin/bash
zach:x:504:500:Zach Brill:/home/zach:/bin/tcsh
hls:x:505:500:Helen Simpson:/home/hls:/bin/bash

$ cut -d: -f5 /etc/passwd
Root
Sam the Great
Max Wild
Zach Brill
Helen Simpson
```

ditto OS X

Copies files and creates and unpacks archives

*ditto [**options**] **source-file destination-file***
*ditto [**options**] **source-file-list destination-directory***
*ditto **–c** [**options**] **source-directory destination-archive***
*ditto **–x** [**options**] **source-archive-list destination-directory***

The ditto utility copies files and their ownership, timestamps, and other attributes, including extended attributes (page 928). It can copy to and from cpio and zip archive files, as well as copy ordinary files and directories.

Arguments

The ***source-file*** is the pathname of the file that ditto is to make a copy of. The ***destination-file*** is the pathname that ditto assigns to the resulting copy of the file.

The ***source-file-list*** specifies one or more pathnames of files and directories that ditto makes copies of. The ***destination-directory*** is the pathname of the directory that ditto copies the files and directories into. When you specify a ***destination-directory***, ditto gives each of the copied files the same simple filename as its ***source-file***.

The ***source-directory*** is a single directory that ditto copies into the ***destination-archive***. The resulting archive holds copies of the contents of ***source-directory***, but not the directory itself.

The ***source-archive-list*** specifies one or more pathnames of archives that ditto extracts into ***destination-directory***.

Using a hyphen (**–**) in place of a filename or a directory name causes ditto to read from standard input or write to standard output instead of reading from or writing to that file or directory.

Options

You cannot use the **–c** and **–x** options together.

–c (**create archive**) Creates an archive file.

––help Displays a help message.

–k (**pkzip**) Uses the zip format, instead of the default cpio (page 644) format, to create or extract archives. For more information on zip, see the tip on page 62.

––norsrc (**no resource**) Ignores extended attributes. This option causes ditto to copy only data forks (the default behavior under Mac OS X 10.3 and earlier).

––rsrc (**resource**) Copies extended attributes, including resource forks (the default behavior under Mac OS X 10.4 and later). Also **–rsrc** and **–rsrcFork**.

–V (**very verbose**) Sends a line to standard error for each file, symbolic link, and device node ditto copies.

–v (**verbose**) Sends a line to standard error for each directory ditto copies.

–X (**exclude**) Prevents ditto from searching directories in filesystems other than the filesystems that hold the files it was explicitly told to copy.

–x (**extract archive**) Extracts files from an archive file.

–z (**compress**) Uses gzip (page 724) or gunzip to compress or decompress cpio archives.

Notes

The ditto utility does not copy the locked attribute flag (page 931). The utility also does not copy ACLs.

By default ditto creates and reads *archives* (page 941) in the cpio (page 644) format.

The ditto utility cannot list the contents of archive files; it can only create or extract files from archives. Use pax or cpio to list the contents of cpio archives, and use unzip with the **–l** option to list the contents of zip files.

Examples

The following examples show three ways to back up a user's home directory, including extended attributes (except as mentioned in "Notes"), preserving timestamps and permissions. The first example copies Zach's home directory to the volume (filesystem) named **Backups**; the copy is a new directory named **zach.0228**:

```
$ ditto /Users/zach /Volumes/Backups/zach.0228
```

The next example copies Zach's home directory into a single cpio-format archive file on the volume named **Backups**:

```
$ ditto -c /Users/zach /Volumes/Backups/zach.0228.cpio
```

The next example copies Zach's home directory into a zip archive:

```
$ ditto -c -k /Users/zach /Volumes/Backups/zach.0228.zip
```

Each of the next three examples restores the corresponding backup archive into Zach's home directory, overwriting any files that are already there:

```
$ ditto /Volumes/Backups/zach.0228 /Users/zach
$ ditto -x /Volumes/Backups/zach.0228.cpio /Users/zach
$ ditto -x -k /Volumes/Backups/zach.0228.zip /Users/zach
```

The following example copies the **Scripts** directory to a directory named **Scripts-Backups** on the remote host **bravo**. It uses an argument of a hyphen in place of ***source-directory*** locally to write to standard output and in place of ***destination-directory*** on the remote system to read from standard input:

```
$ ditto -c Scripts - | ssh bravo ditto -x - ScriptsBackups
```

The final example copies the local startup disk (the root filesystem) to the volume named **Backups.root**. Because some of the files can be read only by **root**, the script must be run by a user with **root** privileges. The **–X** option keeps ditto from trying to copy other volumes (filesystems) that are mounted under **/**.

```
# ditto -X / /Volumes/Backups.root
```

Blank

tr

Replaces specified characters

tr [options] string1 [string2]

The tr utility reads standard input and, for each input character, either maps it to an alternate character, deletes the character, or leaves the character as is. This utility reads from standard input and writes to standard output.

Arguments

The tr utility is typically used with two arguments, ***string1*** and ***string2***. The position of each character in the two strings is important: Each time tr finds a character from ***string1*** in its input, it replaces that character with the corresponding character from ***string2***.

With one argument, ***string1***, and the **–d** (**––delete**) option, tr deletes the characters specified in ***string1***. The option **–s** (**––squeeze-repeats**) replaces multiple sequential occurrences of characters in ***string1*** with single occurrences (for example, **abbc** becomes **abc**).

Ranges

A range of characters is similar in function to a character class within a regular expression (page 889). GNU tr does not support ranges (character classes) enclosed within brackets. You can specify a range of characters by following the character that appears earlier in the collating sequence with a hyphen and the character that comes later in the collating sequence. For example, **1–6** expands to **123456**. Although the range A–Z expands as you would expect in ASCII, this approach does not work when you use the EBCDIC collating sequence, as these characters are not sequential in EBCDIC. See "Character Classes" for a solution to this issue.

Character Classes

A tr character class is not the same as the character class described elsewhere in this book. (GNU documentation uses the term *list operator* for what this book calls a *character class*.) You specify a character class as '*[:class:]*', where ***class*** is one of the character classes from Table V-32. You must specify a character class in ***string1*** (and not ***string2***) unless you are performing case conversion (see the "Examples" section) or you use the **–d** and **–s** options together.

Table V-32 Character classes

Class	Meaning
alnum	Letters and digits
alpha	Letters

Table V-32 Character classes (continued)

Class	Meaning
blank	Whitespace
cntrl	CONTROL characters
digit	Digits
graph	Printable characters but not SPACES
lower	Lowercase letters
print	Printable characters including SPACES
punct	Punctuation characters
space	Horizontal or vertical whitespace
upper	Uppercase letters
xdigit	Hexadecimal digits

Options

Options preceded by a double hyphen (**––**) work under Linux only. Except as noted, options named with a single letter and preceded by a single hyphen work under Linux and OS X.

––complement **–c** Complements ***string1***, causing tr to match all characters *except* those in ***string1***.

––delete **–d** Deletes characters that match those specified in ***string1***. If you use this option with the **–s** (**––squeeze-repeats**) option, you must specify both ***string1*** and ***string2*** (see "Notes").

––help Summarizes how to use tr, including the special symbols you can use in ***string1*** and ***string2***. LINUX

––squeeze-repeats **–s** Replaces multiple sequential occurrences of a character in ***string1*** with a single occurrence of the character when you call tr with only one string argument. If you use both ***string1*** and ***string2***, the tr utility first translates the characters in ***string1*** to those in ***string2***; it then replaces multiple sequential occurrences of a character in ***string2*** with a single occurrence of the character.

––truncate-set1 **–t** Truncates ***string1*** so it is the same length as ***string2*** before processing input. LINUX

Notes

When ***string1*** is longer than ***string2***, the initial portion of ***string1*** (equal in length to ***string2***) is used in the translation. When ***string1*** is shorter than ***string2***, tr uses the last character of ***string1*** to extend ***string1*** to the length of ***string2***. In this case tr departs from the POSIX standard, which does not define a result.

If you use the **–d** (**––delete**) and **–s** (**––squeeze-repeats**) options at the same time, tr first deletes the characters in ***string1*** and then replaces multiple sequential occurrences of a character in ***string2*** with a single occurrence of the character.

Examples

You can use a hyphen to represent a range of characters in ***string1*** or ***string2***. The two command lines in the following example produce the same result:

```
$ echo abcdef | tr  'abcdef' 'xyzabc'
xyzabc
$ echo abcdef | tr  'a-f' 'x-za-c'
xyzabc
```

The next example demonstrates a popular method for disguising text, often called ROT13 (rotate 13) because it replaces the first letter of the alphabet with the thirteenth, the second with the fourteenth, and so forth.

```
$ echo The punchline of the joke is ... |
> tr 'A-M N-Z a-m n-z' 'N-Z A-M n-z a-m'
Gur chapuyvar bs gur wbxr vf ...
```

To make the text intelligible again, reverse the order of the arguments to tr:

```
$ echo Gur chapuyvar bs gur wbxr vf ... |
> tr 'N-Z A-M n-z a-m' 'A-M N-Z a-m n-z'
The punchline of the joke is ...
```

The **––delete** option causes tr to delete selected characters:

```
$ echo If you can read this, you can spot the missing vowels! |
> tr --delete 'aeiou'
If y cn rd ths, y cn spt th mssng vwls!
```

In the following example, tr replaces characters and reduces pairs of identical characters to single characters:

```
$ echo tennessee | tr -s 'tnse' 'srne'
serene
```

The next example replaces each sequence of nonalphabetic characters (the complement of all the alphabetic characters as specified by the character class **alpha**) in the file **draft1** with a single NEWLINE character. The output is a list of words, one per line.

```
$ tr -c -s '[:alpha:]' '\n' < draft1
```

The next example uses character classes to upshift the string **hi there**:

```
$ echo hi there | tr '[:lower:]' '[:upper:]'
HI THERE
```