EXCERPT FROM CHAPTER 5: THE SHELL

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 as mentioned earlier. 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 or the screen. This section demonstrates how to redirect input from and output to ordinary text files and utilities.

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.

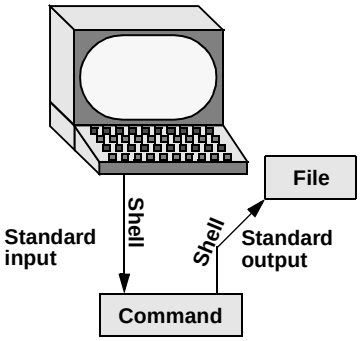

Figure 5-6 Redirecting standard output

Figure 5-7 uses cat to demonstrate output redirection. This figure contrasts with Figure 5-3 on page 114, where both standard input *and* standard output are associated with the keyboard and the 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.

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 "Redirecting output can destroy a file II" caution on page 120.

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

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

Figure 5-7 cat with its output redirected

```
$ cat stationery
2,000 sheets letterhead ordered:   10/7/05
$ cat tape
1 box masking tape ordered:        10/14/05
5 boxes filament tape ordered:     10/28/05
$ cat pens
12 doz. black pens ordered:        10/4/05

$ cat stationery tape pens > supply_orders

$ cat supply_orders
2,000 sheets letterhead ordered:   10/7/05
1 box masking tape ordered:        10/14/05
5 boxes filament tape ordered:     10/28/05
12 doz. black pens ordered:        10/4/05
$
```

Figure 5-8 Using cat to catenate 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. In this case standard output is redirected to the file **supply_orders**. The final cat command shows that **supply_orders** contains the contents of all three files.

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 that was 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 (end of file) signal at the end of an ordinary file.

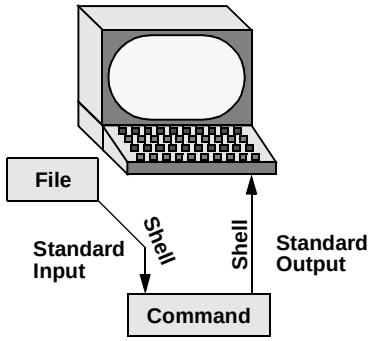

Figure 5-9 Redirecting standard input

```
$ cat < supply_orders
2,000 sheets letterhead ordered:   10/7/05
1 box masking tape ordered:        10/14/05
5 boxes filament tape ordered:     10/28/05
12 doz. black pens ordered:        10/4/05
```

Figure 5-10 cat with its input redirected

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, and grep. These utilities first examine the command line that you use to call them. 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.

noclobber: AVOIDS OVERWRITING FILES

The shell provides a feature called **noclobber** that stops you from inadvertently overwriting an existing file using redirection. When you enable this feature by setting the **noclobber** variable and then attempt to redirect output to an existing file, the shell displays an error message and does not execute the command. If the preceding examples result in one of the following messages, the **noclobber** feature has been set. The following examples set **noclobber**, attempt to redirect the output from echo into an existing file, and then unset **noclobber** under bash and tcsh:

bash

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

tcsh

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

You can override **noclobber** by putting a pipe symbol (tcsh uses an exclamation point) after the symbol you use for redirecting output (>|).

In the following example, the user first creates a file named **a** by redirecting the output of date to the file. Next the user sets the **noclobber** variable and tries redirecting output to **a** again. The shell returns an error message. Then the user tries the same thing but using a pipe symbol after the redirect symbol. This time the shell allows the user to overwrite the file. Finally, the user unsets **noclobber** (using a plus sign in place of the hyphen) and verifies that it is no longer set.

```
$ date > a
$ set -o noclobber
$ date > a
bash: a: Cannot overwrite existing file
$ date >| a
$ set +o noclobber
$ date > a
```

For more information on using **noclobber** under tcsh, refer to page 367.

Redirecting output can destroy a file II

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

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

Although cat displays an error message, the shell goes ahead and 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 this third 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 48) 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 that the command is running correctly. Even after you see the error message, it may take a while to realize that you destroyed the contents of **a**.

APPENDING STANDARD OUTPUT TO A FILE

The *append output symbol* (>>) causes the shell to add new information to the end of a file, leaving any 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
```

You first see the contents of the **orange** file. Next the contents of the **pear** file is added to the end of (catenated with) the **orange** file. The final cat shows 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 on, you will overwrite the **orange** file. Even if you have the **noclobber** feature turned on, it is a good idea to keep backup copies of files you are manipulating in these ways in case you make a mistake.

Although it protects you from making an erroneous redirection, **noclobber** does not stop you from overwriting an existing 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 "cp can destroy a file" tip on page 46.

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
Thu Mar 24 14:31:18 PST 2005
$ who >> whoson
$ cat whoson
Thu Mar 24 14:31:18 PST 2005
root       console      Mar 24 05:00(:0)
alex       pts/4        Mar 24 12:23(:0.0)
alex       pts/5        Mar 24 12:33(:0.0)
jenny      pts/7        Mar 23 08:45 (bravo.example.com)
```

Figure 5-11 Redirecting and appending output

EXCERPT FROM CHAPTER 6: THE vim EDITOR

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 155.

t/T The next two commands are used in the same manner as the Find command. 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 location of the cursor to the second occurrence of the letter **q** on the current line.

SEARCHING FOR A STRING

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 (see 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 159). 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.

The **N** and **n** keys repeat the last search without any need for 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 180) and "Ignore case in searches" (page 178) 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 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 178). Give the command **:set incsearch** to turn on incremental searching. Use **noincsearch** to turn it off. When you set the **compatible** parameter (page 148), 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 834.

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. See "Allow special characters in searches" (page 177) for more information.

^ 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 are **dim** or **dism** or **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.

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

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 an **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 . . .**

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 178 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" below).

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 165). (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 *replace-string*.

The *replace-string* replaces the text matched by the *search-string*. It should be followed by the delimiter character. You can omit the final delimiter when no option follows the *replace-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 *replace-string*. For example, an ampersand (**&**) in the *replace-string* represents the text that was matched by the *search-string*. A backslash

in the *replace-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**
:%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 *replace-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.

EXCERPT FROM CHAPTER 8: THE BOURNE AGAIN SHELL

USER-CREATED VARIABLES

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

```
$ person=alex
$ echo person
person
$ echo $person
alex
```

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 **alex**. 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 (**$**). 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
alex
$ echo "$person"
alex
$ 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="alex and jenny"
$ echo $person
alex and jenny

$ person=alex and jenny
bash: and: command not found
```

When you reference a variable that 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="alex   and   jenny"
$ echo $person
alex and jenny
$ echo "$person"
alex   and   jenny
```

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.

Pathname expansion in assignments

The first line in the following sequence of commands assigns the string **alex*** to the variable **memo**. The Bourne Again Shell does *not expand the string* because bash does not perform pathname expansion (page 127) when assigning 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=alex*
$ echo "$memo"
alex*
```

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
alex.report
alex.summary
$ echo $memo
alex.report alex.summary
```

Here the shell expands **$memo** to **alex***, expands **alex*** to **alex.report** and **alex.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. 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 481).

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, set the value to null (use **set person =** in tcsh):

```
$ person=
$ echo $person

$
```

You can remove a variable with the unset builtin. To remove the variable **person**, give the following command:

```
$ 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=jenny
$ echo $person
jenny
$ 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 "Listing variable attributes" on page 282 for an example (**readonly** and **declare –r** produce the same output).

EXCERPT FROM CHAPTER 12:
THE gawk PATTERN PROCESSING LANGUAGE

ASSOCIATIVE ARRAYS

An 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.

You assign a value to an element of an associative array just as you would assign a value to any other gawk variable. The syntax is

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.

You can use a special **for** structure with an associative array. The syntax is

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*.

The "Examples" section found later in this chapter contains programs that use associative arrays.

printf

You can use the **printf** command in place of **print** to control the format of the output that 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. Refer to "Examples" later in this chapter for examples of how to use **printf**.

Table 12-5 Numeric conversion

conv	Type of conversion
d	Decimal
e	Exponential notation

Table 12-5 Numeric conversion (continued)

f	Floating-point number
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 436 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.

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

```
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 all lines that contain the string **chevy**. When you do not specify an *action*, gawk assumes that 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 special shell characters, you must quote them. Always enclosing the program in single quotation marks is the easiest way of making sure that 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 that 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 lines it selects:

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

Next 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 830) or, in this case, the beginning of the first field:

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

Brackets surround a character-class definition (page 829). 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. A dollar sign followed by a number names a field. Within a regular expression a dollar sign forces a match at the end of a line or field (**5$**). 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 that 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 concluding 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** or **––file** option followed by the name of the file containing the gawk program.

BEGIN The following gawk program, 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 the 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 following examples, the whitespace in the headers is composed of single TABs, so that 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 902) 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.

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 that you can read from and write to the coprocess.

to_upper When used alone the tr utility (page 804) 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 your 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 882) or *fully qualified domain name* (page 876) 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 **cars** file from the server at www.sobell.com; the author has set up this file for you to experiment with. On www.sobell.com the file is located at **/CMDREF1/code/chapter_12/cars**. 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.sobell.com 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.sobell.com/80"

       # use coprocess to send GET request to remote server
       print "GET /CMDREF1/code/chapter_12/cars" |& server

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

$ gawk -f g7
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

ERROR MESSAGES

The following examples show some of the more common causes of gawk error messages (and nonmessages). These examples are run under bash. When you use gawk with tcsh, the error messages from the shell will be different.

The first example leaves the single quotation marks off the command line, so the shell interprets **$3** and **$1** as shell variables. Also, because there are no single quotation marks, the shell passes gawk four arguments instead of two.

```
$ gawk {print $3, $1} cars
gawk: cmd. line:2: (END OF FILE)
gawk: cmd. line:2: syntax error
```

The next command line includes a typo (**prinnt**) that gawk does not catch. Instead of issuing an error message, gawk simply does not do anything useful.

```
$ gawk '$3 >= 83 {prinnt $1}' cars
```

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; 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

Accepts the common options described on page 587.

––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 689) for more information. Normally bzip2 does not overwrite a file; you must use **––force** to overwrite a file during compression or decompression.

Notes

The bzip2 home page is sources.redhat.com/bzip2.

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

Use the **––bzip2** modifier with tar (page 788) to compress archive files with bzip2.

bzcat *file-list* Works like cat except that 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
```

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 it can copy one or more files to a directory (second format). With the **––recursive** option, cp can copy directories.

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 of the copied files the same simple filename as its *source-file*.

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

Options

Accepts the common options described on page 587.

––archive **–a** Attempts to preserve as many attributes of *source-file* as possible. Same as **–dpPR**.

––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.

–d Copies symbolic links, not the files that links point to. Also preserves hard links in *destination-files* that exist between corresponding *source-files*. This option is equivalent to **––no-dereference** and **––preserve=links**.

––force **–f** When the *destination-file* exists and cannot be opened for writing, this option 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 has write permission to the directory containing the *destination-file*. See also **––backup**.

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

––dereference **–L** Copies the file that a symbolic link points to. See **––no-dereference**.

––preserve **–p** Creates a *destination-file* with the same owner, group, permissions, access date, and modification date as the *source-file*.

––no-dereference **–P** Copies symbolic links, not the files that the links point to. Without the **–R**, **–r**, or **––recursive** option, the default behavior is to dereference links (copy the files that links point to, not the links). With one of these options, cp does not dereference symbolic links (it copies the links, not the files that the links point to).

––parents Copies a relative pathname to a directory, creating directories as needed. (See "Examples.")

––preserve=links When recursively copying directories, attempts to preserve hard links in *destination-files* that exist between corresponding *source-files*.

––recursive **–R** *or* **–r**
Recursively copies directory hierarchies including ordinary files. The **––no-dereference** option is implied.

––update **–u** Copies only when the *destination-file* does not exist or when it is older than the *source-file*.

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

Notes

If the *destination-file* exists before you execute cp, 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 group becomes the group associated with the *destination-file*.

With the **–p** option, cp attempts to set the owner, group, permissions, access date, and modification date to match those of the *source-file*.

Unlike ln (page 702), the *destination-file* that cp creates is independent of its *source-file*.

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. Because of the **––preserve**

option, the copied files in **archives** have the same owner, group, permissions, access date, and modification date as the source files.

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

The next example copies **memo** from **/home/jenny** to the working directory:

```
$ cp /home/jenny/memo .
```

The next example 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/jenny/memo** and **/home/jenny/letter**.

```
$ cp memo letter /home/jenny
```

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

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

If Alex had used the **––backup** option in addition to **––force**, cp would have created a backup of **me** named **me~**. Refer to “Directory Access Permissions” on page 94 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

Accepts the common options described on page 587.

––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.

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 742).

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. This time 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 804) 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.

```
$ cat /etc/passwd
root:x:0:0:Root:/:/bin/sh
jenny:x:401:50:Jenny Chen:/home/jenny:/bin/zsh
alex:x:402:50:Alex Watson:/home/alex:/bin/bash
scott:x:504:500:Scott Adams:/home/scott:/bin/tcsh
hls:x:505:500:Helen Simpson:/home/hls:/bin/bash
```

```
$ cut -d: -f5 /etc/passwd
Root
Jenny Chen
Alex Watson
Scott Adams
Helen Simpson
```

tr

Replaces specified characters

tr [options] string1 [string2]

The tr utility reads standard input and, for each input character, maps it to an alternate character, deletes the character, or leaves the character alone. 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 **––delete** option, tr deletes the characters specified in *string1*. The option **––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 829). 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 then 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 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 a character class from Table V-28. You must specify a character class in *string1* unless you are performing case conversion (see "Examples" later in this section) or are using the **–d** and **–s** options together.

table V-28 Character classes

Class	Meaning
alnum	Letters and digits
alpha	Letters

table V-28 Character classes (continued)

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

––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 **––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*.

––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* and then reduces multiple sequential occurrences of characters in *string2*.

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

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 **––delete** and **––squeeze-repeats** options at the same time, tr deletes the characters in *string1* and then reduces multiple sequential occurrences of characters in *string2*.

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 --complement --squeeze-repeats '[:alpha:]' '\n' < draft1
```

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

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