## IN THIS CHAPTER

# 5 The Shell I

This chapter takes a close look at the shell and explains how to use some of its features. The chapter discusses command line syntax and how the shell processes a command line and initiates execution of a program. The chapter shows how to redirect input to and output from a command, construct pipes and filters on the command line, and run a command as a background task. 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, TC, and Z Shells. However, this chapter uses the Bourne Again Shell for most examples; if you use another shell, the exact format or wording of the shell output may differ from what you see here. Refer to Chapters 12 through 15 for shell-specific information and more on writing and executing shell scripts.

## The Command Line

The shell executes a program when you give it a command in response to its prompt. For example, when you give the ls command, the shell executes the utility program named ls. You can cause the shell to execute other types of programs—such as shell scripts, application programs, and programs you have written—in the same way. The line that contains the command, including any arguments, is called the *command line*. In this book the term *command* refers to the characters you type on the command line as well, as to the program that action invokes.

# Syntax

Command line syntax dictates the ordering and separation of the elements on a command line. When you press the RETURN key after entering a command, the shell scans the command line for proper syntax. The syntax for a basic command line is

*command [arg1] [arg2] ... [argn]* RETURN

One or more SPACEs must appear between elements on the command line. The *command* is the command name, *arg1* through *argn* are arguments, and RETURN is the keystroke that terminates all command lines. The arguments in the command line syntax are enclosed in brackets to show that they are optional. Not all commands require arguments: Some commands do not allow arguments; other commands allow a variable number of arguments; and others require a specific number of arguments. Options, a special kind of argument, are usually preceded by a hyphen (also called a dash or minus sign: –). (No *smiley* [page 1492] intended.)

## Command Name

Some useful GNU/Linux command lines consist of only the name of the command without any arguments. For example, ls by itself lists the contents of the working directory. Most commands accept one or more arguments. Commands that require arguments typically give a short error message, called a *usage message*, when you use them without arguments, with incorrect arguments, or with the wrong number of arguments.

## Arguments

On the command line each sequence of nonblank characters is called a *token*, or *word*. An *argument* is a token, such as a filename, string of text, number, or other object that a command acts on. For example, the argument to a vi or emacs command is the name of the file you want to edit.

The following command line shows cp copying the file named **temp** to **tempcopy**:

```
$ cp temp tempcopy
```

Arguments are numbered starting with the command itself as argument zero. In this example **cp** is argument zero, **temp** is argument one, and **tempcopy** is argument two. The cp utility requires two arguments on the command line. (The utility can take more but not fewer: see Part III.) Argument one is the name of an existing file, and argument two is the name of the file that cp is creating or overwriting. Here the arguments are not optional; both arguments must be present for the command to

work. When you do not supply the right number or kind of arguments, cp displays a usage message. Try typing **cp** and then pressing RETURN.

### *Options*

An *option* is an argument that modifies the effects of a command. You can frequently specify more than one option, modifying the command in several different ways. Options are specific to and interpreted by the program that the command line calls.

By convention, options are separate arguments that follow the name of the command. Most utilities require you to prefix options with a hyphen. However, this requirement is specific to the utility and not to the shell. GNU program options are frequently preceded by two hyphens in a row, with **––help** generating a (sometimes extensive) usage message.

Figure 5-1 first shows what happens when you give an ls command without any options. By default, ls lists the contents of the working directory in alphabetical order, vertically sorted in columns. Next, you see that the **–r** (reverse order; because this is a GNU utility, you can also use **––reverse**) option causes the ls utility to display the list of files in reverse alphabetical order, still sorted in columns. The **–x** option causes ls to display the list of files in horizontally sorted rows.

When you need to use several options, you can usually group multiple single-letter options into one argument that starts with a single hyphen; do not put SPACEs between the options. You cannot combine options that are preceded by two hyphens this way. Specific rules for combining options depend on the program you are running. Figure 5-1 shows both the **–r** and **–x** options with the ls utility. Together these options generate a list of filenames in horizontally sorted columns, in reverse alphabetical order. Most utilities allow you to list options in any order; **ls –xr** produces the same results as **ls –rx**. The command **ls –x –r** also generates the same list. For more information, refer to "Option Processing" on page 771.

```
$ ls
alex  house  mark   office    personal  test
hold  jenny  names  oldstuff  temp
$ ls -r
test  personal  office  mark  house  alex
temp  oldstuff  names  jenny  hold
$ ls -x
alex    hold      house     jenny  mark  names
office  oldstuff  personal  temp   test
$ ls -rx
test  temp   personal  oldstuff  office  names
mark  jenny  house     hold      alex
```

**Figure 5-1** Using options

### The Human Readable Option || caution

Most utilities that report on file sizes tell you the size of a file in bytes. That is all right when you are dealing with smaller files, but the numbers can get difficult to read when you are working with file sizes that are measured in megabytes or gigabytes. Give the command **df**, which reports on the space available on your system (and possibly on other systems). Look at the `Used` and `Avail` columns. Now give the command **df –h** or **df ––human-readable**. See the difference? Many utilities that report on file sizes, including **ls** with the **–l** option, have this option.

## Processing the Command Line

As you enter a command line, the Linux tty device driver (part of the Linux operating system kernel) examines each character to see whether it must take immediate action. When you press CONTROL-H (to erase a character) or CONTROL-U (to kill a line), the device driver immediately adjusts the command line as required; the shell never sees the character you erased or the line you killed. Often a similar adjustment occurs when you press CONTROL-W (to erase a word). When the character does not require immediate action, the device driver stores the character in a buffer and waits until it receives additional characters. When you press RETURN, the device driver passes the command line to the shell for processing.

When it processes a command line, the shell looks at the line as a whole and *parses* (breaks) it into its component parts (Figure 5-2). Next, the shell looks for the name of the command. Usually[1] the name of the command is the first thing on the command line after the prompt (argument zero), so the shell takes the first characters on the command line, up to the first blank (TAB or SPACE), and looks for a command with that name. The command name (the first token) can be specified on the command line either as a simple filename or as a pathname. For example, you can call the **ls** command in either of the following ways:

```
$ ls
```

*or*

```
$ /bin/ls
```

1. The shell does not require that the name of the program appear as the first argument on the command line. You *can* structure a command line as follows:

```
$ >bb <aa cat
```

When the shell sees the redirect symbols (page 145), it recognizes and processes them and their arguments before finding the name of the program that the command line is calling. This is a properly structured, although uncommon, command line.

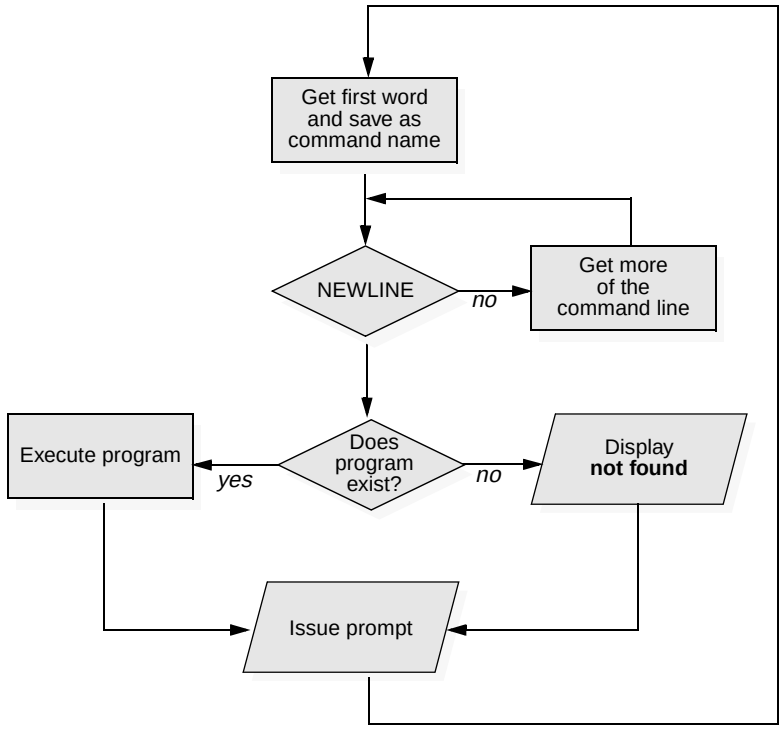


**Figure 5-2** Processing the command line

When you give an absolute pathname on the command line or a relative pathname that is not a simple filename (that is, any pathname that includes at least one slash), the shell looks in the specified directory (**/bin** in this case) for a file that has the name **ls** and that you have permission to execute. When you give a simple filename, the shell searches through a list of directories for a filename that matches the name that you specified and that you have execute permission for. The shell does not look through all directories but rather only the ones specified by the *shell variable* named **PATH**. Refer to page 577 bash zsh or page 708 tcsh for more information on **PATH**. Also refer to the discussion of which and whereis on page 78.

When it cannot find the executable file,[2] the Bourne Again Shell displays a message such as the following:

2. One reason the shell may not be able to find the executable file is that it is not in a directory in your **PATH**. Under bash the following command adds the working directory (**.**) to your **PATH** temporarily:

```
$ PATH=$PATH:.
```

For reasons of security, you may not want to add the working directory to your **PATH** permanently; see the tip "PATH and Security" on page 578.

```
$ abc
bash: abc: command not found
```

When the shell [bash tcsh zsh][3] finds the program but cannot execute it (you do not have execute access to the file that contains the program), you see a message similar to

```
$ def
bash: ./def: Permission denied.
```

## Executing the Command Line

If it finds an executable file with the same name as the command, the shell starts a new process. A *process* is the execution of a program (page 560). The shell makes each command line argument, including options and the name of the command, available to the called program. While the command is executing, the shell waits, inactive, for the process to finish. The shell is in a state called *sleep*. When the program finishes execution, the shell returns to an active state (wakes up), issues a prompt, and waits for another command.

Because the shell does not process command line arguments but only hands them to the called program, the shell has no way of knowing whether a particular option or other argument is valid for a given program. Any error or usage messages about options or arguments come from the program itself. Some utilities ignore bad options.

# Standard Input and Standard Output

The *standard output* is a place that a program can send information, such as text. The command (program) never "knows" where the information it sends to standard output is going (Figure 5-3). The information can go to a printer, an ordinary file, or your screen. The following sections show that, by default, the shell directs standard output from a command to the screen[4] and describe how you can cause the shell to redirect this output to another file. Standard input is a place that a program gets information from. As with standard output, the command never "knows" where the information came from. The following sections also explain how to redirect *standard input* to a command so that it comes from an ordinary file instead of from the keyboard (the default).

3. Refer to "Shell Specifier" on page 26 for an explanation of the [bash tcsh zsh] symbol.

4. The term *screen* is used throughout to mean screen, terminal emulator window, and workstation: *Screen* refers to the device that you see the prompt and messages displayed on.

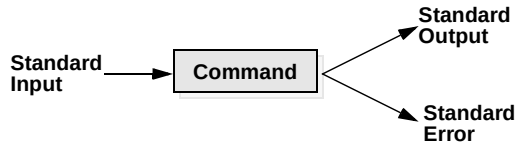


**Figure 5-3** The command does not know where standard input comes from or where standard output and standard error go.

In addition to standard input and standard output, a running program normally has a place to send error messages: *standard error*. Refer to pages 552, 691, and 781 for more information on handling standard error under the various shells.

### chsh: Changes Your Login Shell || tip

The person who sets up your account determines which shell you will use when you first log in on the system or when you open a terminal emulator window in a GUI environment. You can run any shell you like once you are logged in. Enter the name of the shell you want to use (bash, tcsh, or zsh) and press RETURN; the next prompt will be that of the new shell. Experiment with the shell as you like and give an exit command to return to your previous shell. Because shells you call in this manner are nested (one runs on top of the other), you will be able to log out only from your original shell. When you have nested several shells, keep giving exit commands until you are back to your original shell. Then you will be able to log out.

Use the chsh utility when you want to change your login shell permanently: Give the command chsh. Then, in response to the prompts, enter your password and the absolute pathname of the shell you want to use (/bin/bash, /bin/tcsh, or /bin/zsh).

## The Screen as a File

Chapter 4 introduced ordinary files, directory files, and hard and soft links. GNU/Linux has an additional type of file: a *device file*. A device file resides in the GNU/Linux file structure, usually in the **/dev** directory, and represents a peripheral device, such as a terminal emulator window, screen, printer, or disk drive.

The device name that the who utility displays after your login name is the filename of your screen/window. When who displays the device name **pts/4**, the pathname of your screen/window is **/dev/pts/4**. When you work with multiple windows, each one has its own device name. You can also use the tty utility to display the name of the screen that you give the command from. Although you would not normally have occasion, you can read from and write to this file as though it were a text file. Writing to it displays what you wrote on the screen; reading from it reads what you entered on the keyboard.

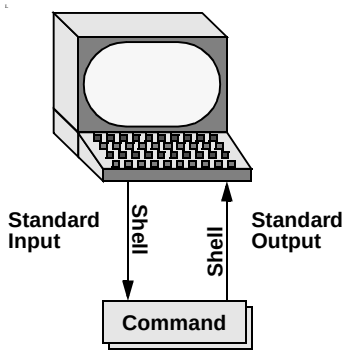


**Figure 5-4** By default, standard input comes from the keyboard, and standard output goes to the screen/window.

## The Screen/Keyboard as Standard Input and Standard Output

When you first log in, the shell directs standard output of your commands to the device file that represents your window/screen (Figure 5-4). Directing output in this manner causes it to appear on your screen. The shell also directs standard input to come from the same file, so that your commands receive anything you type on your keyboard as input.

The cat utility provides a good example of the way the screen/keyboard functions as standard input and standard output. When you use cat, it copies a file to standard output. Because the shell directs standard output to the screen, cat displays the file on the screen.

Up to this point cat has taken its input from the filename (argument) you specified on the command line. When you do not give cat an argument (that is, when you give the command cat followed immediately by a RETURN), cat takes input from standard input. The cat utility can now be described as a utility that, when called without an argument, copies standard input to standard output, one line at a time.

To see how cat works, type **cat** and press RETURN in response to the shell prompt. Nothing happens. Enter a line of text and press RETURN. The same line appears just under the one you entered. The cat utility is working. When you type a line of text using the keyboard, the shell associates that line with cat's standard input. Then cat copies your line of text to standard output, which the shell associated with the screen. This exchange is shown in Figure 5-5.

The cat utility keeps copying until you enter CONTROL-D on a line by itself. Pressing CONTROL-D sends an EOF (end of file) signal to cat to indicate that it has reached the end of standard input and that there is no more text for it to copy. When you enter CONTROL-D, cat finishes execution and returns control to the shell, which gives you a prompt.

```
$ cat
This is a line of text.
This is a line of text.
Cat keeps copying lines of text
Cat keeps copying lines of text
until you press CONTROL-D at the beginning
until you press CONTROL-D at the beginning
of a line.
of a line.
CONTROL-D
$
```

**Figure 5-5** The cat utility copies standard input to standard output.

# 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. As the previous section demonstrated, by default the shell associates standard input and standard output of a command with the window/screen and keyboard. You can cause the shell to redirect standard input and/or standard output of any command by associating the input or output with a command or file other than the device file representing the screen/keyboard. This section demonstrates how to redirect output to and input from 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.

In Figure 5-7 cat demonstrates output redirection. This figure contrasts with Figure 5-3 on page 143, where both standard input *and* standard output are associated with the screen and keyboard. In Figure 5-7 only the input comes from the screen. The redirect output symbol on the command line causes the shell to associate cat's standard output with the file specified on the command line: **sample.txt**.

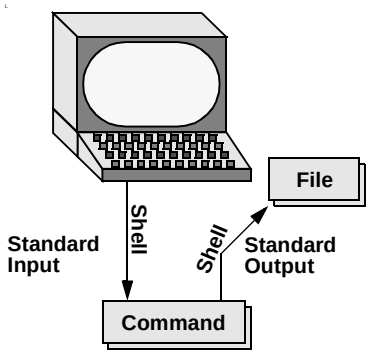


**Figure 5-6** Redirecting standard output

Now the file **sample.txt** contains the text you entered. You can use cat with an argument of **sample.txt** to display the 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 overwrites it and destroys its contents. For more information see the caution named "Redirecting Output Can Destroy a File II" on page 149.

Figure 5-7 shows that redirecting the output from cat is a handy way to make 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.
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

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:

```
$ cat stationery
2000 sheets letterhead ordered:    10/7/02
$ cat tape
1 box masking tape ordered:        10/14/02
5 boxes filament tape ordered:     10/28/02
$ cat pens
12 doz. black pens ordered:        10/4/02
$ cat stationery tape pens > supply_orders
$ cat supply_orders
2000 sheets letterhead ordered:    10/7/02
1 box masking tape ordered:        10/14/02
5 boxes filament tape ordered:     10/28/02
12 doz. black pens ordered:        10/4/02
$
```

**Figure 5-8** Using cat to catenate 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, you can redirect standard input. The *redirect input symbol* (<) instructs the shell to redirect a command's input from the specified file instead of 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, so no CONTROL-D is necessary.

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

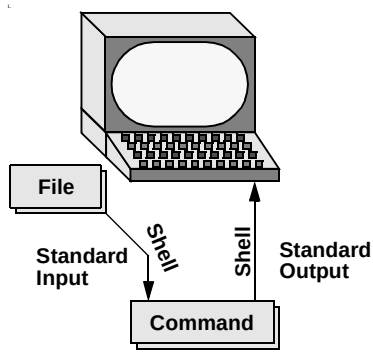


**Figure 5-9** Redirecting standard input

of a class of GNU/Linux utilities that function in this manner. Some of the other members of this class of utilities are lp, 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 the operating system, that functions in this manner.

```
$ cat < supply_orders
2000 sheets letterhead ordered:    10/7/02
1 box masking tape ordered:        10/14/02
5 boxes filament tape ordered:     10/28/02
12 doz. black pens ordered:        10/4/02
```

**Figure 5-10** cat with its input redirected

The shell provides a feature called **noclobber** (page 554) that stops you from inadvertently overwriting an existing file using redirection. When you enable this feature by setting the **noclobber** variable and you attempt to redirect output to an existing file, the shell presents an error message and does not execute the command. If the preceding examples result in one of the following messages, the **noclobber** feature is in effect. The following examples set **noclobber**, attempt to redirect the output from echo into an existing file, and then unset **noclobber** in each of the major shells:

**bash**

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

### 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. If you give the preceding command, 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 third, temporary file, and then use mv to rename the third file as you desire:

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

What happens with the typo 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 64) to the file a.output. Instead, the user enters the filename as a output, omitting the period and leaving 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 takes 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.

**tcsh**

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

**zsh**

```
zsh % set -o noclobber
zsh % echo "hi there" > tmp
zsh: file exists: tmp
zsh % set +o noclobber
```

## Appending Standard Output to a File

The *append output symbol* (>>) causes the shell to add new information to the end of a file, leaving intact any information that was there. 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 on to the end of (catenated with) the **orange** file. The final cat shows the result.

### Do Not Trust noclobber || caution

This technique is simpler to use than the two-step procedure just described, 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 cannot stop you from overwriting an existing file using cp or mv. These utilities include the –i (interactive) option that protects you from this type of mistake by verifying your intentions when you try to overwrite a file. For more information see the Tip titled “cp Can Destroy a File” on page 62.

Figure 5-11 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.

## /dev/null: Data Sink

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**. The output disappears 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 alex     pubs        25315 Oct 24 10:55 messages
$ cat /dev/null > messages
$ ls -l messages
-rw-r--r--   1 alex     pubs            0 Oct 24 11:02 messages
```

```
$ date >whoson
$ cat whoson
Thu Mar 27 14:31:18 PST 2003
$ who >>whoson
$ cat whoson
Thu Mar 27 14:31:18 PST 2003
root       console      Mar 27 05:00(:0)
alex       pts/4        Mar 27 12:23(:0.0)
alex       pts/5        Mar 27 12:33(:0.0)
jenny      pts/7        Mar 26 08:45 (bravo.tcorp.com)
```

**Figure 5-11** Redirecting and appending output

# Pipes

The shell uses a *pipe* to connect standard output of one command directly to standard input of another command. A pipe (sometimes referred to as a *pipeline*) has the same effect as redirecting standard output of one command to a file and then using that file as standard input to another command. A pipe does away with separate commands and the intermediate file. The symbol for a pipe is a vertical bar (|). The syntax of a command line using a pipe is

*command_a [arguments]* | *command_b [arguments]*

This command line uses a pipe to generate the same result as the following group of command lines:

*command_a [arguments]* > *temp*
*command_b [arguments]* < *temp*
*rm temp*

In the preceding sequence of commands, the first line redirects standard output from *command_a* to an intermediate file named *temp*. The second line redirects standard input for *command_b* to come from *temp*. The final line deletes *temp*. The command using a pipe is not only easier to type, it is generally more efficient because it does not create a temporary file.

You can use a pipe with a member of the class of GNU/Linux utilities that accepts input either from a file specified on the command line or from standard input. You can also use pipes with commands that accept input only from standard input. For example, the tr (translate) utility (page 1362 in Part III) takes its input from standard input only. In its simplest usage tr has the following format:

*tr string1 string2*

```
$ ls > temp
$ lpr temp
$ rm temp

or

$ ls | lpr
$
```

**Figure 5-12** A pipe

The tr utility accepts input from standard input and looks for characters that match one of the characters in *string1*. Finding a match, tr translates the matched character in *string1* to the corresponding character in *string2*. (The first character in *string1* translates into the first character in *string2*, and so forth.) In the following examples tr displays the contents of the **abstract** file with the letters **a**, **b**, and **c** translated into **A**, **B**, and **C**, respectively:

```
$ cat abstract | tr abc ABC
```

*or*

```
$ tr abc ABC < abstract
```

The tr utility does not change the contents of the original file.

The lpr (line printer) utility is among the utilities that accept input from either a file or standard input. When you type the name of a file following lpr on the command line, it places that file in the print queue. When you do not specify a filename on the command line, lpr takes input from standard input. This feature enables you to use a pipe to redirect input to lpr. The first set of commands in Figure 5-12 shows how you can use ls and lpr, with an intermediate file (**temp**), to send a list of the files in the working directory to the printer. If the **temp** file exists, the first command overwrites its contents. The second set of commands sends the same list (with the exception of **temp**) to the printer, using a pipe.

The commands in Figure 5-13 redirect the output from the who utility to **temp** and then display this file in sorted order. The sort utility (page 66) takes its input

```
$ who > temp
$ sort < temp
alex       pts/4        Mar 27 12:23
alex       pts/5        Mar 27 12:33
jenny      pts/7        Mar 26 08:45
root       console      Mar 27 05:00
$ rm temp
```

**Figure 5-13** Using a temporary file to store intermediate results

```
$ who | sort
alex       pts/4        Mar 27 12:23
alex       pts/5        Mar 27 12:33
jenny      pts/7        Mar 26 08:45
root       console      Mar 27 05:00
```

**Figure 5-14** A pipe doing the work of a temporary file

from the file specified on the command line or, when a file is not specified, from standard input and sends its output to standard output. The sort command line in Figure 5-13 takes its input from standard input, which is redirected (<) to come from **temp**. The output that sort sends to the screen lists the users in sorted (alphabetical) order.

Because sort can take its input from standard input or from a filename on the command line, you can omit the < symbol from Figure 5-13 to yield the same results.

Figure 5-14 achieves the same result without creating the **temp** file. Using a pipe, the shell redirects the output from who to the input of sort. The sort utility takes input from standard input because no filename follows it on the command line.

When a lot of people are using the system and you want information about only one of them, you can send the output from who to grep (page 64), using a pipe. The grep utility displays the line containing the string you specify—root in the following example:

```
$ who | grep 'root'
root       console      Mar 27 05:00
```

Another way of handling output that is too long to fit on the screen, such as a list of files in a crowded directory, is to use a pipe to send the output through less or more (both on page 54).

```
$ ls | less
```

The less utility displays text a screen at a time.[5] To view another screen, press the SPACE bar. To view one more line, press RETURN. Press **h** for help and **q** to quit.

## Filters

A *filter* is a command that processes an input stream of data to produce an output stream of data. A command line that includes a filter uses a pipe to connect standard

5. Some utilities change the format of their output when you redirect it. Compare the output of ls by itself and when you send it through a pipe to less.

output of one command to the filter's standard input. Another pipe connects the filter's standard output to standard input of another command. Not all utilities can be used as filters.

In the following example sort is a filter, taking standard input from standard output of who and using a pipe to redirect standard output to standard input of lpr. The command line sends the sorted output of who to the printer:

```
$ who | sort | lpr
```

This example demonstrates the power of the shell combined with the versatility of GNU/Linux utilities. The three utilities who, sort, and lpr were not specifically designed to work with each other, but they all use standard input and standard output in the conventional way. By using the shell to handle input and output, you can piece standard utilities together on the command line to achieve the results you want.

## tee: Sends Output in Two Directions

In a pipe the tee utility sends the output of a command to a file and also to standard output. The utility is aptly named: It takes a single input and sends the output in two directions. In Figure 5-15 the output of who is sent via a pipe to standard input of tee. The tee utility saves a copy of standard input in a file named **who.out** and also sends a copy to standard output. Standard output of tee goes via a pipe to standard input of grep, which displays lines containing the string root.

```
$ who | tee who.out | grep root
root       console      Mar 27 05:00
$ cat who.out
root       console      Mar 27 05:00
alex       pts/4        Mar 27 12:23
alex       pts/5        Mar 27 12:33
jenny      pts/7        Mar 26 08:45
```

**Figure 5-15** Using tee

# Running a Program in the Background

In all the examples so far in this book, commands were run in the *foreground*. When you run a command in the foreground, the shell waits for it to finish before

giving you another prompt and allowing you to continue. When you run a command in the *background*, you do not have to wait for the command to finish before you start running another command.

A *job* is a series of one or more commands connected by one or more pipes. You can have only one foreground job in a window or on a screen, but you can have many background jobs. By running more than one job at a time, you are using one of GNU/Linux's important features: multitasking. running a command in the background can be useful when the command will be running for a long time and does not need supervision. The window/screen is free so that you can use it for other work. Of course, when you are using a GUI, you can simply open another window to run another job.

To run a command in the background, type an ampersand (**&**) just before the RETURN that ends the command line. The shell assigns a small number to the job and displays this *job number* between brackets. Following the job number, the shell displays the *process identification* (PID) number—a bigger number assigned by the operating system. Each of these numbers identifies the command running in the background. Then the shell gives you another prompt so you can enter another command. When the background job finishes running, the shell displays a message giving both the job number and the command line used to run the command.

The following examples use the Bourne Again Shell. The TC and Z Shells produce almost identical results. The next example runs in the background and sends its output through a pipe to lpr, which sends it to the printer.

```
bash $ ls -l | lpr &
[1] 22092
bash $
```

The [1] following the command line indicates that the shell has assigned job number 1 to this job. The 22092 is the PID number of the first command in the job. (The TC Shell shows PID numbers for all commands in the job.) When this background job completes execution, you see the message

```
[1]+ Done          ls -l | lpr
```

You can stop a foreground job from running by pressing the suspend key, usually CONTROL-Z. The shell stops the process and disconnects standard input from the screen keyboard. You can put a job in the background and start it running by using the bg command, followed by a percent sign and the job number. You do not need to use the job number when you have only one stopped job.

Only the foreground job can take input from the keyboard. To connect the keyboard to the program running in the background, you must bring it into the foreground: Type **fg** without any arguments when only one job is in the background.

When more than one job is in the background, type **fg** (optional) followed by a percent sign and the job number of the job you want to bring into the foreground. The shell displays the command you used to start the job, and you can enter any input the program requires to continue:

```
bash $ fg %1
[1] promptme
```

Redirect the output of a job you run in the background to keep it from interfering with whatever you are doing on the screen. Refer to "Separating and Grouping Commands" on page 547 for more detail about background tasks.

The interrupt key (usually CONTROL-C) cannot abort a process you are running in the background; you must use kill (page 901) for this purpose. Follow **kill** on the command line with either the PID number of the process you want to abort or a percent sign (%) followed by the job number.

If you forget the PID number, you can use the ps (process status [page 561]) utility to display it. Using the TC Shell, the following example runs a **tail –f outfile** command (the **–f** option causes tail to watch **outfile** and display any new lines that are written to it) as a background job, uses ps to display the PID number of the process, and aborts the job with kill. So that it does not interfere with anything on the screen, the message saying that the job is terminated does not appear until you press RETURN after the RETURN that ends the kill command:

```
tcsh $ tail -f outfile &
[1] 22170
tcsh $ ps | grep tail
22170 pts/7    0:00 tail
tcsh $ kill 22170
tcsh $ RETURN
[1]    Terminated           tail -f outfile
tcsh $
```

If you forget the job number, you can use the jobs command to display a list of job numbers. The following example is similar to the previous one but uses the job number in place of the PID number to kill the job:

```
tcsh $ tail -f outfile &
[1] 3339
tcsh $ bigjob &
[2] 3340
tcsh $ jobs
[1]  - Running              tail -f outfile
[2]  + Running              bigjob
tcsh $ kill %1
tcsh $RETURN
[1]    Terminated           tail -f outfile
tcsh $
```

# Filename Generation/Pathname Expansion

When you give the shell abbreviated filenames that contain special characters, also called *metacharacters*, the shell can generate filenames that match the names of existing files. These special characters are also referred to as *wildcards* because they act as the jokers do in a deck of cards. When one of these special characters appears in an argument on the command line, the shell expands that argument in sorted order (refer to "LC_COLLATE" on page 1431) into a list of filenames and passes the list to the program that the command line calls. Filenames that contain these special characters are called *ambiguous file references* because they do not refer to any one specific file. The process that the shell performs on these filenames is called *pathname expansion*, or *globbing*.

Ambiguous file references refer to a group of files with similar names quickly, saving you the effort of typing the names individually, as well as a file whose name you do not remember in its entirety. If no filename matches the ambiguous file reference, the shell generally passes the unexpanded reference, special characters and all, to the command.

## The ? Special Character

The question mark (**?**) is a special character that causes the shell to generate filenames. The question mark matches any single character in the name of an existing file. The following command uses this special character in an argument to the lpr utility:

```
$ lpr memo?
```

The shell expands the **memo?** argument and generates a list of files in the working directory that have names composed of **memo** followed by any single character. The shell passes this list to lpr. The lpr utility never "knows" that the shell generated the filenames it was called with. If no filename matches the ambiguous file reference, the shell bash tcsh zsh passes the string itself (**memo?**) to lpr or, if it is set up to do so, displays an error message bash tcsh zsh.

The following example uses ls first to display the names of all of the files in the working directory and then to display the filenames that memo? matches:

```
$ ls
mem   memo12  memo9  memoalex  newmemo5
memo  memo5   memoa  memos
$ ls memo?
memo5  memo9  memoa  memos
```

The **memo?** ambiguous file reference does not match **mem**, **memo**, **memo12**, **memoalex**, or **newmemo5**. You can also use a question mark in the middle of an ambiguous file reference:

```
$ ls
7may4report  may4report     mayqreport  may_report
may14report  may4report.79  mayreport   may.report
$ ls may?report
may.report  may4report  may_report  mayqreport
```

To practice generating filenames, you can use echo and ls; echo displays the arguments that the shell passes to it:

```
$ echo may?report
may.report  may4report  may_report  mayqreport
```

The shell expands the ambiguous file reference into a list of all files in the working directory that match the string `may?report` and passes this list to echo, as though you had entered the list of filenames as arguments to echo. The echo utility responds by displaying the list of filenames. A question mark does not match a leading period (one that indicates an invisible filename). When you want to match filenames that begin with a period, you must explicitly include the period in the ambiguous file reference.

## The * Special Character

The asterisk (*) performs a function similar to that of the question mark but matches any number of characters, *including zero characters*, in a filename. The following example shows all the files in the working directory and then shows three commands that display all the filenames that begin with the string **memo**, end with the string **mo**, and contain the string **alx**:

```
$ ls
amemo   memo       memoalx.0620  memosally  user.memo
mem     memo.0612  memoalx.keep  sallymemo
memalx  memoa      memorandum    typescript
$ echo memo*
memo memo.0612 memoa memoalx.0620 memoalx.keep memorandum memosally
$ echo *mo
amemo memo sallymemo user.memo
$ echo *alx*
memalx memoalx.0620 memoalx.keep
```

The ambiguous file reference **memo*** does not match **amemo**, **mem**, **sallymemo**, or **user.memo**. As with the question mark, an asterisk does not match a leading period in a filename.

The **–a** option causes ls to display invisible filenames. The command **echo *** does not display **.** (the working directory), **..** (the parent of the working directory), **.aaa**, or **.profile**. The command **echo .*** displays only those four names:

```
$ ls
aaa memo.0612 memo.sally report sally.0612 saturday thurs
$ ls -a
.   aaa   memo.0612   .profile  sally.0612  thurs
..  .aaa  memo.sally  report    saturday

$ echo *
aaa memo.0612 memo.sally report sally.0612 saturday thurs
$ echo .*
. .. .aaa .profile
```

In the following example **.p*** does not match **memo.0612**, **private**, **reminder**, or **report**. Following that, the **ls .*** command causes ls to list **.private** and **.profile** in addition to the entire contents of the **.** directory (the working directory) and the **..** directory (the parent of the working directory). With the same argument, echo displays only the filenames from the working directory that begin with a dot (**.**):

```
$ ls -a
.       .private   memo.0612  reminder
..      .profile   private    report
$ echo .p*
.private .profile
$ ls .*
.private .profile

.:
memo.0612  private    reminder   report

..:
.
.
$ echo .*
.private .profile
```

When you establish conventions for naming files, you can take advantage of ambiguous file references. For example, when you end all text filenames with **.txt**, you can reference that group of files with ***.txt**. Following this convention, the next command sends all the text files in the working directory to the printer. The ampersand causes lpr to run in the background.

```
$ lpr *.txt &
```

## The [ ] Special Characters

A pair of brackets surrounding a list of characters causes the shell to match filenames containing the individual characters. Whereas memo? matches **memo** followed by any character, memo[17a] is more restrictive, matching only **memo1**, **memo7**, and **memoa**. The brackets define a *character class* that includes all the characters within the brackets. The shell expands an argument that includes a character-class definition, substituting each member of the character class, *one at a time*,

in place of the brackets and their contents. The shell passes the list of matching filenames to the program it is calling.

Each character-class definition can replace only a single character within a filename. The brackets and their contents are like a question mark that substitutes only the members of the character class.

The first of the following commands lists the names of all the files in the working directory that begin with `a`, `e`, `i`, `o`, or `u`. The second command displays the contents of the files named **page2.txt**, **page4.txt**, **page6.txt**, and **page8.txt**:

```
$ echo [aeiou]*
. . .
$ less page[2468].txt
. . .
```

A hyphen within brackets defines a range of characters within a character-class definition. For example, **[6–9]** represents **[6789]**, **[a–z]** represents all lowercase letters in English, and **[a–zA–Z]** represents all letters, upper- and lowercase, in English.

The following command lines show three ways to print the files named **part0**, **part1**, **part2**, **part3**, and **part5**. Each of the command lines causes the shell to call lpr with five filenames:

```
$ lpr part0 part1 part2 part3 part5

$ lpr part[01235]

$ lpr part[0-35]
```

The first command line explicitly specifies the five filenames. The second and third command lines use ambiguous file references, incorporating character-class definitions. The shell expands the argument on the second command line to include all files that have names beginning with **part** and ending with any of the characters in the character class. The character class is explicitly defined as `0`, `1`, `2`, `3`, and `5`. The third command line also uses a character-class definition but defines the character class to be all characters in the range `0`–`3` and `5`.

The following command line prints 39 files, **part0** through **part38**:

```
$ lpr part[0-9] part[12][0-9] part3[0-8]
```

The following two examples list the names of some of the files in the working directory. The first lists the files whose names start with **a** through **m**. The second lists files whose names end with **x**, **y**, or **z**:

```
$ echo [a-m]*
.
.
$ echo *[x-z]
.
.
```

### Optional

When an exclamation point (**!**) or a caret (^) immediately follows the opening bracket (**[**), the string enclosed by the brackets matches any character not between the brackets, so that **[^ab]*** matches any filename that does not begin with **a** or **b**. You can match a hyphen (**–**) or a closing bracket (**]**) by placing it immediately before the final closing bracket.

### The Shell Expands Ambiguous File References

|| tip

*The shell does the expansion* when it processes an ambiguous file reference, not the program that the shell runs. In the examples in this section, *the utilities* (ls, cat, echo, lpr) *never see the ambiguous file references.* The shell expands the ambiguous file references and passes the utility a list of ordinary filenames. In the previous examples echo shows this to be true because all it does is display its arguments and never displays the ambiguous file reference.

The following example demonstrates that the ls utility has no ability to interpret ambiguous file references. First, ls is called with an argument of `?old`. The shell expands `?old` into a matching filename, **hold**, and passes that name to ls. The second command is the same as the first, except the ? is quoted (refer to "Special Characters" on page 55), so the shell does not recognize it as a special character and passes it on to ls. The ls utility generates an error message saying that it cannot find a file named **?old** (because there is no file named **?old**):

```
$ ls ?old
hold
$ ls \?old
?old: No such file or directory
```

As with most utilities and programs, ls cannot interpret ambiguous file references; that work is left to the shell.

## Builtins

A *builtin* is a utility (also called a *command*) that is built into a shell. Each of the three major shells—the Bourne Again, TC, and Z—has its own set of builtins. When it runs a builtin, the shell does not fork a new process. Consequently, builtins run more quickly and can affect the environment of the current shell. Because builtins are used in the same way as utilities, you will not typically be aware of whether a utility is built into the shell or is a stand-alone utility.

The echo bash tcsh zsh utility is a shell builtin. The shell always executes a shell builtin before trying to find a command/utility with the same name. Refer to "which, whereis, and Builtin Commands" on page 79 for information on using which and whereis to locate echo and other builtin commands. See page 670 for bash builtins, page 724 for tcsh builtins, and page 767 for zsh builtins.

To get a complete list of bash builtins, give the command **help | less** from a bash shell prompt. You can also give the command **info bash** to display the top level info page on bash.[6] Next, give the command **m builtin** to display a menu of bash builtin commands. Use the DOWN ARROW key to move the cursor to the line that lists the builtin you are interested in. Press **m** RETURN to display the corresponding info page. Alternatively, after typing **info bash**, give the command **/builtin**, which searches the bash documentation for the string `builtin`. The cursor will rest on the word `Builtin` in a menu; press **m** RETURN to display a menu on builtins. For tcsh, give the command **man tcsh** to display the tcsh man page, and then search for the second occurrence of `Builtin commands` with the following two commands: **/Builtin commands** (search for the string) and **n** (search for the next occurrence of the string). Give the command **man zshbuiltins** for a list of zsh builtins.

## Chapter Summary

The shell is the GNU/Linux command interpreter. It scans the command line for proper syntax, picking out the command name and any arguments. The first argument is referred to as argument one, the second as argument two, and so on. The name of the command itself is sometimes referred to as argument zero. Many programs use options to modify the effects of a command. Most GNU/Linux utilities identify an option by its leading one or two hyphens.

When you give it a command, the shell tries to find an executable program with the same name as the command. When it does, the shell executes the program. When it does not, the shell tells you that it cannot find or execute the program. If the command is expressed as a simple filename, the shell searches the directories given in the variable **PATH** in an attempt to locate the command.

When it executes a command, the shell assigns a file to the command's standard input and standard output. By default, the shell causes a command's standard input to come from the keyboard and standard output to go to the screen. You can instruct the shell to redirect a command's standard input from or standard output to any reasonable file or device. You can also connect standard output of one command to standard input of another using a pipe. A filter is a command that reads its standard input from standard output of one command and writes its standard output to standard input of another command.

6. Because bash was written by GNU, the info page has better information than does the man page.

When a command runs in the foreground, the shell waits for it to finish before it gives you another prompt and allows you to continue. When you put an ampersand (**&**) at the end of a command line, the shell executes the command in the background and gives you another prompt immediately. Put a command in the background when you think it may not execute quickly and you want to enter other commands at the shell prompt. The jobs builtin displays a list of background jobs and includes the job number of each.

The shell interprets shell special characters on a command line for filename generation: A question mark represents any single character, and an asterisk represents zero or more characters. A single character may also be represented by a character class: a list of characters within brackets. A reference that uses special characters (wildcards) to abbreviate a list of one or more filenames is called an ambiguous file reference.

## Utilities Introduced in This Chapter

| | |
|---|---|
| tr | Maps one string of characters into another (page 151) |
| tee | Sends standard input to both a file and to standard output (page 154) |
| bg | Moves a process to the background (page 155) |
| fg | Moves a process to the foreground (page 155) |
| jobs | Displays a list of currently running jobs (page 156) |

## Exercises

1. What does the shell ordinarily do while a command is executing? What should you do if you do not want to wait for a command to finish before running another command?

2. Using sort as a filter, rewrite the following sequence of commands:

```
$ sort list > temp
$ lpr temp
$ rm temp
```

3. What is a PID number? Why are they useful when you run processes in the background?

4. Assume that the following files are in the working directory:

```
$ ls
intro      notesb     ref2       section1   section3   section4b
notesa     ref1       ref3       section2   section4a  sentrev
```

Give commands for each of the following, using wildcards to express filenames with as few characters as possible.

a. List all files that begin with `section`.

b. List the **section1**, **section2**, and **section3** files only.

c. List the **intro** file only.

d. List the **section1**, **section3**, **ref1**, and **ref3** files.

5. Refer to the documentation of utilities in Part III or the man pages to determine what commands will

a. Output the number of lines in the standard input that contain the *word* `a` or `A`.

b. Output only the names of the files in the working directory that contain the pattern `$(`.

c. List the files in the working directory in their reverse alphabetical order.

d. Send a list of files in the working directory to the printer, sorted by size.

6. Give a command to

a. Redirect the standard output from a sort command into a file named **phone_list**. Assume that the input file is named **numbers**.

b. Translate all occurrences of characters **[** and **{** to the character **(**, and all occurrences of the characters **]** and **}** to the character **)** in the file **permdemos.c**. (*Hint:* Refer to tr on page 1362 in Part III.)

c. Create a file named **book** that contains the contents of two others files: **part1** and **part2**.

7. The lpr and sort utilities accept input from either a file named on the command line or from standard input.

a. Name two other utilities that function in a similar manner.

b. Name a utility that accepts its input only from standard input.

8. Give an example of a command that uses grep

a. With both input and output redirected.

b. With only input redirected.

c. With only output redirected.

d. Within a pipe.

In which of the preceding is grep used as a filter?

9. Explain the following error message. What filenames would a subsequent ls display?

```
$ ls
abc  abd  abe  abf  abg  abh
$ rm abc ab*
rm: cannot remove 'abc': No such file or directory
```

## Advanced Exercises

10. When you use the redirect output symbol (>) with a command, the shell creates the output file immediately, before the command is executed. Demonstrate that this is true.
11. In experimenting with shell variables, Alex accidentally deletes his **PATH** variable. He decides that he does not need the **PATH** variable. Discuss some of the problems he may soon encounter, and explain the reasons for these problems. How could he *easily* return **PATH** to its original value?
12. Assume that your permissions allow you to write to a file but not to delete it.

    a. Give a command to empty the file without invoking an editor.

    b. Explain how you might have permission to modify a file that you cannot delete.

13. If you accidentally create a filename with a nonprinting character, such as a CONTROL character in it, how can you rename the file?
14. Why can the **noclobber** variable *not* protect you from overwriting an existing file with cp or mv?
15. Why do command names and filenames usually not have embedded SPACEs? How would you create a filename containing a SPACE? How would you remove it? (This is a thought exercise, not a recommended practice. If you want to experiment, create and work in a directory with nothing but your experimental file in it.)
16. Create a file named **answers** and give the following command:

```
$ > answers.0102 < answers cat
```

Explain what the command does and why. What is a more conventional way of expressing this command?