1. Introduction

It's no secret that the Unix operating system has emerged as a standard operating system. For programmers who have been using Unix for many years now, this came as no surprise: The Unix system provides an elegant and efficient environment for program development. After all, this is what Dennis Ritchie and Ken Thompson strived for when they developed Unix at Bell Laboratories in the late 1960s.

One of the strongest features of the Unix system is its wide collection of programs. More than 200 basic commands are distributed with the standard operating system. These commands (also known as *tools*) do everything from counting the number of lines in a file, to sending electronic mail, to displaying a calendar for any desired year.

But the real strength of the Unix system comes not entirely from this large collection of commands but also from the elegance and ease with which these commands can be combined to perform far more sophisticated functions.

To further this end, and also to provide a consistent buffer between the user and the guts of the Unix system (the *kernel*), the shell was developed. The *shell* is simply a program that reads in the commands you type and converts them into a form more readily understood by the Unix system. It also includes some fundamental programming constructs that let you make decisions, loop, and store values in variables.

The standard shell distributed with Unix and Linux systems derives from AT&T's distribution, which evolved from a version originally written by Stephen Bourne at Bell Labs. Since then, the IEEE created standards based on the Bourne shell and the other more recent shells. The current version of this standard as of this revision is the Shell and Utilities volume of IEEE Std 1003.1-2001, also known as the POSIX standard. This shell is what we propose to teach you about in this book.

The examples in this book were tested on both SunOS 5.7 running on a Sparcstation Ultra-30 and on Silicon Graphics IRIX 6.5 running on an Octane; some examples were also run on Red Hat Linux 7.1 and Cygwin. All examples, except some Bash examples in Chapter 15, were run using the Korn shell, although many were also run with Bash.

Many Unix systems are still around that have Bourne shell derivatives and utilities not compliant with the POSIX standard. We'll try to note this throughout the text wherever possible; however, there are so many different versions of Unix from so many different vendors that it's simply not possible to mention every difference. If you do have an older Unix system that doesn't supply a POSIX-compliant shell, there's still hope. We'll list resources at the end of this book where you can obtain free copies of three different POSIX-compliant shells.

Because the shell offers an interpreted programming language, programs can be written, modified, and debugged quickly and easily. We turn to the shell as our first choice of programming language. After you become adept at programming in the shell, you too may turn to it first.

This book assumes that you are familiar with the fundamentals of the Unix system; that is, that you know how to log in; how to create files, edit them, and remove them; and how to work with directories. But in case you haven't used the Unix system for a while, we'll examine the basics in Chapter 2, "A Quick Review of the Basics." Besides the basic file commands, filename substitution, I/O redirection, and pipes are also reviewed in Chapter 2.

Chapter 3, "What Is the Shell?," reveals what the shell really is. You'll learn about what happens every time you log in to the system, how the shell program gets started, how it parses the command line, and how it executes other programs for you. A key point made in Chapter 3 is that the shell is just a program; nothing more, nothing less.

Chapter 4, "Tools of the Trade," provides tutorials on tools useful in writing shell programs. Covered in this chapter are `cut`, `paste`, `sed`, `grep`, `sort`, `tr`, and `uniq`. Admittedly, the selection is subjective, but it does set the stage for programs that we'll develop throughout the remainder of the book. Also in Chapter 4 is a detailed discussion of regular expressions, which are used by many Unix commands such as `sed`, `grep`, and `ed`.

Chapters 5 through 10 teach you how to put the shell to work for writing programs. You'll learn how to write your own commands; use variables; write programs that accept arguments; make decisions; use the shell's `for`, `while`, and `until` looping commands; and use the `read` command to read data from the terminal or from a file. Chapter 6, "Can I Quote You on That?," is devoted entirely to a discussion on one of the most intriguing (and often confusing) aspects of the shell: the way it interprets quotes.

By this point in the book, all the basic programming constructs in the shell will have been covered, and you will be able to write shell programs to solve your particular problems.

Chapter 11, "Your Environment," covers a topic of great importance for a real understanding of the way the shell operates: the *environment*. You'll learn about local and exported variables; subshells; special shell variables such as `HOME`, `PATH`, and `CDPATH`; and how to set up your `.profile` file.

Chapter 12, "More on Parameters," and Chapter 13, "Loose Ends," tie up some loose ends, and Chapter 14, "Rolo Revisited," presents a final version of a phone directory program called `rolo` that is developed throughout the book.

Chapter 15, "Interactive and Nonstandard Shell Features," discusses features of the shell that either are not formally part of the IEEE POSIX standard shell (but are available in most Unix and Linux shells) or are mainly used interactively instead of in programs.

Appendix A, "Shell Summary," summarizes the features of the IEEE POSIX standard shell.

Appendix B, "For More Information," lists references and resources, including the Web sites where different shells can be downloaded.

The philosophy this book uses is to teach by example. Properly chosen examples do a far superior job at illustrating how a particular feature is used than ten times as many words. The old "A picture is worth…" adage seems to apply just as well to examples. You are encouraged to type in each example and test it on your system, for only by doing can you become adept at shell programming. You also should not be afraid to experiment. Try changing commands in the program examples to see the effect, or add different options or features to make the programs more useful or robust.

At the end of most chapters you will find exercises. These can be used as assignments in a classroom environment or by yourself to test your progress.

This book teaches the IEEE POSIX standard shell. Incompatibilities with earlier Bourne shell versions are noted in the text, and these tend to be minor.

Acknowledgments from the first edition of this book: We'd like to thank Tony Iannino and Dick Fritz for editing the manuscript. We'd also like to thank Juliann Colvin for performing her usual wonders copy editing this book. Finally, we'd like to thank Teri Zak, our acquisitions editor, and posthumously Maureen Connelly, our production editor. These two were not only the best at what they did, but they also made working with them a real pleasure.

For the first revised edition of this book, we'd like to acknowledge the contributions made by Steven Levy and Ann Baker, and we'd like to also thank the following people from Sams: Phil Kennedy, Wendy Ford, and Scott Arant.

For the second revised edition of this book, we'd like to thank Kathryn Purdum, our acquisitions editor, Charlotte Clapp, our project editor, and Geneil Breeze, our copy editor.

2. A Quick Review of the Basics

In This Chapter

This chapter provides a review of the Unix system, including the file system, basic commands, filename substitution, I/O redirection, and pipes.

Some Basic Commands

Displaying the Date and Time: The `date` Command

The `date` command tells the system to print the date and time:

```
$ date
Sat Jul 20 14:42:56 EDT 2002
$
```

`date` prints the day of the week, month, day, time (24-hour clock, the system's time zone), and year. Throughout this book, whenever we use **`boldface type like this`**, it's to indicate what you, the user, types in. `Normal face type like this` is used to indicate what the Unix system prints. *Italic type* is used for comments in interactive sequences.

Every Unix command is ended with the pressing of the Enter key. Enter says that you are finished typing things in and are ready for the Unix system to do its thing.

Finding Out Who's Logged In: The `who` Command

The `who` command can be used to get information about all users currently logged in to the system:

Click here to view code image

```
$ who
pat     tty29    Jul 19 14:40
ruth    tty37    Jul 19 10:54
steve   tty25    Jul 19 15:52
$
```

Here, three users are logged in: `pat`, `ruth`, and `steve`. Along with each user id, the *tty* number of that user and the day and time that user logged in is listed. The tty number is a unique identification number the Unix system gives to each terminal or network device that a user has logged into.

The `who` command also can be used to get information about yourself:

Click here to view code image

```
$ who am i
pat       tty29     Jul 19 14:40
$
```

`who` and `who am i` are actually the same command: `who`. In the latter case, the `am` and `i` are *arguments* to the `who` command.

Echoing Characters: The `echo` Command

The `echo` command prints (or *echoes*) at the terminal whatever else you happen to type on the line (there are some exceptions to this that you'll learn about later):

Click here to view code image

```
$ echo this is a test
this is a test
$ echo why not print out a longer line with echo?
why not print out a longer line with echo?
$ echo
                                        A blank line is displayed
$ echo one        two    three             four   five
one two three four five
$
```

You will notice from the preceding example that `echo` squeezes out extra blanks between words. That's because on a Unix system, the words are important; the blanks are merely there to separate the words. Generally, the Unix system ignores extra blanks (you'll learn more about this in the next chapter).

Working with Files

The Unix system recognizes only three basic types of files: *ordinary* files, *directory* files, and *special* files. An ordinary file is just that: any file on the system that contains data, text, program instructions, or just about anything else. Directories are described later in this chapter. As its name implies, a special file has a special meaning to the Unix system and is typically associated with some form of I/O.

A filename can be composed of just about any character directly available from the keyboard (and even some that aren't) provided that the total number of characters contained in the name is not greater than 255. If more than 255 characters are specified, the Unix system simply ignores the extra characters.[1]

[1] *Modern Unix and Microsoft Windows systems support long filenames; however, some older Unix and Windows systems only allow much shorter filenames.*

The Unix system provides many tools that make working with files easy. Here we'll review many basic file manipulation commands.

Listing Files: The `ls` Command

To see what files you have stored in your directory, you can type the `ls` command:

```
$ ls
READ_ME
names
tmp
$
```

This output indicates that three files called `READ_ME`, `names`, and `tmp` are contained in the current directory. (Note that the output of `ls` may vary from system to system. For example, on many Unix systems `ls` produces multicolumn output when sending its output to a terminal; on others, different colors may be used

for different types of files. You can always force single-column output with the `-l` option.)

Displaying the Contents of a File: The `cat` Command

You can examine the *contents* of a file by using the `cat` command. The argument to `cat` is the name of the file whose contents you want to examine.

```
$ cat names
Susan
Jeff
Henry
Allan
Ken
$
```

Counting the Number of Words in a File: The `wc` Command

With the `wc` command, you can get a count of the total number of lines, words, and characters of information contained in a file. Once again, the name of the file is needed as the argument to this command:


```
$ wc names
          5       5       27 names
$
```

The `wc` command lists three numbers followed by the filename. The first number represents the number of lines contained in the file (5), the second the number of words contained in the file (in this case also 5), and the third the number of characters contained in the file (27).

Command Options

Most Unix commands allow the specification of *options* at the time a command is executed. These options generally follow the same format:

-letter

That is, a command option is a minus sign followed immediately by a single letter. For example, to count just the number of lines contained in a file, the option `-l` (that's the letter l) is given to the `wc` command:

```
$ wc -l names
          5 names
$
```

To count just the number of characters in a file, the `-c` option is specified:

```
$ wc -c names
          27 names
$
```

Finally, the `-w` option can be used to count the number of words contained in the file:

```
$ wc -w names
          5 names
$
```

Some commands require that the options be listed before the filename arguments. For example, `sort names -r` is acceptable, whereas `wc names -l` is not. Let's generalize by saying that command options should *precede* filenames on the command line.

Making a Copy of a File: The `cp` Command

To make a copy of a file, the `cp` command is used. The first argument to the command is the name of the file to be copied (known as the *source file*), and the second argument is the name of the file to place the copy into (known as the *destination file*). You can make a copy of the file `names` and call it `saved_names` as follows:

```
$ cp names saved_names
$
```

Execution of this command causes the file named `names` to be copied into a file named `saved_names`. As with many Unix commands, the fact that a command prompt was displayed after the `cp` command was typed indicates that the command executed successfully.

Renaming a File: The `mv` Command

A file can be renamed with the `mv` command. The arguments to the `mv` command follow the same format as the `cp` command. The first argument is the name of the file to be renamed, and the second argument is the new name. So, to change the name of the file `saved_names` to `hold_it`, for example, the following command would do the trick:

```
$ mv saved_names hold_it
$
```

When executing an `mv` or `cp` command, the Unix system does not care whether the file specified as the second argument already exists. If it does, the contents of the file will be lost.[2] For example, if a file called `old_names` exists, executing the command

[2] *Assuming that you have the proper permission to write to the file.*

```
cp names old_names
```

would copy the file `names` to `old_names`, destroying the previous contents of `old_names` in the process. Similarly, the command

```
mv names old_names
```

would rename `names` to `old_names`, even if the file `old_names` existed prior to execution of the command.

Removing a File: The `rm` Command

To remove a file from the system, you use the `rm` command. The argument to `rm` is simply the name of the file to be removed:

```
$ rm hold_it
$
```

You can remove more than one file at a time with the `rm` command by simply specifying all such files on the command line. For example, the following would remove the three files `wb`, `collect`, and `mon`:

```
$ rm wb collect mon
$
```

Working with Directories

Suppose that you had a set of files consisting of various memos, proposals, and letters. Further suppose that you had a set of files that were computer programs. It would seem logical to group this first set of files into a directory called `documents`, for example, and the latter set of files into a directory called `programs`. Figure 2.1 illustrates such a directory organization.

FIGURE 2.1 Example directory structure.

The file directory `documents` *contains* the files `plan`, `dact`, `sys.A`, `new.hire`, `no.JSK`, and `AMG.reply`. The directory `programs` contains the files `wb`, `collect`, and `mon`. At some point, you may decide to further categorize the files in a directory. This can be done by creating subdirectories and then placing each file into the appropriate subdirectory. For example, you might want to create subdirectories called `memos`, `proposals`, and `letters` inside your `documents` directory, as shown in Figure 2.2.

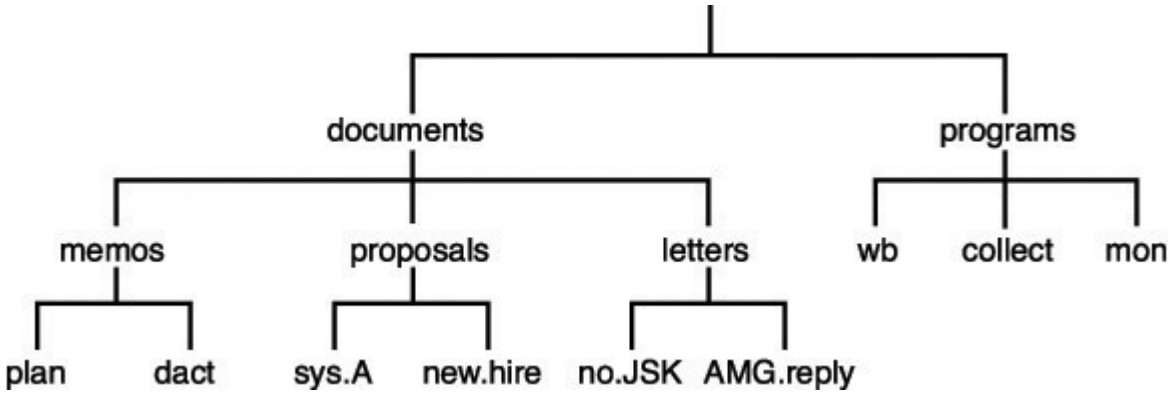

FIGURE 2.2 Directories containing subdirectories.

`documents` contains the subdirectories `memos`, `proposals`, and `letters`. Each of these directories in turn contains two files: `memos` contains `plan` and `dact`; `proposals` contains `sys.A` and `new.hire`; and `letters` contains `no.JSK` and `AMG.reply`.

Although each file in a given directory must have a unique name, files contained in different directories do not. So, for example, you could have a file in your `programs` directory called `dact`, even though a file by that name also exists in the `memos` subdirectory.

The Home Directory and Pathnames

The Unix system always associates each user of the system with a particular directory. When you log in to the system, you are placed automatically into a directory called your *home* directory.

Although the location of users' home directories can vary from one Unix version to the next, and even one user to the next, let's assume that your home directory is called `steve` and that this directory is actually a subdirectory of a directory called `users`. Therefore, if you had the directories `documents` and `programs`, the overall directory structure would actually look something like Figure 2.3. A special directory known as `/` (pronounced *slash*) is shown at the top of the directory tree. This directory is known as the *root*.

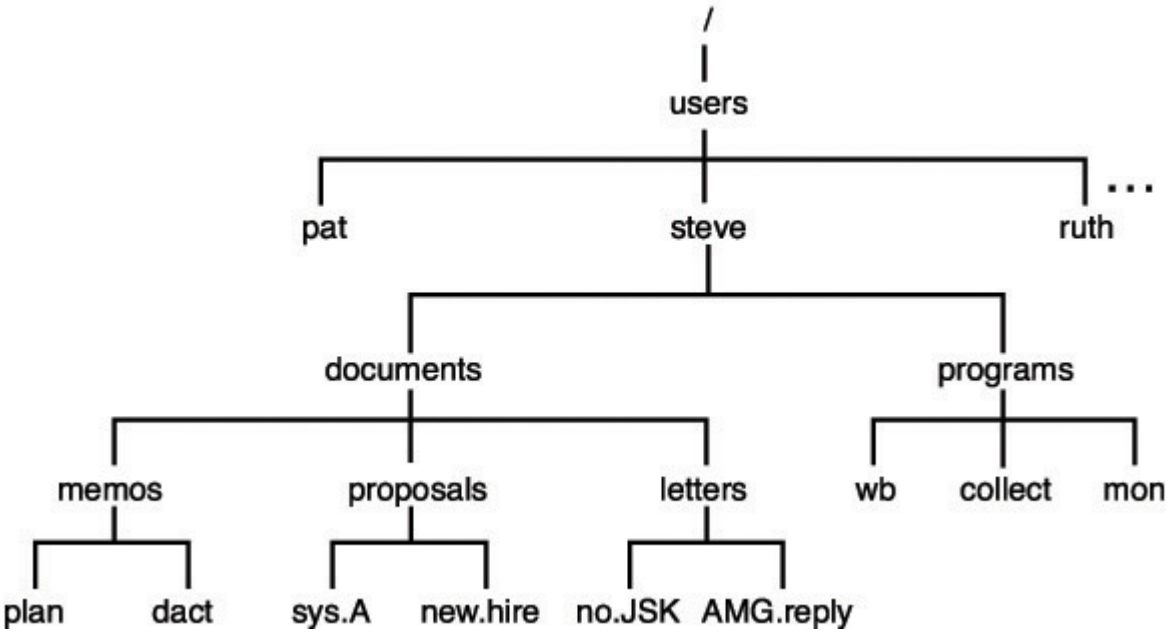

FIGURE 2.3 Hierarchical directory structure.

Whenever you are "inside" a particular directory (called your *current working* directory), the files contained within that directory are immediately accessible. If you want to access a file from another directory, you can either first issue a command to "change" to the appropriate directory and then access the particular file, or you can specify the particular file by its pathname.

A pathname enables you to uniquely identify a particular file to the Unix system. In the specification of a pathname, successive directories along the path are separated by the slash character `/`. A pathname that *begins* with a slash character is known as a *full* pathname because it specifies a complete path from the root. So, for example, the pathname `/users/steve` identifies the directory `steve` contained under the directory `users`. Similarly, the pathname `/users/steve/documents` references the directory `documents` as contained in the directory `steve` under `users`. As a final example, the pathname `/users/steve/documents/letters/AMG.reply` identifies the file `AMG.reply` contained along the appropriate directory path.

To help reduce some of the typing that would otherwise be required, Unix provides certain notational conveniences. Pathnames that do not begin with a slash character are known as *relative* pathnames. The path is relative to your current working directory. For example, if you just logged in to the system and were placed into your home directory `/users/steve`, you could directly reference the directory `documents` simply by typing `documents`. Similarly, the relative pathname `programs/mon` could be typed to access the file `mon` contained inside your `programs` directory.

By convention, the directory name `..` always references the directory that is one level higher. For example, after logging in and being placed into your home directory `/users/steve`, the pathname `..` would reference the directory `users`. And if you had issued the appropriate command to change your working directory to `documents/letters`, the pathname `..` would reference the `documents` directory, `../..` would reference the directory `steve`, and `../proposals/new.hire` would reference the file `new.hire` contained in the `proposals` directory. Note that in this case, as in most cases, there is usually more than one way to specify a path to a particular file.

Another notational convention is the single period `.`, which always refers to the current directory.

Now it's time to examine commands designed for working with directories.

Displaying Your Working Directory: The `pwd` Command

The `pwd` command is used to help you "get your bearings" by telling you the name of your current working directory.

Recall the directory structure from Figure 2.3. The directory that you are placed in after you log in to the

system is called your home directory. You can assume from Figure 2.3 that the home directory for the user `steve` is `/users/steve`. Therefore, whenever `steve` logs in to the system, he will automatically be placed inside this directory. To verify that this is the case, the `pwd` (print working directory) command can be issued:

```
$ pwd
/users/steve
$
```

The output from the command verifies that `steve`'s current working directory is `/users/steve`.

Changing Directories: The `cd` Command

You can change your current working directory by using the `cd` command. This command takes as its argument the name of the directory you want to change to.

Let's assume that you just logged in to the system and were placed inside your home directory, `/users/steve`. This is depicted by the arrow in Figure 2.4.

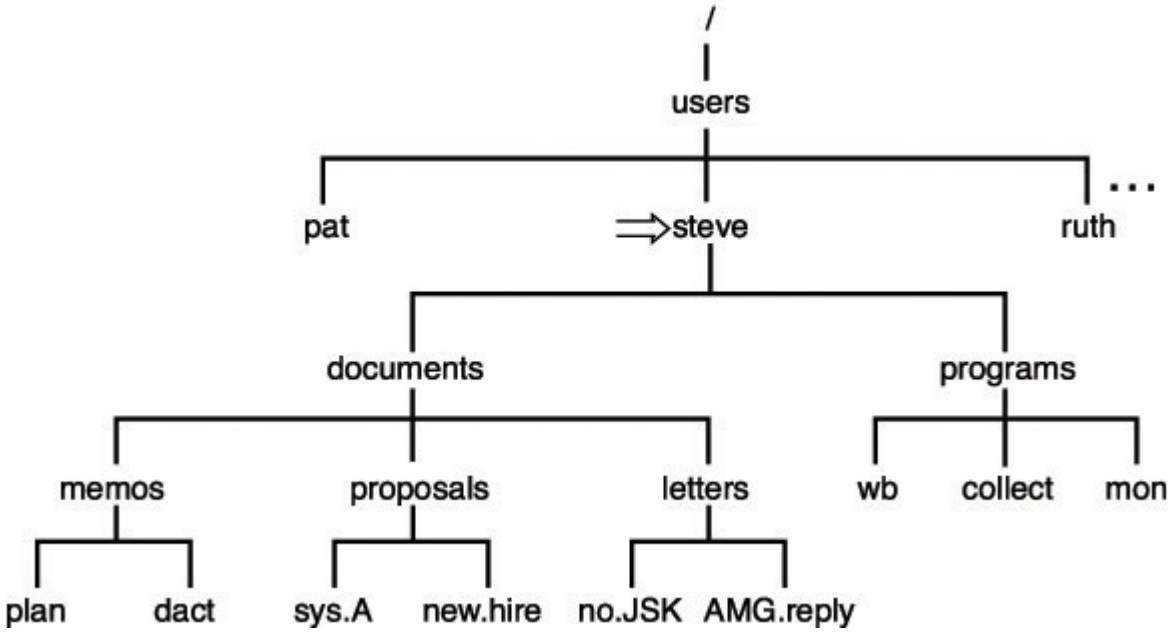

FIGURE 2.4 Current working directory is `steve`.

You know that two directories are directly "below" `steve`'s home directory: `documents` and `programs`. In fact, this can be verified at the terminal by issuing the `ls` command:

```
$ ls
documents
programs
$
```

The `ls` command lists the two directories `documents` and `programs` the same way it listed other ordinary files in previous examples.

To change your current working directory, issue the `cd` command, followed by the name of the directory to change to:

```
$ cd documents
$
```

After executing this command, you will be placed inside the documents directory, as depicted in Figure 2.5.

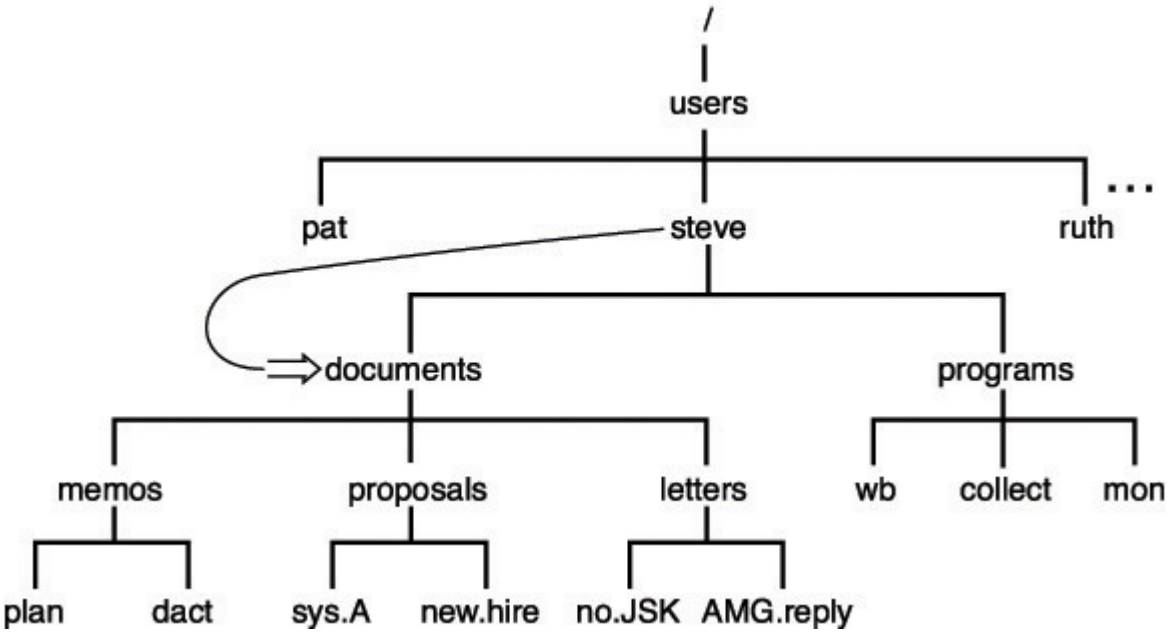

FIGURE 2.5 `cd documents`.

You can verify at the terminal that the working directory has been changed by issuing the `pwd` command:

```
$ pwd
/users/steve/documents
$
```

The easiest way to get one level up in a directory is to issue the command

```
cd ..
```

because by convention `..` always refers to the directory one level up (known as the *parent* directory; see Figure 2.6).

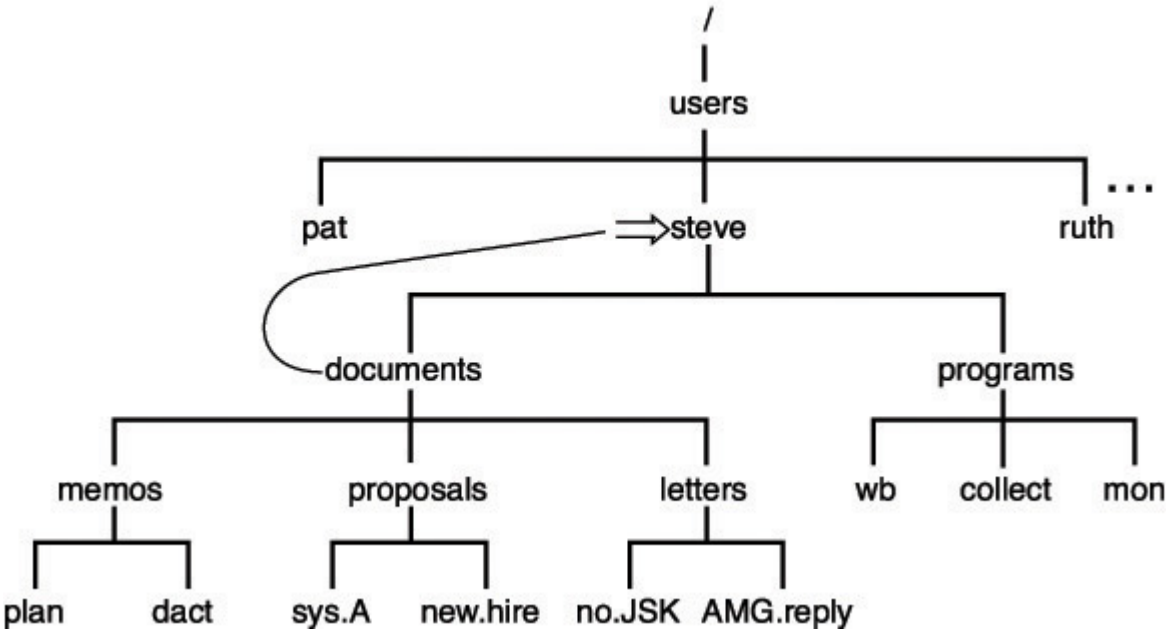

FIGURE 2.6 cd ..

```
$ cd ..
$ pwd
/users/steve
$
```

If you wanted to change to the `letters` directory, you could get there with a single `cd` command by specifying the relative path `documents/letters` (see Figure 2.7):

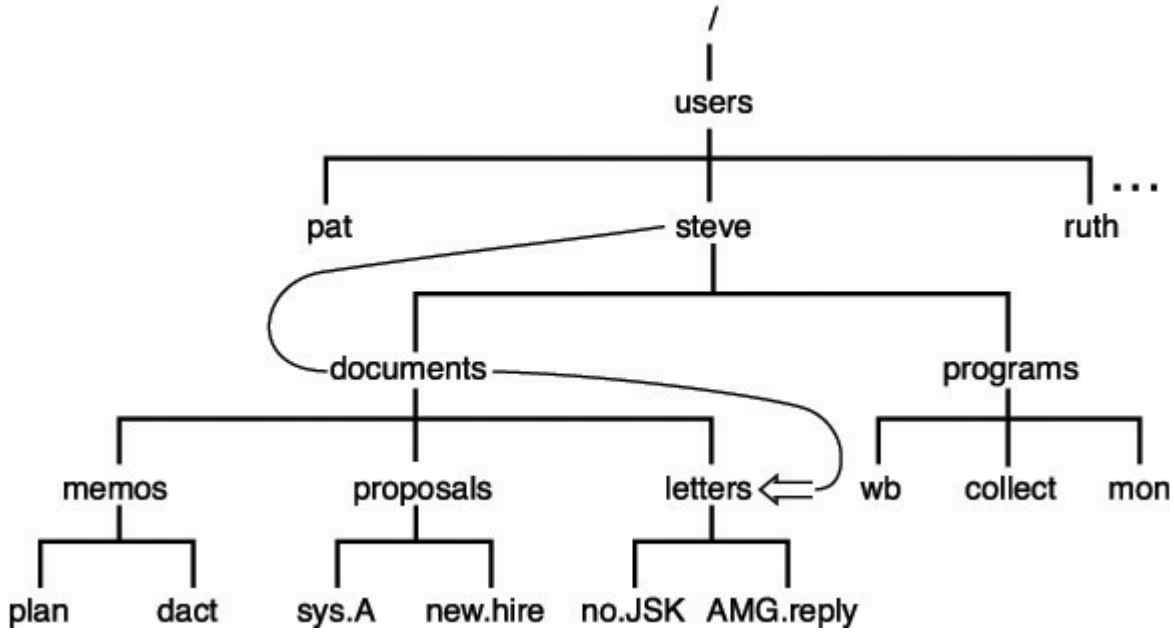

FIGURE 2.7 `cd documents/letters`.

Click here to view code image

```
$ cd documents/letters
$ pwd
/users/steve/documents/letters
$
```

You can get back up to the home directory by using a single `cd` command to go up two directories as shown:

```
$ cd ../..
$ pwd
/users/steve
$
```

Or you can get back to the home directory using a full pathname rather than a relative one:

```
$ cd /users/steve
$ pwd
/users/steve
$
```

Finally, there is a third way to get back to the home directory that is also the easiest. Typing the command `cd` *without* an argument *always* places you back into your home directory, no matter where you are in your directory path:

```
$ cd
$ pwd
/users/steve
$
```

More on the `ls` Command

When you type the `ls` command, the files contained in the current working directory are listed. But you can also use `ls` to obtain a list of files in other directories by supplying an argument to the command. First let's get back to your home directory:

```
$ cd
$ pwd
/users/steve
$
```

Now let's take a look at the files in the current working directory:

```
$ ls
documents
programs
$
```

If you supply the name of one of these directories to the `ls` command, you can get a list of the contents of that directory. So, you can find out what's contained in the `documents` directory simply by typing the command `ls documents`:

```
$ ls documents
letters
memos
proposals
$
```

To take a look at the subdirectory `memos`, you follow a similar procedure:

```
$ ls documents/memos
dact
plan
$
```

If you specify a nondirectory file argument to the `ls` command, you simply get that filename echoed back at the terminal:

```
$ ls documents/memos/plan
documents/memos/plan
$
```

An option to the `ls` command enables you to determine whether a particular file is a directory, among other things. The `-l` option (the letter l) provides a more detailed description of the files in a directory. If you were currently in `steve`'s home directory as indicated in Figure 2.6, the following would illustrate the effect of supplying the `-l` option to the `ls` command:

Click here to view code image

```
$ ls –l
total 2
drwxr-xr-x   5 steve   DP3725    80 Jun 25 13:27 documents
drwxr-xr-x   2 steve   DP3725    96 Jun 25 13:31 programs
$
```

The first line of the display is a count of the total number of *blocks* (1,024 bytes) of storage that the listed files use. Each successive line displayed by the `ls -l` command contains detailed information about a file in the directory. The first character on each line tells whether the file is a directory. If the character is `d`, it is a directory; if it is `-`, it is an ordinary file; finally, if it is `b`, `c`, `l`, or `p`, it is a special file.

The next nine characters on the line tell how every user on the system can access the *particular* file. These *access modes* apply to the file's owner (the first three characters), other users in the same *group* as the file's owner (the next three characters), and finally to all other users on the system (the last three characters). They tell whether the user can read from the file, write to the file, or execute the contents of the file.

The `ls -l` command lists the *link* count (see "Linking Files: The `ln` Command," later in this chapter), the owner of the file, the group owner of the file, how large the file is (that is, how many characters are

contained in it), and when the file was last modified. The information displayed last on the line is the filename itself.

Click here to view code image

```
$ ls -l programs
total 4
-rwxr-xr-x    1 steve    DP3725     358 Jun 25 13:31 collect
-rwxr-xr-x    1 steve    DP3725    1219 Jun 25 13:31 mon
-rwxr-xr-x    1 steve    DP3725      89 Jun 25 13:30 wb
$
```

The dash in the first column of each line indicates that the three files `collect`, `mon`, and `wb` are ordinary files and not directories.

Creating a Directory: The `mkdir` Command

To create a directory, the `mkdir` command must be used. The argument to this command is simply the name of the directory you want to make. For example, assume that you are still working with the directory structure depicted in Figure 2.7 and that you want to create a new directory called `misc` *on the same level* as the directories `documents` and `programs`. If you were currently in your home directory, typing the command `mkdir misc` would achieve the desired effect:

```
$ mkdir misc
$
```

Now if you execute an `ls` command, you should get the new directory listed:

```
$ ls
documents
misc
programs
$
```

The directory structure now appears as shown in Figure 2.8.

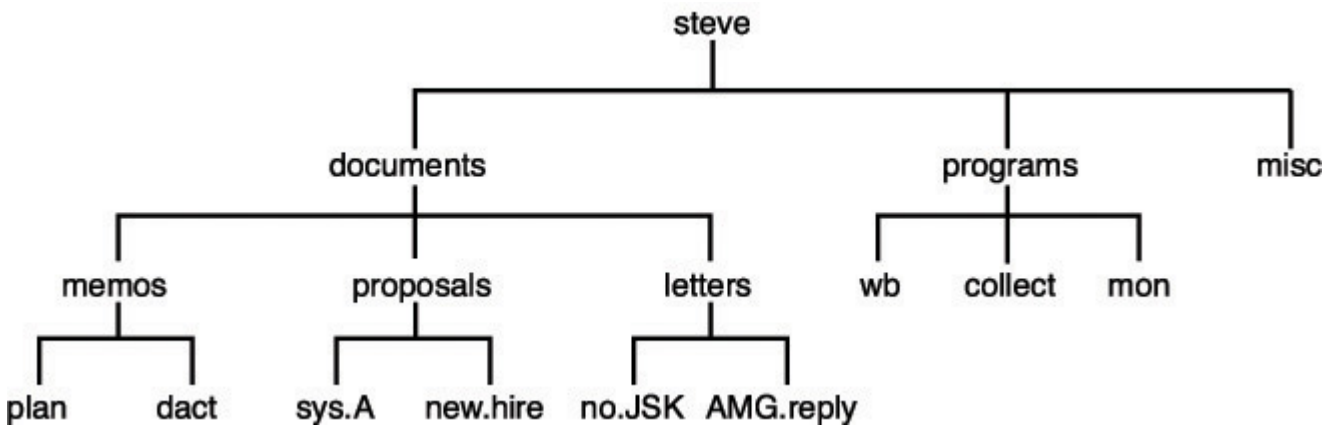

FIGURE 2.8 Directory structure with newly created `misc` directory.

Copying a File from One Directory to Another

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

```
$ cp programs/wb misc/wbx
$
```

Because the two files are contained in different directories, it is not even necessary that they be given

different names:

```
$ cp programs/wb misc/wb
$
```

When the destination file has the same name as the source file (in a different directory, of course), it is necessary to specify only the destination directory as the second argument:

```
$ cp programs/wb misc
$
```

When this command gets executed, the Unix system recognizes that the second argument is the name of a directory and copies the source file into that directory. The new file is given the same name as the source file. You can copy more than one file into a directory by listing the files to be copied before the name of the destination directory. If you were currently in the `programs` directory, the command

```
$ cp wb collect mon ../misc
$
```

would copy the three files `wb`, `collect`, and `mon` into the `misc` directory, under the same names.

To copy a file from another directory into your current one and give it the same name, use the fact that the current directory can always be referenced as '`.`':

```
$ pwd
/users/steve/misc
$ cp ../programs/collect .
$
```

The preceding command copies the file `collect` from the directory `../programs` into the current directory (`/users/steve/misc`).

Moving Files Between Directories

You recall that the `mv` command can be used to rename a file. However, when the two arguments to this command reference different directories, the file is actually moved from the first directory into the second directory. For example, first change from the home directory to the `documents` directory:

```
$ cd documents
$
```

Suppose that now you decide that the file `plan` contained in the `memos` directory is really a proposal and not a memo. So you want to move it from the `memos` directory into the `proposals` directory. The following would do the trick:


```
$ mv memos/plan proposals/plan
$
```

As with the `cp` command, if the source file and destination file have the same name, only the name of the destination directory need be supplied.

```
$ mv memos/plan proposals
$
```

Also like the `cp` command, a group of files can be simultaneously moved into a directory by simply listing all files to be moved before the name of the destination directory:

```
$ pwd
/users/steve/programs
```

```
$ mv wb collect mon ../misc
$
```

This would move the three files `wb`, `collect`, and `mon` into the directory `misc`. You can also use the `mv` command to change the name of a directory. For example, the following renames the directory `programs` to `bin`.

```
$ mv programs bin
$
```

Linking Files: The `ln` Command

In simplest terms, the `ln` command provides an easy way for you to give more than one name to a file. The general form of the command is

`ln` *from to*

This links the file *from* to the file *to*.

Recall the structure of `steve`'s `programs` directory from Figure 2.8. In that directory, he has stored a program called `wb`. Suppose that he decides that he'd also like to call the program `writeback`. The most obvious thing to do would be to simply create a copy of `wb` called `writeback`:

```
$ cp wb writeback
$
```

The drawback with this approach is that now twice as much disk space is being consumed by the program. Furthermore, if `steve` ever changes `wb`, he may forget to make a new copy of `writeback`, resulting in two different copies of what he thinks is the same program.

By linking the file `wb` to the new name, these problems are avoided:

```
$ ln wb writeback
$
```

Now instead of two copies of the file existing, only one exists with two different names: `wb` and `writeback`. The two files have been logically linked by the Unix system. As far as you're concerned, it appears as though you have two *different* files. Executing an `ls` command shows the two files separately:

```
$ ls
collect
mon
wb
writeback
$
```

Look what happens when you execute an `ls -l`:

Click here to view code image

```
$ ls -l
total 5
-rwxr-xr-x    1 steve    DP3725     358 Jun 25 13:31 collect
-rwxr-xr-x    1 steve    DP3725    1219 Jun 25 13:31 mon
-rwxr-xr-x    2 steve    DP3725      89 Jun 25 13:30 wb
-rwxr-xr-x    2 steve    DP3725      89 Jun 25 13:30 writeback
$
```

The number right before `steve` is 1 for `collect` and `mon` and 2 for `wb` and `writeback`. This number is the number of links to a file, normally 1 for nonlinked, nondirectory files. Because `wb` and `writeback` are linked, this number is 2 for these files. This implies that you can link to a file more than once.

You can remove either of the two linked files at any time, and the other will not be removed:

Click here to view code image

```
$ rm writeback
$ ls -l
total 4
-rwxr-xr-x   1 steve   DP3725    358 Jun 25 13:31 collect
-rwxr-xr-x   1 steve   DP3725   1219 Jun 25 13:31 mon
-rwxr-xr-x   1 steve   DP3725     89 Jun 25 13:30 wb
$
```

Note that the number of links on `wb` went from 2 to 1 because one of its links was removed.

Most often, `ln` is used to link files between directories. For example, suppose that `pat` wanted to have access to `steve`'s `wb` program. Instead of making a copy for himself (subject to the same problems described previously) or including `steve`'s `programs` directory in his `PATH` (described in detail in Chapter 11, "Your Environment"), he can simply link to the file from his own program directory; for example:

Click here to view code image

```
$ pwd
/users/pat/bin                          pat's program directory
$ ls -l
total 4
-rwxr-xr-x    1 pat    DP3822    1358 Jan 15 11:01 lcat
-rwxr-xr-x    1 pat    DP3822     504 Apr 21 18:30 xtr
$ ln /users/steve/wb .                  link wb to pat's bin
$ ls -l
total 5
-rwxr-xr-x    1 pat    DP3822    1358 Jan 15 11:01 lcat
-rwxr-xr-x    2 steve  DP3725      89 Jun 25 13:30 wb
-rwxr-xr-x    1 pat    DP3822     504 Apr 21 18:30 xtr
$
```

Note that `steve` is still listed as the owner of `wb`, even though the listing came from `pat`'s directory. This makes sense, because really only one copy of the file exists—and it's owned by `steve`.

The only stipulation on linking files is that for ordinary links, the files to be linked together must reside on the same *file system*. If they don't, you'll get an error from `ln` when you try to link them. (To determine the different file systems on your system, execute the `df` command. The first field on each line of output is the name of a file system.)

To create links to files on different file systems (or perhaps on different networked systems), you can use the `-s` option to the `ln` command. This creates a *symbolic* link. Symbolic links behave a lot like regular links, except that the symbolic link points to the original file; if the original file is removed, the symbolic link no longer works. Let's see how symbolic links work with the previous example:

Click here to view code image

```
$ rm wb
$ ls -l
total 4
-rwxr-xr-x    1 pat      DP3822     1358 Jan 15 11:01 lcat
-rwxr-xr-x    1 pat      DP3822      504 Apr 21 18:30 xtr
$ ln -s /users/steve/wb ./symwb              Symbolic link to wb
$ ls -l
total 5
-rwxr-xr-x    1 pat      DP3822     1358 Jan 15 11:01 lcat
lrwxr-xr-x    1 pat      DP3822       15 Jul 20 15:22 symwb -> /users/steve/wb
-rwxr-xr-x    1 pat      DP3822      504 Apr 21 18:30 xtr
$
```

Note that `pat` is listed as the owner of `symwb`, and the file type is `l`, which indicates a symbolic link. The

size of the symbolic link is 15 (the file actually contains the string `/users/steve/wb`), but if we attempt to access the contents of the file, we are presented with the contents of its symbolic link, `/users/steve/wb`:

Click here to view code image

```
$ wc symwb
         5     9    89 symwb
$
```

The `-L` option to the `ls` command can be used with the `-l` option to get a detailed list of information on the file the symbolic link points to:

Click here to view code image

```
$ ls -Ll
total 5
-rwxr-xr-x   1 pat     DP3822     1358 Jan 15 11:01 lcat
-rwxr-xr-x   2 steve   DP3725       89 Jun 25 13:30 wb
-rwxr-xr-x   1 pat     DP3822      504 Apr 21 18:30 xtr
$
```

Removing the file that a symbolic link points to invalidates the symbolic link (because symbolic links are maintained as filenames), although the symbolic link continues to stick around:

Click here to view code image

```
$ rm /users/steve/wb          Assume pat can remove this file
$ ls -l
total 5
-rwxr-xr-x   1 pat    DP3822    1358 Jan 15 11:01 lcat
lrwxr-xr-x   1 pat    DP3822      15 Jul 20 15:22 wb -> /users/steve/wb
-rwxr-xr-x   1 pat    DP3822     504 Apr 21 18:30 xtr
$ wc wb
Cannot open wb: No such file or directory
$
```

This type of file is called a *dangling symbolic link* and should be removed unless you have a specific reason to keep it around (for example, if you intend to replace the removed file).

One last note before leaving this discussion: The `ln` command follows the same general format as `cp` and `mv`, meaning that you can link a bunch of files at once into a directory using the format

`ln` *`files directory`*

Removing a Directory: The `rmdir` Command

You can remove a directory with the `rmdir` command. The stipulation involved in removing a directory is that no files be contained in the directory. If there *are* files in the directory when `rmdir` is executed, you will not be allowed to remove the directory. To remove the directory `misc` that you created earlier, the following could be used:

```
$ rmdir /users/steve/misc
$
```

Once again, the preceding command works only if no files are contained in the `misc` directory; otherwise, the following happens:

Click here to view code image

```
$ rmdir /users/steve/misc
rmdir: /users/steve/misc not empty
$
```

If this happens and you still want to remove the `misc` directory, you would first have to remove all the files contained in that directory before reissuing the `rmdir` command.

As an alternate method for removing a directory and the files contained in it, you can use the `-r` option to the `rm` command. The format is simple:

`rm -r` *dir*

where *dir* is the name of the directory that you want to remove. `rm` removes the indicated directory and *all* files (including directories) in it.

Filename Substitution

The Asterisk

One powerful feature of the Unix system that is actually handled by the shell is *filename substitution*. Let's say that your current directory has these files in it:

```
$ ls
chapt1
chapt2
chapt3
chapt4
$
```

Suppose that you want to print their contents at the terminal. Well, you could take advantage of the fact that the `cat` command allows you to specify more than one filename at a time. When this is done, the contents of the files are displayed one after the other:

Click here to view code image

```
$ cat chapt1 chapt2 chapt3 chapt4
     ...
$
```

But you can also type in

```
$ cat *
     ...
$
```

and get the same results. The shell automatically *substitutes* the names of all the files in the current directory for the `*`. The same substitution occurs if you use `*` with the `echo` command:

```
$ echo *
chapt1 chapt2 chapt3 chapt4
$
```

Here the `*` is again replaced with the names of all the files contained in the current directory, and the `echo` command simply displays them at the terminal.

Any place that `*` appears on the command line, the shell performs its substitution:

Click here to view code image

```
$ echo * : *
chapt1 chapt2 chapt3 chapt4 : chapt1 chapt2 chapt3 chapt4
$
```

The `*` can also be used in combination with other characters to limit the filenames that are substituted. For example, let's say that in your current directory you have not only `chapt1` through `chapt4` but also files `a`, `b`, and `c`:

```
$ ls
a
b
c
chapt1
chapt2
chapt3
chapt4
$
```

To display the contents of just the files beginning with `chapt`, you can type in

```
$ cat chapt*
      .
      .
      .
$
```

The `chapt*` matches any filename that *begins* with `chapt`. All such filenames matched are substituted on the command line.

The `*` is not limited to the end of a filename; it can be used at the beginning or in the middle as well:

```
$ echo *t1
chapt1
$ echo *t*
chapt1 chapt2 chapt3 chapt4
$ echo *x
*x
$
```

In the first `echo`, the `*t1` specifies all filenames that end in the characters `t1`. In the second `echo`, the first `*` matches everything up to a `t` and the second everything after; thus, all filenames containing a `t` are printed. Because there are no files ending with `x`, no substitution occurs in the last case. Therefore, the `echo` command simply displays `*x`.

Matching Single Characters

The asterisk (`*`) matches *zero* or more characters, meaning that `x*` matches the file `x` as well as `x1`, `x2`, `xabc`, and so on. The question mark (`?`) matches exactly one character. So `cat ?` prints all files with one-character names, just as `cat x?` prints all files with two-character names beginning with `x`.

Click here to view code image

```
$ ls
a
aa
aax
alice
b
bb
c
cc
report1
report2
report3
$ echo ?
a b c
$ echo a?
aa
$ echo ??
aa bb cc
$ echo ??*
aa aax alice bb cc report1 report2 report3
$
```

In the preceding example, the `??` matches two characters, and the `*` matches zero or more up to the end. The net effect is to match all filenames of two or more characters.

Another way to match a single character is to give a list of the characters to use in the match inside square brackets `[ ]`. For example, `[abc]` matches *one* letter `a`, `b`, or `c`. It's similar to the `?`, but it allows you to choose the characters that will be matched. The specification `[0-9]` matches the characters 0 *through* 9. The only restriction in specifying a *range* of characters is that the first character must be alphabetically less than the last character, so that `[z-f]` is not a valid range specification.

By mixing and matching ranges and characters in the list, you can perform some complicated substitutions. For example, `[a-np-z]*` matches all files that start with the letters `a` through `n` *or* `p` through `z` (or more simply stated, any lowercase letter but `o`).

If the first character following the `[` is a `!`, the sense of the match is inverted. That is, any character is matched *except* those enclosed in the brackets. So

`[!a-z]`

matches any character except a lowercase letter, and

`*[!o]`

matches any file that doesn't end with the lowercase letter `o`.

Table 2.1 gives a few more examples of filename substitution.

Command	Description
`echo a*`	Print the *names* of the files beginning with a
`cat *.c`	Print all files ending in `.c`
`rm *.*`	Remove all files containing a period
`ls x*`	List the names of all files beginning with `x`
`rm *`	Remove *all* files in the current directory (Note: Be careful when you use this.)
`echo a*b`	Print the names of all files beginning with a and ending with `b`
`cp ../programs/* .`	Copy all files from `../programs` into the current directory
`ls [a-z]*[!0-9]`	List files that begin with a lowercase letter and don't end with a digit

TABLE 2.1 Filename Substitution Examples

Standard Input/Output and I/O Redirection

Standard Input and Standard Output

Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from a place called *standard input*, which happens to be your terminal by default. Similarly, a command normally writes its output to *standard output*, which is also your terminal by default. This concept is depicted in Figure 2.9.

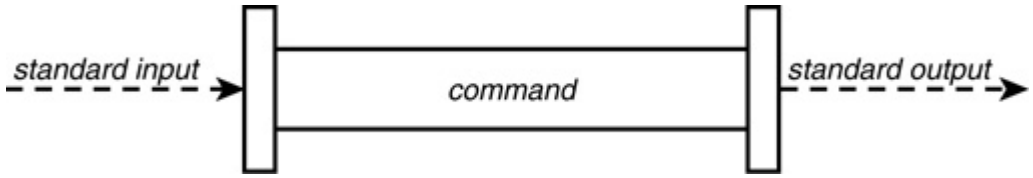

FIGURE 2.9 Typical Unix command.

Recall that executing the `who` command results in the display of the currently logged-in users. More formally, the `who` command writes a list of the logged-in users to standard output. This is depicted in Figure 2.10.

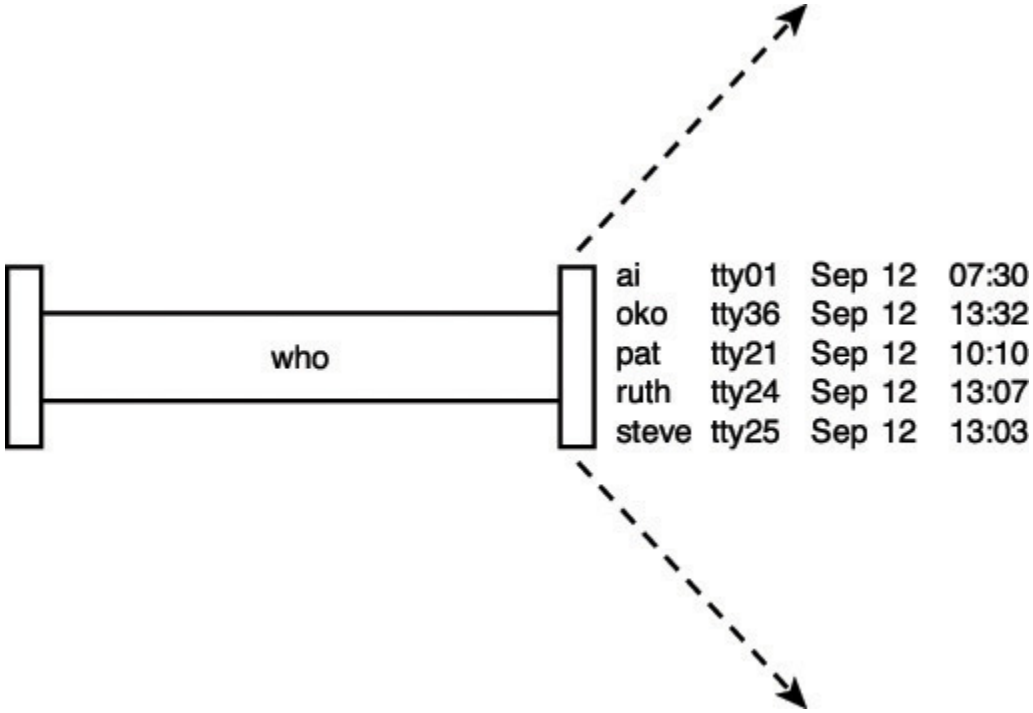

FIGURE 2.10 who command.

If a `sort` command is executed *without* a filename argument, the command takes its input from standard input. As with standard output, this is your terminal by default.

When entering data to a command from the terminal, the *Ctrl* and *d* keys (denoted *Ctrl+d* in this text) must be simultaneously pressed after the last data item has been entered. This tells the command that you have finished entering data. As an example, let's use the `sort` command to sort the following four names: Tony, Barbara, Harry, Dick. Instead of first entering the names into a file, we'll enter them directly from the terminal:

```
$ sort
Tony
Barbara
Harry
Dick
Ctrl+d
Barbara
Dick
Harry
Tony
$
```

Because no filename was specified to the `sort` command, the input was taken from standard input, the terminal. After the fourth name was typed in, the *Ctrl* and *d* keys were pressed to signal the end of the data. At that point, the `sort` command sorted the four names and displayed the results on the standard output device, which is also the terminal. This is depicted in Figure 2.11.

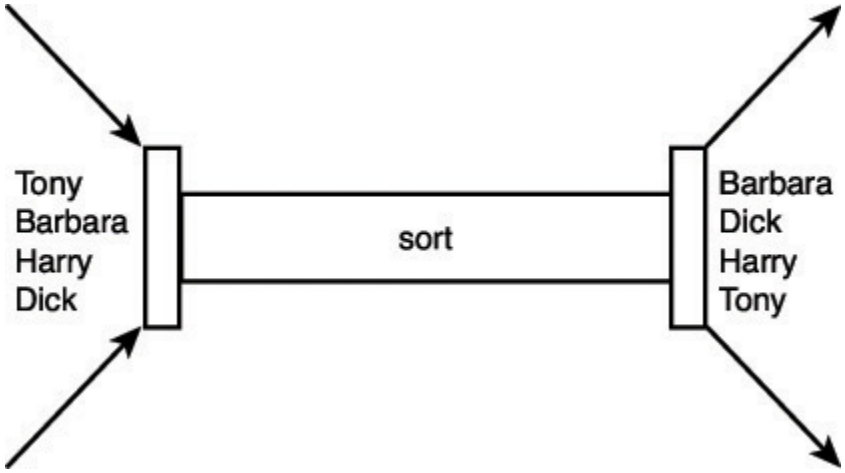

FIGURE 2.11 `sort` command.

The `wc` command is another example of a command that takes its input from standard input if no filename is specified on the command line. So the following shows an example of this command used to count the number of lines of text entered from the terminal:

```
$ wc -l
This is text that
is typed on the
standard input device.
Ctrl+d
      3
$
```

Note that the *Ctrl+d* that is used to terminate the input is not counted as a separate line by the `wc` command. Furthermore, because no filename was specified to the `wc` command, only the count of the number of lines (3) is listed as the output of the command. (Recall that this command normally prints the name of the file directly after the count.)

Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as *output redirection.*

If the notation > *file* is appended to *any* command that normally writes its output to standard output, the output of that command will be written to *file* instead of your terminal:

```
$ who > users
$
```

This command line causes the `who` command to be executed and its output to be written into the file `users`. Notice that no output appears at the terminal. This is because the output has been *redirected* from the default standard output device (the terminal) into the specified file:

```
$ cat users
oko   tty01  Sep 12 07:30
ai    tty15  Sep 12 13:32
ruth  tty21  Sep 12 10:10
pat   tty24  Sep 12 13:07
steve tty25  Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example:

```
$ echo line 1 > users
$ cat users
line 1
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

The second `echo` command uses a different type of output redirection indicated by the characters `>>`. This character pair causes the standard output from the command to be *appended* to the specified file. Therefore, the previous contents of the file are not lost, and the new output simply gets added onto the end.

By using the redirection append characters `>>`, you can use `cat` to append the contents of one file onto the end of another:

Click here to view code image

```
$ cat file1
This is in file1.
$ cat file2
This is in file2.
$ cat file1 >> file2                Append file1 to file2
$ cat file2
This is in file2.
This is in file1.
$
```

Recall that specifying more than one filename to `cat` results in the display of the first file followed immediately by the second file, and so on:

Click here to view code image

```
$ cat file1
This is in file1.
$ cat file2
This is in file2.
$ cat file1 file2
This is in file1.
This is in file2.
$ cat file1 file2 > file3              Redirect it instead
$ cat file3
This is in file1.
This is in file2.
$
```

Now you can see where the `cat` command gets its name: When used with more than one file, its effect is to *catenate* the files together.

Incidentally, the shell recognizes a special format of output redirection. If you type

> file

not preceded by a command, the shell creates an empty (that is, zero character length) *file* for you. If *file* previously exists, its contents will be lost.

Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. And as the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command. Of course, only commands that normally take their input from standard input can have their input redirected from a file in this manner.

To redirect the input of a command, you type the < character followed by the name of the file that the input is to be read from. So, for example, to count the number of lines in the file `users`, you know that you can execute the command `wc -l users`:

```
$ wc -l users
      2 users
$
```

Or, you can count the number of lines in the file by redirecting the standard input of the `wc` command from the file `users`:

```
$ wc -l < users
      2
$
```

Note that there is a difference in the output produced by the two forms of the `wc` command. In the first case, the name of the file `users` is listed with the line count; in the second case, it is not. This points out the subtle distinction between the execution of the two commands. In the first case, `wc` knows that it is reading its input from the file `users`. In the second case, it only knows that it is reading its input from standard input. The shell redirects the input so that it comes from the file `users` and not the terminal (more about this in the next chapter). As far as `wc` is concerned, it doesn't know whether its input is coming from the terminal or from a file!

Pipes

As you will recall, the file `users` that was created previously contains a list of all the users currently logged in to the system. Because you know that there will be one line in the file for each user logged in to the system, you can easily determine the *number* of users logged in by simply counting the number of lines in the `users` file:

```
$ who > users
$ wc -l < users
      5
$
```

This output would indicate that currently five users were logged in. Now you have a command sequence you can use whenever you want to know how many users are logged in.

Another approach to determine the number of logged-in users bypasses the use of a file. The Unix system allows you to effectively connect two commands together. This connection is known as a *pipe*, and it enables you to take the output from one command and feed it directly into the input of another command. A pipe is effected by the character `|`, which is placed between the two commands. So to make a pipe between the `who` and `wc -l` commands, you simply type `who | wc -l`:

```
$ who | wc -l
      5
$
```

The pipe that is effected between these two commands is depicted in Figure 2.12.

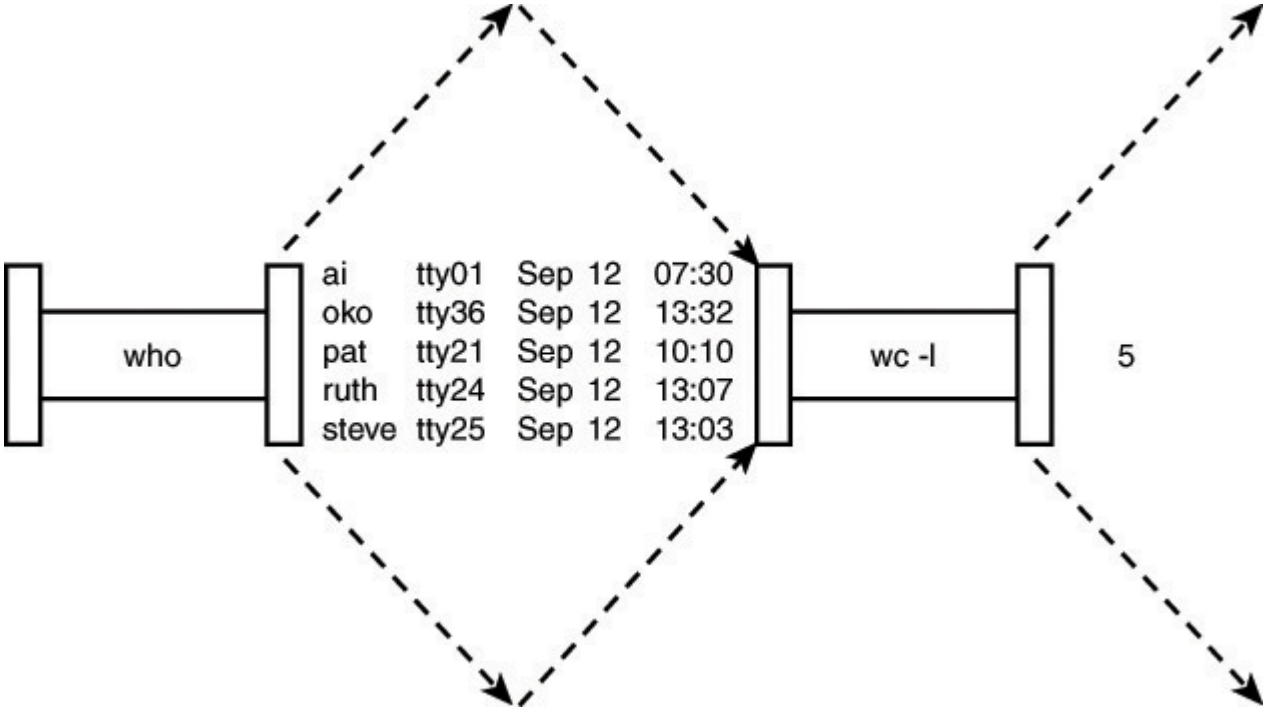

FIGURE 2.12 Pipeline process: `who | wc -l`.

When a pipe is set up between two commands, the standard output from the first command is connected directly to the standard input of the second command. You know that the `who` command writes its list of logged-in users to standard output. Furthermore, you know that if no filename argument is specified to the `wc` command, it takes its input from standard input. Therefore, the list of logged-in users that is output from the `who` command automatically becomes the input to the `wc` command. Note that you never see the output of the `who` command at the terminal because it is piped directly into the `wc` command. This is depicted in Figure 2.13.

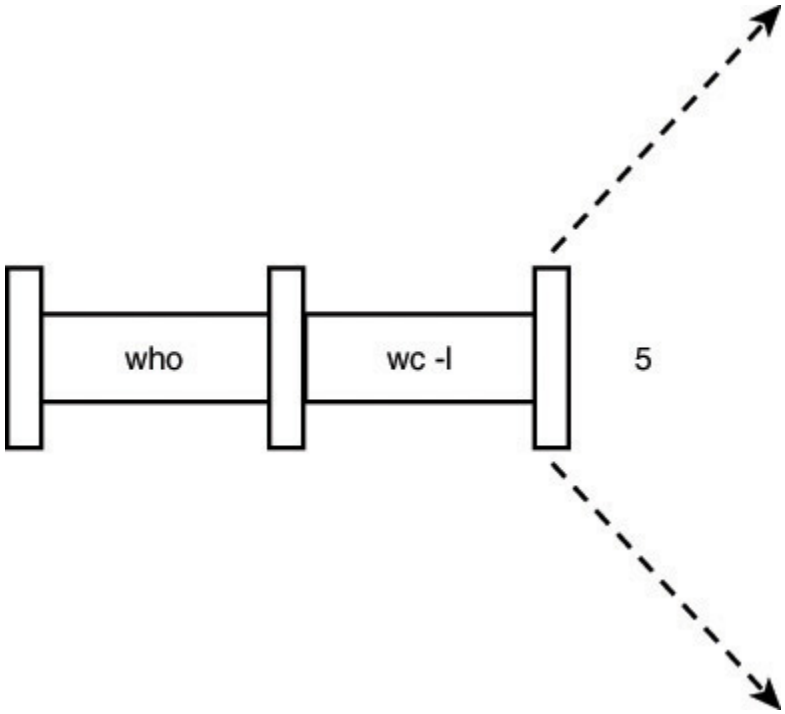

FIGURE 2.13 Pipeline process.

A pipe can be made between *any* two programs, provided that the first program writes its output to standard output, and the second program reads its input from standard input.

As another example of a pipe, suppose that you wanted to count the number of files contained in your directory. Knowledge of the fact that the `ls` command displays one line of output per file enables you to use the same type of approach as before:

```
$ ls | wc -l
      10
$
```

The output indicates that the current directory contains 10 files.

It is also possible to form a pipeline consisting of more than two programs, with the output of one program feeding into the input of the next.

Filters

The term *filter* is often used in Unix terminology to refer to any program that can take input from standard input, perform some operation on that input, and write the results to standard output. More succinctly, a filter is any program that can be used between two other programs in a pipeline. So in the previous pipeline, `wc` is considered a filter. `ls` is not because it does not read its input from standard input. As other examples, `cat` and `sort` are filters, whereas `who`, `date`, `cd`, `pwd`, `echo`, `rm`, `mv`, and `cp` are not.

Standard Error

In addition to standard input and standard output, there is another place known as *standard error*. This is where most Unix commands write their error messages. And as with the other two "standard" places, standard error is associated with your terminal by default. In most cases, you never know the difference between standard output and standard error:


```
$ ls n*                          List all files beginning with n
n* not found
$
```

Here the "not found" message is actually being written to standard error and not standard output by the `ls` command. You can verify that this message is not being written to standard output by redirecting the `ls` command's output:

```
$ ls n* > foo
n* not found
$
```

So, you still get the message printed out at the terminal, even though you redirected standard output to the file `foo`.

The preceding example shows the *raison d'être* for standard error: so that error messages will still get displayed at the terminal even if standard output is redirected to a file or piped to another command.

You can also redirect standard error to a file by using the notation

command `2>` *file*

No space is permitted between the `2` and the `>`. Any error messages normally intended for standard error will be diverted into the specified *file*, similar to the way standard output gets redirected.

```
$ ls n* 2> errors
$ cat errors
n* not found
$
```

More on Commands

Typing More Than One Command on a Line

You can type more than one command on a line provided that you separate each command with a semicolon. For example, you can find out the current time and also your current working directory by typing in the `date` and `pwd` commands on the same line:

```
$ date; pwd
Sat Jul 20 14:43:25 EDT 2002
/users/pat/bin
$
```

You can string out as many commands as you want on the line, as long as each command is delimited by a semicolon.

Sending a Command to the Background

Normally, you type in a command and then wait for the results of the command to be displayed at the terminal. For all the examples you have seen thus far, this waiting time is typically short—maybe a second or two. However, you may have to run commands that require many seconds or even minutes to execute. In those cases, you'll have to wait for the command to finish executing before you can proceed further *unless you execute the command in the background*.

If you type in a command followed by the ampersand character `&`, that command will be sent to the background for execution. This means that the command will no longer tie up your terminal, and you can then proceed with other work. The standard output from the command will still be directed to your terminal; however, in most cases the standard input will be dissociated from your terminal. If the command does try to read any input from standard input, it will be stopped and will wait for you to bring it to the foreground (we'll discuss this in more detail in Chapter 15, "Interactive and Nonstandard Shell Features").[3]

[3] *Note that the capability to stop a command when it reads from standard input may be missing on non-Unix implementations of the shell or on older shells that do not conform to the POSIX standard. On these implementations, any read from standard input will get an end-of-file condition as if Ctrl+d were typed.*

Click here to view code image

```
$ sort data > out &        Send the  sort  to the background
[1] 1258                   Process id
$ date                     Your terminal is immediately available to do other work
Sat Jul 20 14:45:09 EDT 2002
$
```

When a command is sent to the background, the Unix system automatically displays two numbers. The first is called the command's *job number* and the second the *process id*. In the preceding example, 1 was the job number and 1258 the process id. The job number is used by some shell commands that you'll learn more about in Chapter 15. The process id uniquely identifies the command that you sent to the background and can be used to obtain status information about the command. This is done with the `ps` command.

The `ps` Command

The `ps` command gives you information about the processes running on the system. `ps` without any options prints the status of just your processes. If you type in `ps` at your terminal, you'll get a few lines back describing the processes you have running:

Click here to view code image

```
$ ps
   PID   TTY   TIME COMMAND
   195    01   0:21 sh         The shell
  1353    01   0:00 ps         This ps command
  1258    01   0:10 sort       The previous sort
$
```

The `ps` command prints out four columns of information: `PID`, the process id; `TTY`, the terminal number that the process was run from; `TIME`, the amount of computer time in minutes and seconds that process has used; and `COMMAND`, the name of the process. (The `sh` process in the preceding example is the shell that was started when you logged in, and it has used 21 seconds of computer time.) Until the command is finished, it shows up in the output of the `ps` command as a running process. Process number 1353 in the preceding example is the `ps` command that was typed in, and 1258 is the `sort` from the preceding example.

When used with the `-f` option, `ps` prints out more information about your processes, including the *parent* process id (`PPID`), the time the processes started (`STIME`), and the command arguments:

Click here to view code image

```
$ ps -f
     UID   PID  PPID  C    STIME TTY      TIME COMMAND
   steve   195     1  0 10:58:29 tty01    0:21 -sh
   steve  1360   195 43 14:54:48 tty01    0:01 ps -f
   steve  1258   195  0 14:45:04 tty01    3:17 sort data
$
```

Command Summary

Table 2.2 summarizes the commands reviewed in this chapter. In this table, *file* refers to a file, *file(s)* to one or more files, *dir* to a directory, and *dir(s)* to one or more directories.

Command	Description
`cat file(s)`	Display contents of *file(s)* or standard input if not supplied
`cd dir`	Change working directory to *dir*
`cp file1 file2`	Copy *file1* to *file2*
`cp file(s) dir`	Copy *file(s)* into *dir*
`date`	Display the date and time
`echo args`	Display *args*
`ln file1 file2`	Link *file1* to *file2*
`ln file(s) dir`	Link *file(s)* into *dir*
`ls file(s)`	List *file(s)*
`ls dir(s)`	List files in *dir(s)* or in current directory if *dir(s)* is not specified
`mkdir dir(s)`	Create directory *dir(s)*
`mv file1 file2`	Move *file1* to *file2* (simply rename it if both reference the same directory)
`mv file(s) dir`	Move *file(s)* into directory *dir*
`ps`	List information about active processes
`pwd`	Display current working directory path
`rm file(s)`	Remove *files(s)*
`rmdir dir(s)`	Remove empty directory *dir(s)*
`sort file(s)`	Sort lines of *file(s)* or standard input if not supplied
`wc file(s)`	Count the number of lines, words, and characters in *file(s)* or standard input if not supplied
`who`	Display who's logged in

TABLE 2.2 Command Summary

Exercises

1. Given the following files in your current directory:

```
$ ls
feb96
jan12.02
jan19.02
jan26.02
jan5.02
jan95
jan96
jan97
jan98
mar98
memo1
memo10
memo2
memo2.sv
$
```

What would be the output from the following commands?

Click here to view code image

```
echo *                          echo *[!0-9]
echo em[a-df-z]*                echo [A-Z]*
echo jan*                       echo *.*
echo ?????                      echo *02
echo jan?? feb?? mar??          echo [fjm][ae][bnr]*
```

2. What is the effect of the following command sequences?

Click here to view code image

```
ls | wc -l                      rm ???
who | wc -l                     mv progs/* /users/steve/backup
ls *.c | wc -l                  rm *.o
who | sort                      cd; pwd
cp memo1 ..                     plotdata 2>errors &
```