CHAPTER

3

Touring Utilities and System Features

SKILLS CHECK

Before beginning this chapter, you should be able to:

- Access the system
- Run programs to obtain system and user information
- Properly communicate basic instructions to the shell
- Navigate to other directories in the filesystem
- Use standard programs to create, examine, and manage files

OBJECTIVES

After completing this chapter, you will be able to:

- Use utilities to locate lines containing specific criteria, count elements, and sort lines of a file
- Manage input and output from utilities
- Employ shell special characters to give instructions
- Manage user processes
- Modify the computing environment
- Create and execute a basic shell script

This chapter completes the tour of the major features of the system including communicating with the shell to execute processes, navigating the filesystem and employing permissions. Getting work done in UNIX/Linux generally entails asking the shell to execute utilities in specific, often complex ways. To accomplish real work, we need to give the shell exact and detailed instructions about what it should do, as well as what instructions it should pass to the needed utilities. We can issue commands to tailor or modify many aspects of UNIX and Linux to meet our particular needs. A functioning system also includes system files, directories of programs, and a system of permissions for security.

3.1 Employing Fundamental Utilities

In the previous chapter, you used several programs, or utilities, to locate system information and output it to the display. You also used utilities to remove, rename, and manipulate user files. Each utility is a tool that performs a set of very specific tasks. This section examines several new utilities. Some utilities read input from files, modify the data that they read, and send the output to your screen, to a file, or to another utility. Others provide information about the contents of a directory or information about other utilities.

Listing the Contents of the Directory

We use **ls** to output the filenames listed in the current directory. We can also instruct **ls** to provide more information by passing options as arguments.

1. List the files in the current directory with each of the following:

   ls

   ls -F

   The **ls** utility simply lists all filenames. With the **-F** option, **ls** places a forward slash at the end of the name of each directory. If you have any files that are executable, they will each have an asterisk (*****) after the name.

2. List the files in the current directory, but tell **ls** to provide a long listing:

   ls -l (minus *el*)

   The output is a list of filenames and other information, one file to a line. The important part on each line is the file's access permissions and other data about the file.

PERMISSION	LINKS	OWNER	GROUP	SIZE	DATE	TIME	NAME
drwxr-x- - -	3	cassy	staff	4096	Jan 3	14:27	Desk
-rw-rw-r- -	1	cassy	staff	62	Jan 5	08:14	cream-puff

TABLE 3-1 Fields in the Output of ***ls -l***

If a filename begins with a period, **ls** is programmed to treat it as a hidden "housekeeping" file, and does not include it when you ask for a listing of the directory's filenames.

3. We can instruct **ls** to list the dot files in its output by including the appropriate option. Enter:

   ls

   ls -a

   When we include the **-a** (**a**ll) option to **ls**, it includes in the output **a**ll files in the current directory, including the dot or hidden files. The files such as ***.profile***, ***.login***, ***.cshrc***, and so on are files read by shells and other programs when they are started. We use them to convey instructions to our programs.

4. We can combine options to **ls**. Enter:

   ls -alF

The output is the result of all three options. It includes all files, with the long listing of information about each, and directories are marked with a slash.

Counting the Elements of a File

In Chapter 2, we used **wc** to count the number of lines, words, and characters in a file.

1. Enter the following command to examine the contents of a file created in Chapter 2:

   wc *users_on*

   The output from the **wc** (**w**ord **c**ount, not water closet) utility consists of four fields:

```
2      12      102      users_on
```

   The meaning of each field in the output of the **wc** utility is shown here:

Number of Lines	Number of Words	Number of Characters	File
2	12	102	users_on

In addition to counting the elements in files, we can instruct **wc** to count the words, lines, and characters in the output of previous utilities in a command line.

2. Redirect output to **wc** by entering:

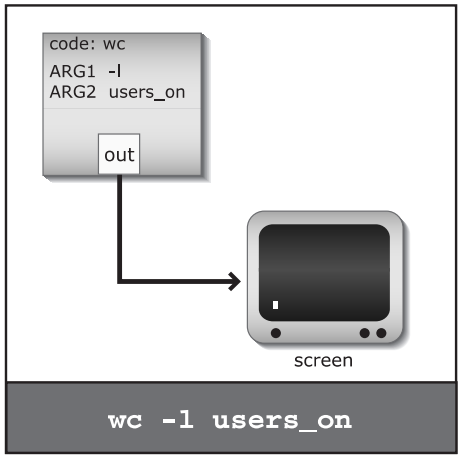
   date | wc

   who | wc

   The **who** utility outputs one line of information for each current user. The | is instruction to connect the output from **who** to the input of **wc**, which counts the elements. The **wc** utility then tosses the information that comes from **who** and just outputs its count totals.

3. We have used the **-l** option with **wc**. There are others. Try each of the options to **wc** we list in Table 3-2.

In each of the previous commands, we instructed the shell to pass two arguments to **wc**. The first argument included a dash such as **-c** and was interpreted as an option to output specific results. The second argument is interpreted as the name of a file to examine.



OPTION	OUTPUT
wc -l *users_on*	The count of lines only.
wc -w *users_on*	The count of words only.
wc -c *users_on*	The count of characters only.

TABLE 3-2 Options for the ***wc*** Command

Combining Utility Options in Arguments

The **wc** utility, like most other utilities, interprets more than one argument. Several questions arise: Can we issue multiple options? If so, must they be entered as separate arguments or can they be combined? What is the impact of the order of arguments?

For example, enter the following commands:

wc -c -l *users_on*
wc -lc *users_on*
wc -cl *users*_on

Both the **l**ine count and the **c**haracter count options are passed as arguments to **wc**, and the results are displayed. Evidently it makes no difference what order the arguments are entered (**-lc** or **-cl**), nor whether the options are entered as separate arguments or combined in one argument (**-c**, **-l,** or **-cl**).

Sorting Lines in a File

Many files contain data concerning users or individuals. We have briefly looked at ***/etc/passwd*** (the *password file*), which contains one line of information (a *record*) for each user. Every time a new user is added to the system, a new line is added (usually to the bottom of the file). As a result, the password file is not in a sorted order. In the following exercise, you will sort lines from the password file.

To make visual examination easier, start by creating a file consisting of the first portion of the password file.

1. Use **head** to create a file containing the first 20 lines of the password file on your system by entering:

   head ***-20 /etc/passwd*** > *mypasswd*

2. Examine the file by entering:

   cat *mypasswd*

3. Output a sorted version of the file by entering:

   sort *mypasswd*
   cat *mypasswd*

   The **sort** utility reads the file *mypasswd* into memory and rearranges the lines into a sorted order. Output is displayed on the screen. The *mypasswd* file itself is not modified; rather, its data is read, sorted, and written to your screen. We can save the sorted version in a new file.

4. Tell the shell to connect the output of **sort** to a file:

   sort *mypasswd* **>** *s-mypasswd*
   more *s-mypasswd*

The sorted version is redirected from the screen to the new file.

Employing Multiple Files with Utilities

We can also use the **sort** utility to sort multiple files.

1. Review the contents of two of your files by entering the following commands:

   cat *mypasswd*
   cat *users_on*

2. Use **sort** to sort the lines from the two files you just examined by entering:

   sort *mypasswd users_on* **|** **more**

   The contents of both files are read and sorted together. The resulting output is the lines from *mypasswd* and *users_on*, merged together and sorted.

   Examine the output. The **sort** utility reads both files (*mypasswd* and *users_on)* and sorts all the lines that it reads from both files. The files are not sorted individually. Neither the original *users_on nor mypasswd* file is changed.

3. Enter:

   wc *mypasswd users_on*
   more *mypasswd users_on*

Unlike **sort**, the **wc** utility operates on the files individually. It outputs the stats for each file and then produces a total.

Examining the Order Used by *sort*

Although the output from **sort** is sorted, it is not like a dictionary sort.

1. Examine the output after entering:

   sort *users_on lost-days* **|** **more**

   In the **sort** order, lines beginning with numbers are output first, then lines beginning with uppercase letters, and last, lines that begin with lowercase letters. This is the same order that characters are listed in the ASCII (American Standard Code for Information Interchange) order.

2. On most systems, you can examine the ASCII order by entering this command:

   man ***ascii***

When we press a key on the keyboard, we cannot send a character such as *k* down the wire to the computer. We can only transmit numbers. The ASCII table is an agreed-on set of numbers that represent all 128 characters we use. The letter *k* is 107. The order that characters are listed in the ASCII table is referred to as the *ASCII order*.

In ASCII order, most nonalphanumeric characters are first, then numbers, followed by uppercase characters, more nonalphanumeric characters, and then lowercase characters. Unless we instruct otherwise, the **sort** utility follows ASCII order when sorting lines.

Reversing the Sorted Order

To sort a file in reverse ASCII order, we must specify an option to the utility on the command line, instructing it to work in a particular way.

Type the following command:

```
sort users_on mypasswd | more
sort -r users_on mypasswd | more
```

Compare the two outputs. In the second command, you instruct the shell to run the **sort** utility and to pass it three arguments: the **-r** option and two arguments that **sort** interprets as filenames. The **-r** "**r**everse option" is one of several options to the **sort** utility that instruct **sort** to change the way it functions. We will examine others in Chapter 5.

Taking a Nap

Utilities perform a wide variety of functions. One of the most specialized simply counts a prescribed number of seconds and exits.

1. Try the following command:

   ```
   sleep 2
   ```

   There appears to be no response to the command; then, after two seconds, the shell displays a new prompt.
2. Employ different arguments:

   ```
   sleep 8
   sleep 4
   ```

The sleep utility interprets its argument as the number of seconds it should wait before exiting. As soon as it exits, the shell displays a new prompt. This utility is very useful for exploring how the system functions and for use in scripts.

Comparing Utilities' Interpretation of Arguments

When we pass an argument to a utility, that utility's code determines how the argument is interpreted.

1. Enter:

   cal *2004*

   The **cal** utility interprets the argument *2004* as instruction to output the calendar for the year 2004.
2. Instruct the shell to redirect the output from **cal** to a file and confirm it worked:

   cal *2004* **>** *2004*

   ls

   Among your files is a file named *2004*.
3. Enter:

   wc *2004*
   cat *2004*
   cal *2004*
   rm *2004*
   ls

The same argument is passed to four different utilities with four different results. The **wc** utility interprets the argument *2004* as the name of a file to read and to count the lines, words, and characters. To **cat**, *2004* is a file to read and output. To **cal**, *2004* is a calendar year to calculate and display. To **rm**, *2004* is a file to remove from the directory. The shell passes the argument. The utility interprets it.

Visiting echo Point

One utility simply reads whatever arguments we give it and writes the arguments to output.

1. We can give the same argument to **echo**:

   echo *2004*

   The argument consisting of the characters *2004* is displayed on the screen.

To **echo**, the argument *2004* is simply a string of characters that it reads and then writes to its output, which is connected to the screen unless we tell the shell to redirect it somewhere else.

2. Try several arguments:

   echo *these are five different arguments*

Five arguments are passed to **echo**, which interprets each as simply a character string to read and output. Because **echo** reads arguments and writes them to output, we will use the utility on the command line to see how arguments are processed and in shell scripts to display text on the user's screen.

Passing Arguments to Utilities

We enter command lines that consist of at least one utility and arguments. The shell passes the arguments to the utility, which interprets them.

1. Enter:

   echo *who date ls cat*

   The shell interprets the command line as instruction to run **echo** and pass it four arguments consisting of strings of characters, namely *who*, *date*, *ls*, and *cat*. The command is of the form:

   util *arg arg arg arg*

   To the shell, the tokens *who*, *date*, *ls*, and *cat* are not utilities to be run, because of their location on the command line. Rather, they are just arguments to pass to **echo**, which interprets them as just character strings. **echo** is programmed to read the arguments and write them to output.

2. Instruct the shell to redirect the output of **echo** to a file by entering:

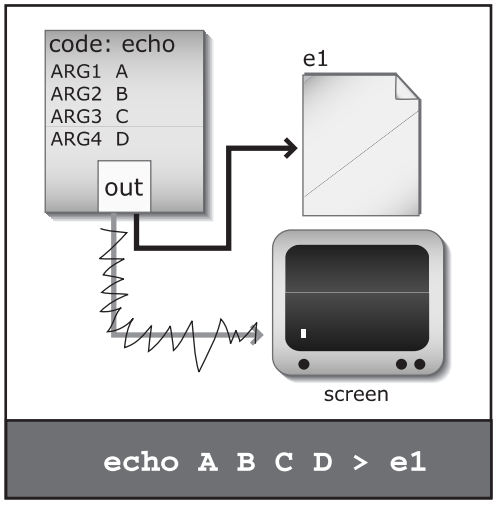
   echo *A B C D* **>** *e1*
   more *e1*

   This command line tells the shell to:

   a. Start a child process.

   b. Pass four arguments to the process.

c. Redirect the output of the process to a new file, *e1*.
d. Have the process execute the **echo** utility code.



When **echo** is executed, it reads the four arguments and simply writes them to output, which the shell had redirected to the new file *e1*.

Creating Combination Files

People often place related data in several different files, such as individual chapters of a book. At times, the data that is in several files needs to be brought together.

1. Type the following commands to create new files:

```
date > c1
echo hello this is echo > c2
ls > c3
ls
ls -l c1 c2 c3
```

The **ls** utility interprets the *c1*, *c2*, and *c3* arguments as names of files. A long listing of the information about each file is displayed.

2. Give **cat** the same three arguments:

```
cat c1 c2 c3
```

This command line instructs the shell to run the **cat** utility and to pass it three arguments. The **cat** utility interprets each argument as the name of

a file to open, read, and write to output. The **cat** utility reads each line from the first file and writes it to output (which is the screen by default). After **cat** reads and writes all the lines from the first file, it opens the next file, reads all the lines in it, and writes them to output. This process continues until **cat** reaches the end of the last file listed as an argument. The resulting output is the three files "spliced together" or con**cat**enated, hence the name for the utility.

We can also tell the shell to redirect the output of **cat** to a file.

3. Enter the following:

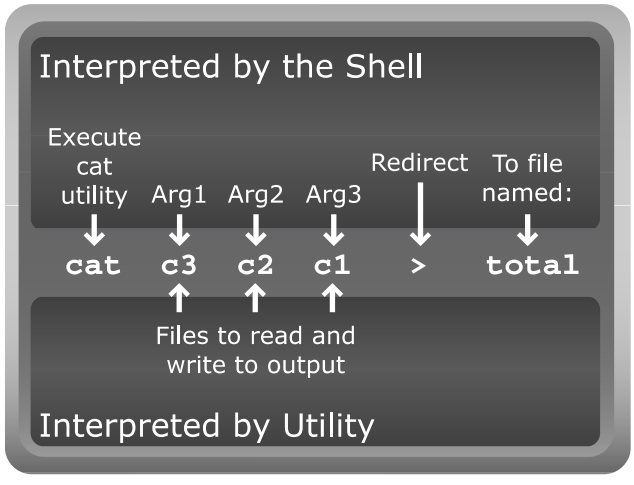
   cat *c3 c2 c1* **>** *total*

4. Examine the *total* file by entering:

   more *total*

The file named *total* consists of the contents of the file *c3* followed by the contents of the file *c2* followed by the lines from *c1.* All lines read by **cat** are written to the new file *total*.

When you look at the output, *total,* there is no way to tell where one file ends and another begins.



This command line instructs the shell to start a new process, pass the process three arguments (*c3*, *c2*, and *c1*), and then redirect the process's output to a new file *total.* Lastly, the shell instructs the process to run the **cat** utility. Once started, **cat** interprets all of its arguments as names of files to locate, open, read, and write to output, which is connected to the file *total*.

Locating Specific Lines in a File

We often need to locate the lines in a file that contain a word or string of characters.

1. Reexamine the file *total*:

   cat *total*

2. Select lines from the file that contain a target string:

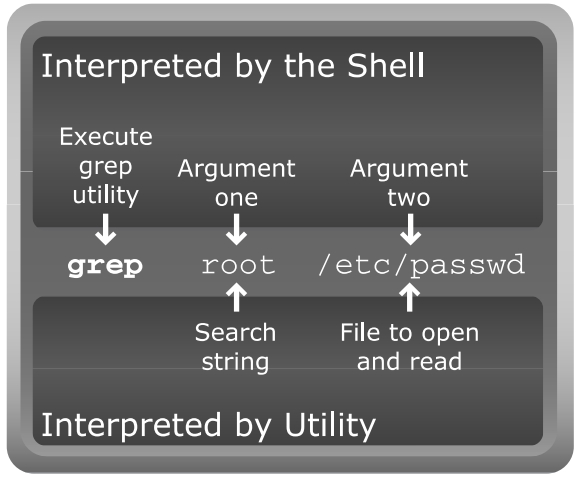
   grep *is total*

   Every line containing the character string *is* in the file *total* is selected and output.

3. Instruct **grep** to locate all lines in the file ***/etc/passwd*** that contain the string *root* by entering:

   grep *root* ***/etc/passwd***

   This command line asks the shell to run the **grep** utility and pass it two arguments. Many utilities, including **sort** and **rm**, interpret all arguments as files to be acted on. They sort or remove them all. Not **grep**. To the **grep** utility, the first argument is the *target string*, and all *other* arguments are files to be opened and searched. In this case, **grep** looks through the file ***/etc/passwd*** for lines that contain the target string *root* and selects those lines that match. It outputs only the matched lines. The original file is not affected.



4. Look in several files for lines that contain the string *is*:

grep *is c1 c2 c3 total*

grep interprets the string *is* as the target search string and all other arguments as files to open and search. The filename and matching lines are output.

3.2 Starting Additional Linux Terminal Sessions

In the UNIX and Linux environment, we can have multiple active sessions at the same time. For example, in UNIX we can log on from several terminals connected to the same system. These different sessions all belong to the same user, but are independent of one another. In Linux, one monitor and keyboard can be used for multiple login sessions.

Once you are logged on to your UNIX or Linux machine, in graphical or terminal mode, you can access other virtual terminals.

1. Press:

   CTRL-ALT-F2

   (While holding down the CTRL key and the ALT key, press the F2 key.)

   A new logon screen appears.
2. Log on again at this prompt.

   You are given a new shell prompt.
3. Type the following:

   who

   and press ENTER.

   The **who** program lists current users, and you are listed twice. The second column of output lists the terminal port that each user is employing. You are logged on through the initial terminal and again through a virtual terminal, the one that you accessed with F2.
4. Press:

   CTRL-ALT-F3

   A new logon prompt appears again.
5. Log on again.

Exiting a Virtual Terminal

To leave a second terminal we need to terminate the shell.

1. Enter:

   exit

2. The **exit** command ends the login session in this virtual terminal; it does not log you off of others.

In Linux environments, we can press CTRL-ALT-F2, CTRL-ALT-F3, CTRL-ALT-F4, through CTRL-ALT-F7 (and sometimes CTRL-ALT-F8) to create seven (sometimes eight) independent sessions on the same computer using the same monitor. When you log on, if it is a terminal window, it is probably terminal F1. If you log on and are placed in the graphical environment, it is using virtual terminal F7 or F8.

Locating the Graphical Virtual Terminal

You can toggle back and forth between active sessions by using the CTRL-ALT-F# command, where # is 1 through 7 or 8.

1. Press:

   CTRL-ALT-F7

   If you started a graphical desktop, this is where it is usually located, although it may be at F8.

2. If you logged on into a terminal, return to it, probably by pressing:

   CTRL-ALT-F1

The multiple login sessions available through the F keys allow us to log in a second time to test multi-user features, to have both graphical and character-based sessions running, and, as you will see, to kill processes that have frozen the terminal.

3.3 Managing Input and Output

Every process that is started has three defined communication locations or "doors." One is its input, one a place to write output, and a third to write any error messages. Usually, the output of utilities is by default directed to your terminal. You have redirected output to files and to other utilities. We can also tell the shell where to connect input.

Specifying a File as Input

There are two ways to get a utility to open a file and read it. You have been using one way, passing a filename as an argument to a utility. The utility simply goes out, locates the file, and reads it.

1. For example, enter the following command:

   sort *mypasswd*

   Here the shell is instructed to run the **sort** utility and pass it one argument, *mypasswd*. To the **sort** utility, the argument *mypasswd* is interpreted as a file to open and read. **sort** reads the lines from *mypasswd*, sorts the contents, and outputs the data to its output, which by default is connected to the workstation screen.

   Because **sort** has an argument, it interprets the argument as a file and does not read from its input "door." Only when there is no filename argument does **sort** read from its input.

2. Enter:

   who | sort

   In this case, **sort** does not have an argument, so it reads from its input which the shell connected to the output of **who**.

   We can also instruct the shell to connect a file to a utility's input.

3. Enter the following:

   sort < *mypasswd*

   The results are the same as with the **sort** *mypasswd* command. A sorted version of the file *mypasswd* is displayed on the screen. The < in this command is instruction to the *shell* to open the file *mypasswd* and connect the file to the *input* of **sort**. Because no argument is given to **sort**, the **sort** utility does not open a file. The shell opens the file itself and connects the file to the input to **sort**.

4. We can also connect files to the input of other utilities. For example, type:

   cat < *mypasswd*

   In this case, we are instructing the shell to connect the file *mypasswd* to the *input* of **cat**. Because no output destination is specified, the output is connected to the monitor by default. Thus, **cat** reads the contents of the file *mypasswd* and writes it to output, connected to the workstation screen.

COMMAND	INTERPRETATION
utility > *filename*	Shell connects the output of the utility to *filename*.
utility >> *filename*	Shell connects the output of the utility to the end of the file (appends).
utility < *filename*	Shell connects *filename* to the input of the utility.
utility1 | utility2	Shell connects the output of **utility1** to the input of **utility2**.

TABLE 3-3 Redirection Operators

The input redirection symbol is an important feature because, as you will soon see, several utilities are not programmed to open files. We must instruct the shell to open a file and connect it to the utility's input.

The redirection operators we have discussed thus far are described in Table 3-3.

Redirecting a File to *spell*'s Input

Many UNIX systems contain a spell check program that examines files for misspelled words. On some systems, the **spell** program only reads from input. It cannot open files. We must instruct the shell to open the file and connect it to the input of the process running **spell**.

1. Examine *users_on* for misspelled words with the following command:
   spell < *users_on*
2. If you are told the command is not found, enter:
   ispell -l < *users_on*

All strings in the file that **spell** does not find in the online dictionary file are viewed as misspelled words and are displayed on the screen. In this case, the shell opens the file *users_on* and connects it to the input of **spell**. Because you did not redirect the output, it is displayed on the screen.

Determining Where Utilities Read Input

Throughout these exercises, you have often specified filenames as arguments.

Enter the following representative command:
sort *total*

The character string *total* is given to **sort** as an argument. The **sort** program interprets *total* as a file to open, read, and sort. The output is a sorted version of the lines in the file *total*.

Employing the Default Input Source

The previous exercise raises a question: If no filename argument is provided, the utility reads from its input "door." If we do not tell the shell to connect a file or the output from another utility to the input, what does the utility find when it reads from its input?

1. Enter the following command. No filename is given as an argument to read, and no utility's output is redirected to the process.

   sort

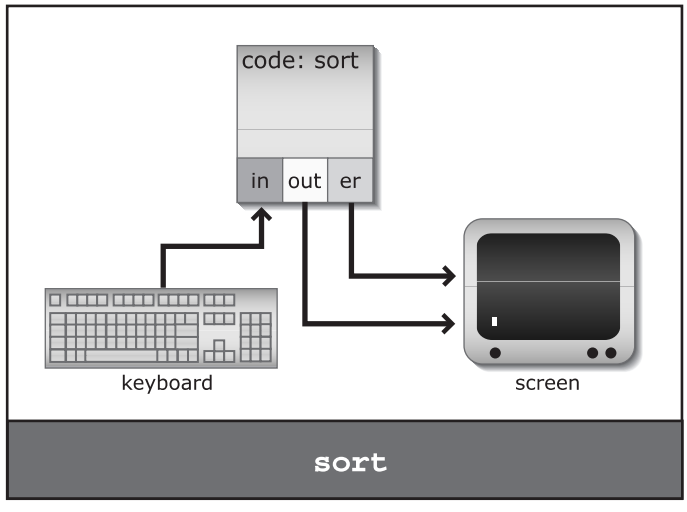
   The cursor moves to a new line. No shell prompt is displayed.
2. Enter the following lines:

```
hello
DDD
2
Hello
110
good-bye
```

3. Press ENTER to move the cursor to a new line, and then press ENTER again.
4. On a line by itself, press:

   CTRL-D (Hold down the CTRL key and press the D key one time.)

   A sorted version of the lines you just entered is displayed on the screen



When a new process is first started, the default input is the terminal keyboard. Output and error messages are initially connected to the terminal screen.

Because no filename arguments are specified in the preceding command, the **sort** utility reads from its input, which is connected to the default input source, your keyboard. You enter lines of text and then the end-of-file command, CTRL-D. This key combination indicates to the utility that there is no more input and that the utility can do its thing and then quit. The utility **sort** reads the lines you enter, sorts them, and writes to its output. Because the output of **sort** is not redirected, it is written to your screen.

5. Start another utility without specifying input or output destination:

   cat

6. Enter several lines.

   As you type a line, it is displayed on the screen. When you press ENTER, the line is passed to **cat**, which reads the line and writes it to output, the screen. The line appears as a duplicate below the line that you entered.

7. After entering several lines, go to a new line and press:

   CTRL-D

   After **cat** terminates, the shell displays a new prompt.

8. We can specify output redirection with the default input source. Enter the following:

   sort > *sort-test*

9. Enter several lines of text.

10. When you are finished, press ENTER to put the cursor on a line by itself. Then press:

    CTRL-D

11. Examine the contents of the new file:

    more *sort-test*

You instructed the shell to connect the output of **sort** to the new file *sort-test*, but you did not specify any input for **sort**. By default, input is connected to the keyboard if it is not redirected to another source. The **sort** utility read what you entered as input and wrote its output. Because you instructed the shell to connect the output of **sort** to the new file *sort-test* when **sort** wrote its output, it went to the new file.

Creating Text Files with *cat*

You will usually create text files using an editor such as the visual editor, **vi**, which is discussed in Chapter 4. However, you can quickly create small text files without first mastering an editor, by using one of several other utilities.

1. Type the following:

   cat > *first_file*

   and press ENTER.

   The cursor returns to the beginning of the next line. The shell does not display a new prompt. This command line instructs the shell to start the **cat** utility and to connect its output to the new file *first_file.* There is no request to redirect the input, so input is still connected to the default—your keyboard. You are no longer in communication with the shell; what you now type is read by the **cat** utility, which simply writes to output whatever it reads from input.
2. Type the following lines:

   ```
   This is a line of text in the first_file.
   ```
3. Press ENTER and then type:

   ```
   This is another.
   ```

   The **cat** utility reads your input and writes it to its output, which the shell connected to a new file named *first_file.*
4. To inform the **cat** utility that you have finished adding text, press ENTER to advance to a new line and then press:

   CTRL-D

   This CTRL-D (end-of-file, or EOF, character) tells **cat** there is no additional input. The **cat** utility terminates, and the shell displays another prompt.
5. From the shell, obtain a listing of your files using:

   ls

   The file named *first_file* is listed.
6. Examine the contents of *first_file* by typing:

   more *first_file*

   The *first_file* file consists only of the text you typed. No additional data about the file, such as the file's name or your name, is added to the file by the system. The file contains just the text you typed. The file's name is kept in the directory. The information about a file is in a system storage unit associated with your file.
7. Create another text file with another **cat** command:

   cat > *second_file*
8. Add some text, and return to the shell by pressing CTRL-D.
9. Obtain a listing of the files in your current directory with:

   ls

10. Examine the contents of *second_file* with:

more *second_file*

The file consists of the lines you entered as input to **cat**.

By default, the keyboard is connected to the input of **cat**. Whatever you type is read by **cat** from your keyboard and written to output, which is connected to the file. The **cat** utility is not very complicated; it simply reads input and writes output, making no modifications. The command **cat >** *filename* instructs the shell to connect the output from **cat** to the file *filename* and to execute the **cat** utility.

Managing Input and Output with Redirection

The role and effect of file input and output redirection symbols are summarized in Table 3-4.

1. Create a new file named *file2* with the following:

head ***-12 /etc/passwd*** > *file2*

2. Try each of the commands in Table 3-4 and confirm the data in the table.

Self Test 1

Answer the following, and then check your answers using the information within the chapter.

1. What is the effect of each of the following commands?

A. **sort** *file1 file2*

_____

B. **wc** *file1 file2*

_____

C. **grep** *file1 file2*

_____

D. **who | sort >** *abc*

_____

E. **cat** *file1 file2 file3*

_____

2. What command results in a reverse sorting of all lines in the */etc/passwd* file that contain a zero somewhere on the line?

_____

3. What would be in the file *file1* as a result of each of the following?
   A. **echo** *file1* **>>** *file1*
   B. **cat >** *file1*

_____

4. How can you access virtual terminal 3?

_____

5. What command reads its arguments and writes them to output?

_____

COMMAND	INPUT	OUTPUT	EFFECT
sort	Keyboard	Display screen	The **sort** utility receives no arguments, so it opens no file. Instead, **sort** reads from input, which is by default connected to the keyboard.
sort > *file1*	Keyboard	*file1*	Keyboard input is sorted, and output is connected to *file1.*
sort >> *file1*	Keyboard	*file1*	Keyboard input is sorted and its output is appended to the end of *file1.*
sort < *file2*	*file2*	Display screen	*file2* is opened by the shell and connected to the input of **sort**. The output is not redirected, but displayed on the screen.
sort *file1*	*file1*	Display screen	*file1* is passed as an argument to **sort**, which opens the file and reads its contents. Output connected to the screen.
sort < *file1* **>** *file3*	*file1*	*file3*	The shell connects *file1* to the input and *file3* to the output of **sort**. When **sort** runs, it reads from input, sorts the lines, and writes to output. The lines from the file *file1* are sorted and the output placed in *file3*.

TABLE 3-4 Passing Arguments and Opening Files

COMMAND	INPUT	OUTPUT	EFFECT
sort *file1* **>** *file4*	*file1*	file4	The shell passes *file1* as an argument to **sort** and connects the output of **sort** to the file *file4*. The **sort** utility opens *file1* and sorts the lines; the output goes to *file4.*

TABLE 3-4 Passing Arguments and Opening Files *(continued)*

3.4 Employing Special Characters in Command Lines

When we are communicating with the shell, our only tool is the keyboard. The characters available on the keyboard and words or tokens created by combining those characters constitute the entire language we can use to communicate with the shell. Many characters have special meaning to the shell, such as the redirect symbol, >. The shell interprets the > as instruction to connect the output of the previous utility to a file named right after the redirect. Likewise, the | is instruction to connect the output of one utility to the input of another utility.

This section introduces other special characters interpreted by the shell.

Replacing a Wildcard Character with Filenames

One special character to the shell is a *wildcard* character we can use when specifying filenames.

1. If you list several filenames after the **wc** utility on the command line, the **wc** utility examines all of the files. Enter this command:

   wc *total lost-days*

   The number of elements in each file is counted and output, along with the total for all files.

2. To have **wc** examine all files whose names begin with the letter *u*, enter:

   wc *u**

   The shell interprets the *u** as instruction to replace the string *u** on the command line with the names of all files in the current directory that start with the letter *u* and have zero or more additional characters following the *u* in their names. The shell then runs the **wc** utility, passing it all the arguments that it generated—the names of all files in the directory that were matched.

3. Confirm that it is the shell that is expanding (replacing the string with) the * into filenames by entering:

   echo *u**

   The shell replaces the string *u** with the filenames in the current directory that begin with the letter *u*. Those names are passed as arguments, this time to **echo**, which writes the arguments to output.

4. You can also have the shell list *all* of the files in your current directory as arguments to a command by typing:

   echo *

   The shell replaces the asterisk with the names of all the files in your directory and then executes **echo**, passing all the filenames it generated as arguments. The **echo** utility reads its arguments (the filenames) and writes them to output, which in this case is your screen.

5. Do a word count of all the elements of all the files in your directory by entering:

   wc *

   The shell replaces the * with the names of all files in the current directory and passes all the names as arguments to **wc**. The **wc** utility examines all files listed as arguments and displays output like this:

```
  8    39    190    junk
  9    29    175    mypasswd
  8    39    190    phon
  1     5     22    today
 12    59    310    total
 14    80    220    users_on
 52   251   1107    total
```

   The output from **wc** is a list of information pertaining to all input files, followed by a total of these counts.

NOTE: If you created a file named *total*, it is listed in alphabetical order among the other files. The *total* at the end of the **wc** output is the sum of the statistics for all files examined by the utility.

You can also have the shell pass all filenames as arguments to the **grep** utility. The **grep** utility searches for the target string in all the files in your directory.

6. Type the following:

   grep *is* *

This command line tells the shell to replace the asterisk with all the filenames listed in your current directory. The first argument passed to **grep** is a string of characters that **grep** interprets as the *target*. The remaining arguments **grep** interprets as the names of *files*. The **grep** utility then searches each line in all files listed as arguments for the target string of characters and outputs the lines that contain a match.

Accessing Shell Variables

We use variables in life all the time. We fill out forms such as:

Last name: _____

First name: _____

All of us, except Cher, have values in our memories for the last name and first name variables.

We can use specific characters to indicate which tokens in a sentence are variables. For example, consider:

I am $fname $lname. I live in $city. I was born in $birthplace.

When I read the previous line, I read:

I am John Muster. I live in Berkeley. I was born in Canton, Ohio.

To me, the value of the *fname* variable is *John*.

Everyone reads the line differently because everyone has different *values* for the variables that are identified with dollar signs. In this case, we interpret the **$** to mean, "Find the value of the variable that has the following name and replace both the dollar sign and variable name with the variable's value."

1. Ask the shell to evaluate a variable and pass its value to **echo** by entering:

   echo $*USER*

2. If there is no ***USER*** variable, try:

   echo $*LOGNAME*

   The **$** character has the same special meaning to the shell. It tells the shell to "locate the variable whose name follows, and replace this string with the variable's value." In the command you just entered, ***USER*** is the variable

that the shell evaluates, because it is preceded with a **$**. After replacing the variable and **$** with its value, your actual login name, the shell passes your name as an argument to **echo**. Then **echo** reads the argument and writes it to your screen. The important distinction here is that the shell passes your login name, not **$*USER***, to **echo**. The shell interprets the variable and passes its value, not its name.

Having the shell evaluate **$*USER*** and replacing the variable with its value can be very useful.

3. Enter:

   who | grep $*USER*

   The shell replaces the **$*USER*** with your login ID and then passes that value to **grep** as its first argument. To **grep**, the first argument is its search string. The line from the output of **who** that contains your login ID is selected by **grep** and output to your screen.

4. Have the shell evaluate some other variables by entering:

   echo my shell is **$*SHELL*** and my home is **$*HOME***

   The output on the screen is all the arguments, with variables replaced by their values.

   In this case, the shell evaluates two variables. The resulting values are passed to **echo** as arguments. The **echo** utility reads all its arguments and writes them to output. By default, the output is connected to your monitor. The value of the first variable, ***SHELL***, is the shell that is started up for you at login; the other variable, ***HOME***, is where your home directory is located on the system. Your shell obtained these variables and their values when you logged on. Your colleagues have their own variable values. The shell and all other programs you run are given these variables and these values.

   We can employ both a variable and the filename expansion on the same command line.

5. Enter:

   grep $*USER* *

   The shell replaces the variable ***USER*** with its value, the login name, which becomes argument one. The shell replaces the * with the names of all files in the current directory, which become arguments two, three, and so forth. **grep** interprets the your name as the search string and looks through all files for lines containing your login name.

6. The variable can be used to count the number of times you are logged on:

   who | grep $*USER* | wc -l

Listing Environment Variables

The shell program that interprets your commands is started as a process when you log on. The values of several variables are given to your particular shell so that your computing environment is appropriate. We can obtain a listing of those environmental variables.

↳ From the shell, type the following:

env | more

or

printenv | more

The output is a listing of some of the variables that are currently set for your shell. Among the many lines displayed, you should find something like the following.

For C shell users:

```
USER    forbes
SHELL   /bin/csh
HOME    /users1/programmers/forbes
PATH    /usr/ucb:/bin:/usr/bin:/usr/local:/lurnix/bin:/usr/new:.
```

For **bash** and Korn shell users:

```
HOME=/usr/home/nate
LOGNAME=nate
PATH=/usr/ucb:/bin:/usr/bin:/usr/local:/lurnix/bin:/usr/new:.
SHELL=/bin/ksh     or   (/bin/bash)
```

The variables and their values are essential to a functioning shell. Your output includes variables such as the following, but with values that are appropriate for your account.

- The ***user*** or ***USER*** or ***LOGNAME*** variable is your account name that you entered when you logged on.
- The ***shell*** or ***SHELL*** line indicates which of several shell programs is started at login to interpret the commands that you enter: **csh** is the C shell, **sh** is the Bourne shell, **ksh** is the Korn shell, **bash** is the **bash** shell, and **tcsh** is the **tcsh** shell. They all handle basic commands in essentially the same way, and for now it makes little difference which is running.
- The ***home*** or ***HOME*** variable is the location of your workspace or home directory.

- The ***path*** or ***PATH*** variable lists the directories where the shell looks to find UNIX utilities you request.

The subject of local and environment variables is explored in some detail in Chapter 9.

Instructing the Shell Not to Interpret Special Characters

The characters *, **!**, |, **>**, and **$** have special meaning to the shell. Sometimes we need to instruct the shell not to interpret special characters but to treat them as ordinary characters instead. There are several ways to tell the shell to turn off interpretation of special characters.

1. For instance, enter the following:

   echo *we can output a* ***** uninterpreted

   The output includes a literal * character.

   In response to this command, the shell does not expand the asterisk to match filenames, but passes it as a one-character argument to **echo**. It is not interpreted. The **echo** utility reads the asterisk as a one-character argument and outputs it, in this case, to your screen. When a special character is preceded by a backslash (\), the shell interprets that character as ordinary (lacking any special meaning). To put it the other way, the shell interprets the backslash character as instruction to treat whatever character follows as an ordinary character having no special meaning. Once the * is passed to **echo**, the * is just an asterisk that it outputs.

2. Enter the following:

   echo \$*HOME*

   The output is the literal string `$HOME` without the backslash. The **$** is *not* interpreted as instruction to evaluate the ***HOME*** variable, because it is preceded by a backslash. The shell interprets the **$** as just an ordinary dollar sign, and passes the string *$HOME* to the **echo** utility as an argument.

   In the previous commands, the arguments the shell gave to **echo** after it interpreted the ***** and **\$** did not include the backslash. When the shell interprets *****, it reads the \ as a specific instruction: Don't ascribe special meaning to the character that immediately follows. The only character that gets passed is the one character that follows the \ character. The backslash is not passed to **echo** because the shell interpreted it as instruction not to interpret the character that follows.

3. Try the following:

   echo \$*USER* * \$*USER*

   The output is **\$*USER*** * and your login name. The last **\$*USER*** is interpreted because there is no backslash in front of the **$** telling the shell not to interpret it.

   Likewise, we can use **echo** to place interpreted and not interpreted variables into a file.

4. Enter:

   echo *var1* **\$*USER* >** *test-interp*
   echo *var2* **\$*USER* >>** *test-interp*
   more *test-interp*

   The shell interprets the **$** in **\$*USER*** in the *var1* line, but does not interpret the **$** in the *var2* line. Note the second line in *test-interp* is:

   var2 ***$USER***

   The shell does not pass the backslash to **echo**.

5. Have **grep** look for strings in the *test-interp* file:

   grep \$*USER* *test-interp*

   The shell interprets the variable **\$*USER*** and passes the value of your login name to **grep** as the first argument. Then **grep** interprets the first argument (your login name) as the search string. The line with your login name is selected and output:

   grep \\\$*USER* *test-interp*

   This time the shell interprets the backslash as instruction *not* to interpret the very next character, the **$**, so the string **\$*USER*** is passed to **grep** as the first argument. The **grep** utility searches for the actual string **\$*USER*** in the file.

   The same is true with the filename expansion wildcard * character.

6. Enter:

   echo *u**
   echo *u**
   wc *u**
   wc *u**

The shell interprets the *u** as instruction to replace the characters *u** with all the filenames in the current directory that start with the letter *u*. The shell then passes the names as arguments. **echo** reads the arguments and writes them to output. **wc** interprets the arguments as filenames and counts the elements of each named file.

The shell interprets *u** as a *u* and an uninterpreted *. The argument to **echo** is *u**, which it displays. To **wc**, the *u** is the name of a file, a file that it cannot locate.

Not Interpreting ENTER

When you press ENTER at the end of a command line, you are signaling the end of the command. The shell interprets ENTER as a special character, one that indicates the end of the command to be executed, and starts processing. When we are entering a long command line, we often want to put part of it on a second line. However, as soon as we press ENTER, the command is executed. We need to be able to tell the shell not to interpret ENTER.

1. Enter the following command:

   **who > **

   and press ENTER

   The backslash instructs the shell not to interpret the character that immediately follows. Hence, ENTER is not interpreted. There is no end of the command. At this point, the shell has not been told to process the command line, because no real ENTER has been received. It waits for more input. In fact, what you have entered so far is not a complete command. The shell needs to redirect the output of **who** to a file, but the filename is not included.
2. Enter the filename:

   users2

   and press ENTER again.

   This time, ENTER is not preceded by a backslash. The shell interprets it as a real ENTER, signifying the end of the command line, which now happens to span two input lines. It is processed.
3. Confirm that the new file is created by entering:

   ls
   more *users2*
4. Examine the history list to see the previous command:

   history

   The **who >** *users_on* is one command even though it was entered on two lines.

When you want a command line to span more than one input line, precede the first line's ENTER with a backslash character to instruct the shell not to interpret ENTER's special meaning.

Not Interpreting Several Characters in a String

The backslash turns off interpretation for one character only, whatever single character follows.

We can turn off interpretation for more than one character.

1. Enter the following using single quotes:

   echo '$*USER* * $*USER*'
   echo '$*HOME* $*USER*'

   The output is the literal string of characters entered. When inside single quotes, the * and **$** are seen just as characters, so the shell does not expand the * to match filenames. The ***$HOME*** and ***$USER*** are not evaluated for the variable values. The arguments passed to **echo** are just the uninterpreted character strings.

   We can instruct the shell to interpret part of a line and not interpret other parts of the line.

2. Enter the following, paying careful attention to the single quotes:

   echo $*USER* '$*HOME* $*USER*' $*USER*

The portion of the command line inside single quotes is not interpreted, but passed as an argument to **echo** as is. The parts of the line not in quotes are interpreted and variable values are substituted for the ***$USER*** variables.

When the shell encounters the first single quote, the shell turns off interpretation of all special characters. The second single quote turns interpretation back on again. Any special characters inside the single quotes are not interpreted.

Creating Multiple Token Arguments

Because we can tell the shell not to interpret special characters inside single quotes, we can tell the shell not to interpret spaces.

1. Pass several arguments to **echo** with several spaces between them:

   echo *AA BB CC DD*

   The output does not include the multiple spaces:

   ```
   AA BB CC DD
   ```

   The shell interprets one or more spaces as separating the tokens on the command line. The shell interprets **echo** as the utility, and passes it four distinct arguments. It does not pass the whole line of words and spaces.

When **echo** reads its argument list, it outputs the first argument, then a single space, then the next argument, a space, and so forth.

2. Use quotes to tell the shell not to interpret special characters (including spaces):

   echo '*AA BB CC DD*'

   A single argument is passed to **echo**. The argument consists of the line as it was entered, including spaces, namely:

```
AA      BB      CC      DD
```

The shell does not interpret spaces as indicating different arguments, so it is all one argument. **echo** outputs the single argument, spaces and all.

Passing Complex Arguments

One of the most useful functions of modern computers is database management. The UNIX operating system provides several utilities that are used with database information. One of the most versatile is **awk.**

1. Type the following commands:

   ps -ef

   ps -ef | awk '{print $*1*}' | more

   The **awk** utility extracts the first field from each line of the output of **ps**. The output of **awk** is displayed on the screen.

2. Change the command line to instruct **awk** to select the second field. Enter:

   ps -ef | awk '{print $2}'

   This table describes the pieces of the command line:

COMMAND	INTERPRETATION
ps -ef	Instructs the shell to run the **ps** utility and pass it the argument **-ef**, which **ps** interprets as instruction to list all processes.
|	Instructs the shell to connect the output of **ps** to the input of the next utility, **awk**.
awk	Instructs the shell to run the **awk** utility.
' '	Instructs the shell not to interpret any special character between the single quotes, but to pass the enclosed characters as is to **awk** as an argument.
{print $*2*}	This is the contents of the quoted string that is passed to **awk**. The **awk** utility interprets this instruction as, "For every line of input, print out only the second field."

3. Have **awk** select more than one field by entering:

```
ps -ef | awk '{print $2, $1, $4}'
awk '{print $2, $1}' lost-days
```

The output is the second field, a space, and then the first field of all records (lines) in the file *lost-days*. This works because the shell does not interpret special characters such as the **$2**; rather, it passes the whole **{print $2, $1}** exactly as it is to **awk**, which interprets it as instructions.

The **awk** utility can be used to select and print specific fields, make calculations, and locate records by the value of specific fields. You will use it more extensively in Chapter 5.

Communicating with Processes

We instruct the shell to start processes. Usually the processes simply complete their tasks and then exit. We can also use control characters to send important signals to processes.

Signaling the End of File

We have used the control character CTRL-D to end input.

1. For example, enter:

```
wc
```

2. Add a few words and then press ENTER.
3. On the new line, press:

CTRL-D

The CTRL-D is the end-of-file (EOF) marker. When we are entering text from the keyboard to the input of a utility, we signal the end of our input by pressing CTRL-D. The **wc** utility counts the lines, words, and characters in whatever text that we enter until we press CTRL-D. The **wc** utility then displays the results and quits. Every file has a CTRL-D (EOF) character at the end to indicate where to stop reading.

Telling a Process to Quit

There are other important control characters.

1. Start **wc** again without input or arguments:

```
wc
```

2. Enter some text.

3. Instead of the usual end-of-file character, press:

   CTRL-C

   The **wc** program stops, and a shell prompt is displayed. However, no output from **wc** is displayed. CTRL-C is the interrupt signal, which kills the process. The end-of-file CTRL-D says, "End of input, do whatever you do with input and then exit," but the CTRL-C says, "Stop, put toys away, process nothing more, and be gone." No output is displayed.

   Which control character we use depends on the way the utility functions.

4. For example, enter:

   sleep *40*

   No new shell prompt is displayed. The **sleep** utility is counting 40 seconds.

5. Try to end the **sleep** process with:

   CTRL-D

   Because sleep is not **reading** CTRL-D the EOF has no meaning.

6. Kill the process by pressing:

   CTRL-C

The interrupt signal reaches the process and it terminates.

Sending a Process to the Background

Many of the processes running on the system are not associated with a particular user, but are important elements of the operating system. These processes are running *in the background* and are invisible to most users. This section examines how users can run processes in the background.

1. Type the following command:

   sleep *6*

   The shell runs **sleep** and gives it the number *6* as an argument. While **sleep** is counting to *6*, the shell waits. No new prompt is presented until **sleep** is finished.

2. Enter:

   sleep *60* **&**

   This command line tells the shell to run the **sleep** command in the background. The ampersand (**&**) at the end tells the shell to execute the whole command line, but instead of waiting until **sleep** is finished, the shell is to return a new shell prompt so that you can continue working.

3. Obtain a list of current programs by entering:

   ps

The **sleep** process is still running. When you execute this command, a number is displayed. This is the process ID number of the **ps** utility as it is executed. When the process is finished, a message is sent to the screen.

This feature allows us to run time-consuming programs in the background while we continue to work in the foreground on some other task. Obviously, with a command process that runs quickly, placing it in the background does not do much good, but there are times—such as when you are running long searches, database queries, and so forth—that it saves time and work.

Programming with Utilities

Thus far in these examples, communication with the shell has been interactive. We enter a command line; then the shell reads and processes whatever we enter from the keyboard. The shell will also read instructions that are placed in a file.

Creating a File of Commands

Until we examine the visual editor in the next chapter, we can use **cat** to read whatever we type and write it to a file.

1. Enter:

   cat > *commands-file*

2. Enter the following lines, pressing ENTER after each:

```
echo Hi $USER
date
cal
sleep 2
ps
echo Bye $USER
```

3. Conclude the input by pressing:

   ENTER

   and then on a new line press:

   CTRL-D

4. Make sure the file contents are as just described:

   more *commands-file*

If there is a mistake, remove the file with **rm** *commands-file* and re-create it. There is no simple way to modify a file until you learn to use the editor.

Instructing the Shell to Read a File

We can tell the current shell to read a file and execute each line in the file.

1. If you are in a **csh** or **tcsh** shell, enter:
   source *commands-file*
2. If you are in the **ksh** or **bash** or **sh** shell, use the "dot" command by entering:
   . *commands-file*

   The commands that are the contents of the file are executed one after the other.

Both the **source** and dot (**.**) commands instruct the current shell to read the file named as an argument and to execute every line in the file as though we just typed it in from the keyboard. In either case we refer to it as sourcing the file.

Self Test 2

Answer the following, and then check your answers using the information within the chapter.

1. What would be the contents of *file1* if each of the following were independently executed?
   A. **grep** *file1* * > *file1*
   B. **echo** **$*USER*** **$*HOME*** > *file1*

   _____

2. How do each of the following commands work? What is the result?
   A. **spell** *file1*

   _____

   B. **spell** < *file1*

   _____

   C. **cat** *file1* | **spell**

   _____

   D. **grep** '*Joan Heller*' *faculty*

   _____

   E. **grep** **\$*USER*** *file1*

   _____

3. What command instructs **awk** to output just the time that each user logged on?

   _____

4. What is the output of the command **echo** ***$PATH***?

   _____

3.5 Modifying the User Environment

One of the strengths of the UNIX operating system is its flexibility. The system allows us to customize a variety of programs to meet our individual needs. We can tailor how the shell behaves, and instruct the editor to include features. We can operate with a graphical desktop that includes color, icons, and startup programs of our own choice. Each user can tailor or modify all of these environmental features. This section explores a small part of the tailoring functionality.

Instructing the Shell Not to Overwrite Files

Thus far, we have used the **>** symbol to instruct the shell to connect the output of a utility to a new file. What happens when we redirect output to an existing file depends on the shell you are using and the value of a variable named ***noclobber***. As users, we make the decision whether the shell should overwrite existing files or not overwrite them.

1. To make sure that the shell is functioning in its "file clobbering" mode for this demonstration, enter the following commands.

   If you are in a **csh** or **tcsh** shell, enter:

   unset ***noclobber***

   If you are in the **ksh** or **bash** shell, enter:

   set +o ***noclobber***

   The **sh** shell always overwrites files when we redirect output to an existing file. We cannot have it do otherwise.

2. Create a new file and examine its contents by entering:

   ls > *test-list*
   cat *test-list*

3. Instruct the shell to put the output of **date** into the same file and examine the file:

date > *test-list*
cat *test-list*

The original contents of the file have been *replaced* or *overwritten* by the output of **date**.

When we tell the shell to redirect the output of a utility to a file, the shell creates the file *if it does not exist.* If there is a file by that name, the current contents are removed to make room for the new output.

4. Instruct the shell not to clobber files when redirecting output:

   In the **csh** or **tcsh** shells, enter:

   set ***noclobber***

   In the **ksh** or **bash** shells, enter:

   set -o ***noclobber***

5. Attempt to redirect output from another utility to the file:

   ls > *test-list*

   An error message is displayed.

6. To see whether shell variables such as ***noclobber*** are on or off, try the following:

   In the **csh** or **tcsh** shell, enter:

   set

   The list of variables currently set is displayed.

   In the **ksh** or **bash** shell, enter:

   set -o

A list of shell operational variables is displayed.

In a **csh** or **tcsh** shell, we turn ***noclobber*** on with **set** and off with **unset.** In the **ksh** and **bash** shells, we turn ***noclobber*** on with **set -o** and off with **set +o** commands, as follows:

SHELL	TURN *noclobber* ON	TURN *noclobber* OFF
csh, tcsh	**set *noclobber***	**unset *noclobber***
ksh, bash	**set -o *noclobber***	**set +o *noclobber***

Avoiding Accidental Logout

If you accidentally enter a CTRL-D to your login shell, you may be logged out. The end-of-file character says, "No more input; exit," so the shell exits.

1. Start a child shell process by entering:

   csh

2. List your processes:

   ps

   The child **csh** is listed among the processes.

3. Issue an end-of-file signal to your shell by pressing:

   CTRL-D

4. List your processes again:

   ps

   The **csh** shell is gone. Pressing CTRL-D instructs it to exit.

5. Start another child shell by entering one of the following:

   bash

   or

   ksh

6. Identify the current processes with:

   ps

   The child shell is among those listed.

7. Instruct the child shell to terminate by giving it the end-of-file signal:

   CTRL-D

   ps

   The child shell is no longer running.

8. We can tell the shells to ignore an end-of-file character. Enter one of the following commands.

   In the **csh** or **tcsh** shells:

   set *ignoreeof*

   In the **ksh** and **bash** shells:

   set -o *ignoreeof*

9. Now press CTRL-D

   You receive a message telling you to use **exit** or **logout**, not CTRL-D.

You will soon customize your account to have ***ignoreeof*** and ***noclobber*** set at all times. For now, enter each after you log on to protect yourself from accidental overwrite and accidental logout.

Changing the Prompt

Throughout this book, we have talked about the shell prompt. There are some standard shell prompts, shown in the following table. The prompt that your shell displays, like much of your user environment, can be modified.

PROMPT	SHELL
$	Bourne and Korn shells (**sh**, **bash**, and **ksh**)
%	C shells (**csh** and **tcsh**)
#	Any shell as ***root***

1. Look at the current variables:

   set | more

   If the list includes a variable named ***prompt***, you are interacting with a **csh** or **tcsh** shell. If the output of **set** includes a ***PS1*** variable, you are in a **ksh** or **bash** shell.

2. If you are using a **csh** or **tcsh** shell, type the following command:

   set *prompt*='*myname*** '**

   where *myname* is whatever you want the prompt to be.

3. If you are using the **sh**, **bash**, or **ksh** shells, type the following:

   ***PS1*=**'*myname* '

Your prompt is now reset. This "personalized" prompt remains set until you log out. Later you will learn more about setting up your computer environment, and you will have the opportunity to customize various aspects of your workspace permanently, such as the shell prompts and the shell's behavior.

3.6 Surveying Elements of a Functioning System

We have been examining the system utilities, processes, shell interactions, and parts of the filesystem. This section lifts up the hood to addresses questions such as, where on the system are the utilities? When I log in, why do I wind up in my home directory? Why can I read some files but not others? How can we put commands in a file and run them by reading the file?

Examining the Toolboxes That Contain the Utilities

Throughout this chapter, you have been issuing commands that call for the shell to execute a utility, but we have not examined where the programs are actually located.

1. Enter the following misspelled command:

   dzatte

2. The error message displayed on the screen is something like:

   ```
   Command Not Found
   ```

   Where is the shell looking before it reports "Command Not Found"? I ask my five-year-old to go upstairs and get a book that is either on the desk or the nightstand. She leaves and returns with the book. It must have been on the desk or on the nightstand. Had it been on the bed with large red arrows pointing at it, she would have returned with the error message:

   ```
   Book Not Found
   ```

   Just like the five-year-old, the shell looks only where it's told to look.

3. Ask the shell to display the value of the ***PATH*** variable:

   echo $*PATH*

   The variable ***PATH*** is something like:

   ```
   /bin:/usr/bin:/usr/local/bin:/usr/bin/x11:/usr/hosts
   ```

   This variable consists of a series of directories separated by colons. These are the "desk" or "nightstand" places that the shell checks for a utility when you ask for one to be executed. The shell looks first in the directory listed on the left, then the next, and so on.

   The ***/bin*** directory contains some of the utilities available on the system in the form of **bin**ary files.

4. Obtain a listing of the utilities in ***/bin***. (Note: The **/** is important; do not omit it.)

   ls */bin* | more

The **ls** utility outputs a list of the files in the directory ***/bin***. You may recognize some of these files—they are utilities you have already used, including **cat**, **rm**, and **ls**. These are some of the executable programs that you access when you type a command. As you saw when you examined your ***PATH***, the ***/bin*** directory is not the only directory that contains executable code. The list of directories (***PATH***) that your shell examines can be modified to include other directories.

Determining Where a Utility Is Located

The shell looks for utilities in the directories in the *PATH* variable and reports on their location.

Enter:

which *who*
which *xterm*
which *ls*
which *set*

If you ask for the location of a utility that you know exists, such as **set**, and the shell reports it cannot be found, that program, like **set**, is built into the shell. The shell cannot find it in the path because the code is not in a different utility file; rather, it is in the shell itself.

The search path is one of the features that make UNIX and Linux so flexible. When we want to add a new application or program, we can add it to one of the directories in the path and instantly give everyone access to it. Alternatively, we can create a new directory, put the application there, and modify the path variable for the users who should have access to the new application.

Examining the Elements of the Password File

The utilities consult many system files as they perform their jobs. When you log on, a program called **login** asks for your password and starts your shell. Your shell gets information such as ***USER*** and ***HOME*** so it can access the needed information about your account.

Your user ID number, probably your password, and other information about you reside in a file called ***/etc/passwd***. On a stand-alone system, the ***/etc/passwd*** file that resides on the machine contains information about all users. If a system is a part of a network that allows users to log on from any of several machines with the same password and login information, one of the network machines contains the complete password file information, and it serves the other machines as needed. This Network Information Service is called NIS. Whenever you or any other user logs on, your entry from either the local or the NIS network ***passwd*** file is consulted.

1. Examine the local password file by entering the following command (note the spelling of ***passwd***):

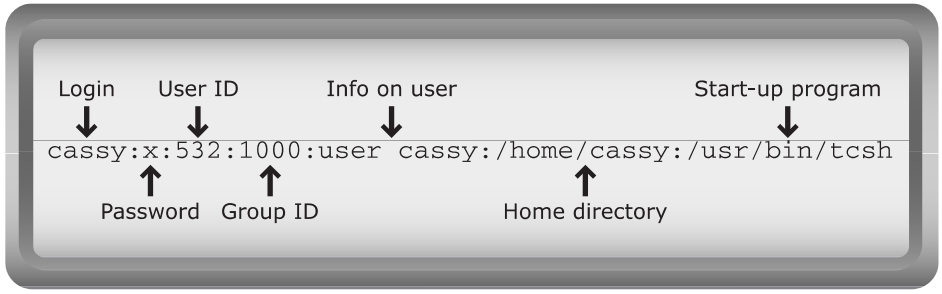
   more */etc/passwd*

2. If it is a very long file, page through the file by pressing several times:
   SPACEBAR
3. Locate the entry for your account.
4. Press **q** to stop and return to the shell.
   If your login name is not in the output, you are probably on a network that provides the passwords.
5. To see the ***passwd*** information from the network server, enter:
   ypcat *passwd* | more
6. Locate the entry for your account.
7. Press **q** to stop and return to the shell.
   We can use **grep** to select one record from the ***passwd*** data.
8. Depending on whether you are on a local or network served password file, type one of the following commands:
   grep *$USER* */etc/passwd*
   or
   ypcat *passwd* | grep *$USER*

In these versions, you are asking the shell to evaluate the variable ***USER*** and pass its value to **grep** as the search target. The **grep** utility then selects the line that contains your login name.

NOTE: If your login name record is output when you enter **grep */etc/passwd***, you are on a locally served system, and the ***/etc/passwd*** file should be used for password lookups throughout this text. If the **ypcat** command outputs your login name record, your system is using NIS and you should use the **ypcat *passwd*** form of commands to access password information.

The records in the ***/etc/passwd*** file consist of seven fields separated by colons. The general format is as follows:



The fields of the password file are described in the following table:

FIELD	INFORMATION
login	The login or name for your account.
password	Your encrypted password. (May be an *x* if the passwords are kept in a secure ***/etc/shadow*** file. May also be an *.)
uid	Your user ID, the unique number that is assigned to your account.
gid	Your group ID. Each user must be a member of at least one group. Every user who has the same number in this field as you have is in your group. You can share files with group members using permissions.
misc	Information about the user such as the user's full name. The miscellaneous field is often blank.
home	Your home directory. This is your current directory when you first log on.
Startup program	The program that is started when you log on—it is usually a shell such as the Bash shell (***/usr/bin/bash***) or the **tcsh** (***/bin/tcsh***) or the Korn shell (***/bin/ksh***), but it does not have to be a shell. It can be anything, including a data entry program or a menu for accessing your accounts at a bank.

Changing Your Password

One of the most important ways to protect the data in your account is to choose a secure but memorable password. This is not only convenient, but necessary for maintaining the security of everyone's data on the computer.

Before you begin the process of changing your password, decide on an appropriate new one. When choosing a password, there are several words you should avoid because they are easily guessed. It is unsafe to use any of the following:

- Your login name

- Any first or last name
- Your address
- A word listed in a dictionary in any language
- Obscenities
- Pop culture words

It is best to include both upper- and lowercase letters, and at least one numeral and at least one special character in addition to regular alphabetic characters.

With all these considerations, it can be difficult to create a password that we can remember and is secure. One way to formulate a memorable yet difficult-to-crack password is to use the first letters of every word in a sentence that has meaning. For example, if you enjoy the work of a particular author, your password might be:

```
MfaiMT!60
```

This looks difficult to remember. It *is* extremely difficult to crack—but for me it's easy to recall because it is the first letter of each word in the following sentence, followed by the year my daughter Cassy was born. Cassy was borne in 91.

My **f**avorite **a**uthor **i**s **M**ark **T**wain**!** and Cassy was born in 91.

or

```
{OwaiS19}
```

which stands for

{Our **w**edding **a**nniversary **i**s **S**eptember **19}**

When you have decided on a new password and are ready to change your current password, take the following steps:

1. Determine whether your system is running the Network Information Service (NIS).

   Once you have decided on a new password, type whichever of the following commands is appropriate.
2. If your system is running NIS, type:

   yppasswd
3. Otherwise, enter:

   passwd

4. You are prompted for your *current* password. To protect you, the program will not continue unless you identify yourself by correctly providing the current password.
5. Type your *current* password and press ENTER
   You are now prompted for a *new* password
6. Type your new password and press ENTER
   The program asks you to repeat the new password to make certain that you can type it correctly and can remember it.
7. Type your new password again and press ENTER.
   When the shell prompt returns with no error messages, your password has been changed. It might even confirm that all went well.

The **passwd** utility accomplishes tasks you cannot do. It actually changes a system file that you are not permitted to alter. Because you have that power when running the **passwd** utility, it grills you extensively to be sure you are legitimate and that you can remember your new password.

Forgetting Passwords

Sometimes it happens. The system administrator (***root***) cannot find your current password. It is located on the system only in the encrypted form. However, the ***root*** user is a 600-pound gorilla and can sit wherever she wishes, so the ***root*** user can *change* your password to a new one without knowing the original. See Chapter 13.

Modifying Permissions on Files

As the owner of a file, you determine who has permission to read the contents of the file or to change the contents of the file. If it is a command file, you can specify who can execute it. We can modify permissions only on files we own.

➥ List the files in the current directory with the long option:
ls -l

The **-l** option is interpreted by **ls** as instruction to provide a long listing of information about the file. The first field in the output, which consists of 10 character places, shows the permissions currently set for that file. In Chapter 9 you will explore setting file permissions in much more depth. For now, however, look at the permissions field. The very first character is a dash for

files and a *d* for directories. The remaining nine characters are a mix of dashes, and **r**, **w**, and **x** characters that show which permissions are currently granted.

Denying and Adding Read Permission on a File

All files have a set of permissions that determine who can do what with the file.

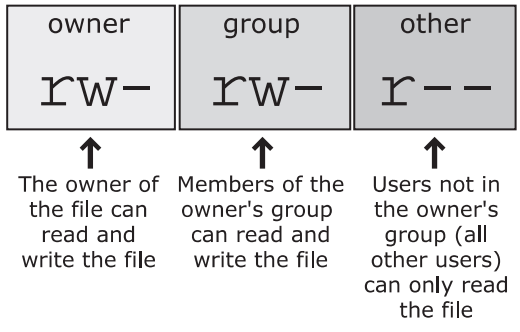
1. To view the permissions of the *users_on* file, type the following command:

   ls -l *users_on*

2. The output resembles the following:

```
-rw-rw-r--  1  cassy        453 Jul 18 11:17 users_on
```

   In this example, the first **rw-** indicates that *you* (the file's owner) have permission to **r**ead and **w**rite to the file. The second **rw-** indicates that other users who have been assigned to your group have **r**ead and **w**rite permissions for your file. The last three characters indicate the permission granted to all other users who are not in your group; in this case, they get **r**ead permission only.



   If you are the owner, you can change the permissions on a file to make the file inaccessible to all users, including yourself. Because you own the file, you can still change its permissions again at any time. No user can read or copy your file if you don't grant read permission.

3. Type the following command to remove read permission from the file *users_on*:

   chmod -r *users_on*

4. Examine the permissions field for *users_on* by typing:

   ls -l

Notice that although the previous permissions were read and write for you and your group, the new permissions only include a **w**. By placing the minus in front of the **r** in the **chmod** command, you removed read permission. The **chmod** command is used to **ch**ange the **mod**e of files.

5. Verify the state of the file's permissions by trying to display the file with the following command:

   cat *users_on*

   You immediately receive an error message saying that you do not have permission to read the file. Even though you own the file, if you deny yourself read permission, you can't read the file. You still own the file, however, so you can change the permissions again.

6. Return the read permission to the file with the following command:

   chmod +r *users_on*

   ls -l *users_on*

By preceding the **r** with a **+**, you add read permission.

In Chapter 10, we will examine what each permission (**r**, **w**, and **x**) controls for files and directories, as well as how to set the permissions specifically for owner, group, and other users.

Creating a Shell Script

You can use UNIX to program in a variety of formats and languages. The UNIX operating system gives programmers a number of programming tools that either are packaged with the system or that can be added.

One of the most basic and useful program tools is the shell itself. You have been using the shell as an interactive command interpreter. It is also a powerful programming environment.

1. Type the following command to create a new file:

   cat > *new_script*

2. Type the following lines:

```
echo  Your  files  are
ls
echo  today  is
date
echo  Current processes are:
ps
```

3. Press ENTER to move the cursor to a new line.

4. Press CTRL-D

   At this point, the file *new_script* contains a series of shell commands.

5. Examine the file to be certain it is correct:

   cat *new_script*

   If there are errors, remove the file *new_script* and return to step 1 to create it again.

6. Try to run the script by entering its name:

   new_script

   It does not run when we enter its name. You could source it and have the current shell read the file and execute its contents as we did earlier. However, the goal is to enter the filename and have it executed.

7. Display the permissions of the file by entering:

   ls -l *new_script*

   The permissions indicate that the file is not *executable.* To run the script by simply calling its name, you must grant yourself execute permission.

8. Type the following command to make *new_script* executable:

   chmod +x *new_script*

9. To see the new permissions, enter:

   ls -l

   You now have execute permission, as well as read and write permissions for the file.

10. Execute the new script by typing its name:

    new_script

11. If you receive an error message such as:

    ```
    Command not found
    ```

    type the following:

    ./new_script

    This command line tells the shell exactly where to find the shell script, *new_script*, in your current directory known as "dot."

All the commands that you typed into the file are executed, and their output is sent to the screen.

The output from **ps** indicates that a new process was started that ran the script. Your shell executed this child process to run the commands inside the script.

The steps to create and use a shell script are:

1. Create a file of shell commands.
2. Make the file executable with **chmod**.
3. Execute the file by entering the script name.

When we execute a script, the shell that is reading the script follows those instructions. It executes each line of the script as though it were a line you entered at the keyboard. All utilities in the script are executed.

In an earlier exercise, you told your current shell to read the script and execute the contents of the file with the **source** command. In this case, you issue the script name, which your current shell interprets as instruction to start a child process to read the script. Read permission is enough when you source a script. Execute permission is needed if you start a child shell to execute its contents. You will create many scripts in later chapters.

Self Test 3

Answer the following, and then check your answers at the end of the chapter.

1. When you enter the following command, what happens and why?
   cat > *dog*

   _____

2. What results from the following commands?
   A. **who** | **grep** $USER

   _____

   B. **grep \$*HOME*** *file1*

   _____

   C. **echo u* >>** *file1*

   _____

D. **echo** 'u*' **>>** *file1*

_____

E. **ps**

_____

F. **.** ***fileA***

_____

G. **set** ***noclobber***

_____

H. **set -o**

_____

I. **chmod +x** *file2*

_____

3. How can we change the prompt to be "Next?" in both families of shells (C shells and Korn shells)?

   _____

4. What data is in each field in the *passwd* file?

   _____: _____: _____: _____: _____: _____: _____

5. What command instructs the shell not to accept CTRL-D as a signal to log off?

   _____

Chapter Review

Use this section to review the content of this chapter and test yourself on your knowledge of the concepts.

Chapter Summary

UNIX is a multiuser, multitasking operating system. It includes numerous utilities that can be linked together for efficiency. UNIX is a complex, powerful, and occasionally unusual operating system. In this chapter, you examined the fundamental commands and concepts:

- Utilities are essential tools for accomplishing work in Linux/UNIX.
- The **sort** utility sorts lines from input or from a filename argument. It sorts in ASCII order by default, or in reverse order if given the **-r** option as an argument.

- The **wc** utility counts lines, words, and characters, by default. It outputs only lines if given the **-l** option, words if given **-w,** and only characters if given the **-c** option. Options can be combined to output more than one count.
- The **grep** utility interprets its first argument as a target search string and all other arguments as files to examine. **grep** searches through all lines of input for the target string. If the string is on the line, the line is output; otherwise, it is ignored.
- The shell redirects input and output for processes running utilities:

|	Connects the output from the previous utility to the input of the next.
>	Connects the output of the previous utility to a file. The file's name follows the redirection character.
>>	Connects the output of the previous utility to the end of the file whose name follows; thus, the output is appended to the end of the file.
<	Connects the file whose name follows the redirection to the input of the previous utility.

- The shell interprets the **$** as specifying a variable such as ***$USER*** and ***$HOME***. The shell replaces both the **$** and the *variable name* with its value.
- The * character is a special character to the shell which it interprets as a "wildcard" for matching filenames in the current directory. It is a wildcard such that *w**** is interpreted to mean all filenames that start with a *w* followed by any number of any characters. The shell replaces the string that includes the * on the command line with all matching filenames.
- The shell interprets the \ as instruction to turn off the interpretation of the special meaning for whatever character follows.
- If special characters are between single quotes, the shell does not interpret the special meaning.
- CTRL-D is the end-of-file character (EOF), and CTRL-C is the interrupt. When a process is sent the end-of-file signal, it completes whatever processing it is programmed to do on input data and then dies. With an interrupt, the process just terminates.
- We can tell the shell to protect existing files and not overwrite them when we use the **>** redirect by setting ***noclobber***:
  - C shell family: **set *noclobber***
  - Bash, Korn shell family: **set -o *noclobber***

- We can change the password using the **passwd** or **yppasswd** commands.
- Permissions determine who has read, write, and execute access to a file. The **chmod** command takes arguments such as **-r** as instruction to remove read permissions.
- When we create a file of shell commands, we can have the *current* shell execute the commands that are its contents by entering:
  - C shell family: **source** *file*
  - Bash, Korn shell family: **.** *file*
- We can have a child shell run the contents of a script by first changing the permissions on the script file to include execute and read permissions, then enter the filename as a utility on the command line.

Assignment

What Commands Accomplish the Following?

1. What options to **ls** accomplish the following?
   - **A.** Lists all files, including dot files._____
   - **B.** Provides a long listing of information about the files._____
   - **C.** Identifies directories with a /._____
2. What command line outputs a number that is the total number of files, including dot files, listed in the current directory?

   _____
3. What command obtains information about the **ps** utility?

   _____
4. What command sorts the contents of all files in the current directory with names beginning with the letter *f*?

   _____
5. In the following commands, how does each utility interpret each of the arguments:
   - **A. echo** *dogfish*_____
   - **B. rm** *dogfish*_____
   - **C. grep** *dogfish*_____
6. What command creates a new file called *all* consisting of the lines in the files *file1*, *file2*, and *file3*?

   _____

7. Explain what the following command lines accomplish:
   A. **grep** '*Linux is fun*' *_____
   B. **sort** > *file4*_____
   C. **grep** *$USER /etc/passwd*_____
   D. **echo** *\$PATH $PATH*_____
   E. **grep** '*1 2 3*' *f**_____
   F. **set -o EOF**_____

Fill-in

8. What is the difference between CTRL-D and CTRL-C?

   _____

9. Assume the contents of a file are the following series of shell commands:

   date
   ps -ef

   A. If you want to source the file, what permissions are needed?

      _____

   B. If you wish to execute the script simply by calling its name, what permission is needed?

      _____

   C. Which process executes the contents when you source a script?

      _____

   D. Which process executes the contents when you execute the script by calling its name?

      _____

Project

1. Write a command line that outputs the first, second, third, and eighth fields of the output of **ps -ef**.

   _____

2. Create a script named *allproc* that outputs the first, second, third, and eighth fields of the output of **ps -ef**. How would you make it executable? How would you run it?

   _____

COMMAND SUMMARY

passwd	Changes the user's password.
ls	**Ls**ts the contents of the current directory.
ls -l	Outputs a **l**ong listing of the contents of the current directory with one file or directory per line.
FILE DISPLAYING UTILITIES	
cat file1 file2	Con**cat**enates *file1* and *file2*. (Outputs *file1,* then *file2*.)
grep word filename	Searches for lines containing a particular *word* (or pattern) in *filename*.
wc filename	**c**ounts the lines, **w**ords, and characters in *filename*.
DATABASE UTILITIES	
awk '{print $#}' file	Prints the #th field of *file*.
DATA PRODUCING AND EXAMINING UTILITIES	
grep word filename	Searches for lines containing a particular *word* (or pattern) in *filename*.
sort filename	Displays the lines in *filename* in sorted order.
spell filename	Checks the spelling in *filename*.
REDIRECTION OF INPUT AND OUTPUT	
utility < filename	Makes *filename* the input for **utility**.
utility > filename	Sends the output of **utility** to *filename*.
utility1 | utility2	Makes the output of **utility1** the input of **utility2**.
FILE PERMISSIONS	
chmod -r filename	Removes permission to read *filename*.
chmod +r filename	Gives permission to read *filename*.
chmod +x filename	Grants execute permission on the file.
chmod -x filename	Removes execute permission on the file.
SHELL PROGRAMMING	
set	Lists the variables that are set for your shell and their values. In C shell, lists local variables. In Korn shell, lists local and environment variables.
env	Lists environment variables.

REDIRECTION OF INPUT AND OUTPUT	
printenv	Lists environment variables.
$var	Evaluates a variable, *var*.
*	Expands to match filenames.
\	Interprets the next character as an ordinary character without special meaning.
' '	Turns off interpretation of all characters between the single quotes. They are seen as ordinary characters.
scriptname	Executes the commands in the file ***scriptname***.
SETTING THE USER ENVIRONMENT	
***set* prompt = ' string '**	In the C and **tsch** shells, makes *string* the new prompt.
PS1=' string '	In the Bourne, **bash**, or Korn shell, makes *string* the new prompt.
PROCESS MONITORING	
ps	Displays the current processes for this login session.
ADDITIONAL UTILITIES	
clear	Clears the terminal screen.
echo	Reads arguments and writes them to output.
***which* utility-name**	Reports the location of *utility-name*.
ypcat	Reads network database files the way that **cat** reads local files.