

MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

CLASS : I MCA

SUBJECT CODE : 23PCA12

SUBJECT NAME : LINUX AND SHELL PROGRAMMING

SYLLABUS

UNIT -1

Basic bash Shell Commands: Interacting with the shell-Traversing the file system-Listing files and directories-Managing files and directories-Viewing file contents. **Basic Script Building:**Using multiple commands-Creating a script file-Displaying messages-Using variables-Redirecting input and output-Pipes-Performing math-Exiting the script. **Using Structured Commands:**Working with the if-then statement-Nesting ifs-Understanding the test command-Testing compound conditions-Using double brackets and parentheses-Looking at case.

(Book-1, Chapters: 3, 11, and 12)

Basic bash Shell Commands

IN THIS CHAPTER

Interacting with the shell

Using the bash manual

Traversing the filesystem

Listing files and directories

Managing files and directories

Viewing file contents

The default shell used in many Linux distributions is the GNU bash shell. This chapter describes the basic features available in the bash shell, such as the bash manual, tab auto-completion and how to display a file's contents. You will walk through how to work with Linux files and directories using the basic commands provided by the bash shell. If you're already comfortable with the basics in the Linux environment, feel free to skip this chapter and continue with Chapter 4 to see more advanced commands.

Starting the Shell

The GNU bash shell is a program that provides interactive access to the Linux system. It runs as a regular program and is normally started whenever a user logs in to a terminal. The shell that the system starts depends on your user ID configuration.

The `/etc/passwd` file contains a list of all the system user accounts, along with some basic configuration information about each user. Here's a sample entry from a `/etc/passwd` file:

```
christine:x:501:501:Christine Bresnahan:/home/christine:/bin/bash
```

Each entry has seven data fields, with fields separated by colons. The system uses the data in these fields to assign specific features for the user. Most of these entries are discussed in more detail in Chapter 7. For now, just pay attention to the last field, which specifies the user's shell program.

NOTE

Though the focus is on the GNU bash shell, additional shells are reviewed in this book. Chapter 23 covers working with alternative shells, such as dash and tcsh.

In the earlier `/etc/passwd` sample entry, the user `christine` has `/bin/bash` set as her default shell program. This means when `christine` logs into the Linux system, the bash shell program is automatically started.

Although the bash shell program is automatically started at login, whether a shell command line interface (CLI) is presented depends on which login method is used. If a virtual console terminal is used to log in, the CLI prompt is automatically presented, and you can begin to type shell commands. However, if you log into the Linux system via a graphical desktop environment, you need to start a graphical terminal emulator to access the shell CLI prompt.

Using the Shell Prompt

After you start a terminal emulation package or log in to a Linux virtual console, you get access to the shell CLI *prompt*. The prompt is your gateway to the shell. This is the place where you enter shell commands.

The default prompt symbol for the bash shell is the dollar sign (`$`). This symbol indicates that the shell is waiting for you to enter text. Different Linux distributions use different formats for the prompt. On this Ubuntu Linux system, the shell prompt looks like this:

```
christine@server01:~$
```

On the CentOS Linux system, it looks like this:

```
[christine@server01 ~]$
```

Besides acting as your access point to the shell, the prompt can provide additional helpful information. In the two preceding examples, the current user ID name, `christine`, is shown in the prompt. Also, the name of the system is shown, `server01`. You learn later in this chapter about additional items shown in the prompt.

TIP

If you are new to the CLI, keep in mind that, after you type in a shell command at the prompt, you need to press the Enter key for the shell to act upon your command.

The shell prompt is not static. It can be changed to suit your needs. Chapter 6, “Using Linux Environment Variables,” covers modifying your shell CLI prompt configuration.

Think of the shell CLI prompt as a helpmate, assisting you with your Linux system, giving you helpful insights, and letting you know when the shell is ready for new commands. Another helpful item in the shell is the bash Manual.

Interacting with the bash Manual

Most Linux distributions include an online manual for looking up information on shell commands, as well as lots of other GNU utilities included in the distribution. You should become familiar with the manual, because it's invaluable for working with commands, especially when you're trying to figure out various command line parameters.

The `man` command provides access to the manual pages stored on the Linux system. Entering the `man` command followed by a specific command name provides that utility's manual entry. Figure 3-1 shows an example of looking up the `xterm` command's manual pages. This page was reached by typing the command **`man xterm`**.

FIGURE 3-1

Manual pages for the `xterm` command

```

XTERM(1)                X Window System                XTERM(1)
NAME
    xterm - terminal emulator for X
SYNOPSIS
    xterm [-toolkitoption ...] [-option ...] [shell]
DESCRIPTION
    The xterm program is a terminal emulator for the X Window System. It
    provides DEC VT102/VT220 and selected features from higher-level termi-
    nals such as VT320/VT420/VT520 (VTxxx). It also provides Tektronix
    4014 emulation for programs that cannot use the window system directly.
    If the underlying operating system supports terminal resizing capabili-
    ties (for example, the SIGWINCH signal in systems derived from 4.3bsd),
    xterm will use the facilities to notify programs running in the window
    whenever it is resized.

    The VTxxx and Tektronix 4014 terminals each have their own window so
    that you can edit text in one and look at graphics in the other at the
    same time. To maintain the correct aspect ratio (height/width), Tek-
    tronix graphics will be restricted to the largest box with a 4014's
    aspect ratio that will fit in the window. This box is located in the
    upper left area of the window.

    Although both windows may be displayed at the same time, one of them is
    considered the "active" window for receiving keyboard input and termi-
    nal output. This is the window that contains the text cursor. The
    active window can be chosen through escape sequences, the "VT Options"
Manual page xterm(1) line 1 (press h for help or q to quit)

```

Notice the `xterm` command DESCRIPTION paragraphs in Figure 3-1. They are rather sparse and full of technical jargon. The bash manual is not a step-by-step guide, but instead a quick reference.

TIP

If you are new to the bash shell, you may find that the man pages are not very helpful at first. However, get into the habit of using them, especially to read the first paragraph or two of a command's `DESCRIPTION` section. Eventually, you will learn the technical lingo, and the man pages will become more helpful to you.

When you use the `man` command to view a command's manual pages, they are displayed with something called a *pager*. A pager is a utility that allows you to page through displayed text. Thus, you can page through the man pages by pressing the spacebar, or you can go line by line using the Enter key. In addition, you can use the arrow keys to scroll forward and backward through the man page text (assuming that your terminal emulation package supports the arrow key functions).

When you are finished with the man pages, press the `q` key to quit. When you quit the man pages, you receive a shell CLI prompt, indicating the shell is waiting for your next command.

TIP

The bash manual even has reference information on itself. Type `man man` to see manual pages concerning the man pages.

The manual page divides information about a command into separate sections. Each section has a conventional naming standard as shown in Table 3-1.

TABLE 3-1 The Linux man Page Conventional Section Names

Section	Description
Name	Displays command name and a short description
Synopsis	Shows command syntax
Configuration	Provides configuration information
Description	Describes command generally
Options	Describes command option(s)
Exit Status	Defines command exit status indicator(s)
Return Value	Describes command return value(s)
Errors	Provides command error messages
Environment	Describes environment variable(s) used
Files	Defines files used by command
Versions	Describes command version information

Conforming To	Provides standards followed
Notes	Describes additional helpful command material
Bugs	Provides the location to report found bugs
Example	Shows command use examples
Authors	Provides information on command developers
Copyright	Defines command code copyright status
See Also	Refers similar available commands

Not every command's man page has all the section names described in Table 3-1. Also, some commands have section names that are not listed in the conventional standard.

TIP

What if you can't remember the command name? You can search the man pages using keywords. The syntax is `man -k keyword`. For example, to find commands dealing with the terminals, you type `man -k terminal`.

In addition to the conventionally named sections for a man page, there are man page section areas. Each section area has an assigned number, starting at 1 and going to 9; they are listed in Table 3-2.

TABLE 3-2 The Linux man Page Section Areas

Section Number	Area Contents
1	Executable programs or shell commands
2	System calls
3	Library calls
4	Special files
5	File formats and conventions
6	Games
7	Overviews, conventions, and miscellaneous
8	Super user and system administration commands
9	Kernel routines

Typically, the man utility provides the lowest numbered content area for the command. For example, looking back to Figure 3-1 where the command **man xterm** was entered, notice that in the upper-left and upper-right display corners, the word `XTERM` is followed by a number in parentheses, (1). This means the man pages displayed are coming from content area 1 (executable programs or shell commands).

Occasionally, a command has man pages in multiple section content areas. For example, there is a command called `hostname`. The man pages contain information on the command as well as an overview section on system hostnames. To see the pages desired, you type **man section# topic**. For the command's man pages in section 1, type **man 1 hostname**. For the overview man pages in section 7, type **man 7 hostname**.

You can also step through an introduction to the various section content areas by typing **man 1 intro** to read about section 1, **man 2 intro** to read about section 2, **man 3 intro** to read about section 3, and so on.

The man pages are not the only reference. There are also the information pages called info pages. You can learn about the info pages by typing **info info**.

In addition, most commands accept the `-help` or `--help` option. For example, you can type **hostname -help** to see a help screen. For more information on using help, type **help help**. (See a pattern here?)

Obviously, several helpful resources are available for reference. However, many basic shell concepts still need detailed explanation. In the next section, we cover navigating through the Linux filesystem.

Navigating the Filesystem

When you log into the system and reach the shell command prompt, you are usually placed in your home directory. Often, you want to explore other areas in the Linux system besides just your home directory. This section describes how to do that using shell commands. To start, you need to take a tour of just what the Linux filesystem looks like so you know where you are going.

Looking at the Linux filesystem

If you're new to the Linux system, you may be confused by how it references files and directories, especially if you're used to the way the Microsoft Windows operating system does that. Before exploring the Linux system, it helps to have an understanding of how it's laid out.

The first difference you'll notice is that Linux does not use drive letters in pathnames. In the Windows world, the physical drives installed on the computer determine the pathname

of the file. Windows assigns a letter to each physical disk drive, and each drive contains its own directory structure for accessing files stored on it.

For example, in Windows you may be used to seeing the file paths such as:

```
c:\Users\Rich\Documents\test.doc
```

The Windows file path tells you exactly which physical disk partition contains the file named `test.doc`. For example, if you saved `test.doc` on a flash drive, designated by the J drive, the file path would be `J:\test.doc`. This path indicates that the file is located at the root of the drive assigned the letter J.

This is not the method used by Linux. Linux stores files within a single directory structure, called a *virtual directory*. The virtual directory contains file paths from all the storage devices installed on the computer, merged into a single directory structure.

The Linux virtual directory structure contains a single base directory, called the root. Directories and files beneath the root directory are listed based on the directory path used to get to them, similar to the way Windows does it.

TIP

You'll notice that Linux uses a forward slash (/) instead of a backward slash (\) to denote directories in file paths. The backslash character in Linux denotes an escape character and causes all sorts of problems when you use it in a file path. This may take some getting used to if you're coming from a Windows environment.

In Linux, you will see file paths similar to the following:

```
/home/Rich/Documents/test.doc
```

This indicates the file `test.doc` is in the directory `Documents`, under the directory `rich`, which is contained in the directory `home`. Notice that the path doesn't provide any information as to which physical disk the file is stored on.

The tricky part about the Linux virtual directory is how it incorporates each storage device. The first hard drive installed in a Linux system is called the *root drive*. The root drive contains the virtual directory core. Everything else builds from there.

On the root drive, Linux can use special directories as *mount points*. Mount points are directories in the virtual directory where you can assign additional storage devices. Linux causes files and directories to appear within these mount point directories, even though they are physically stored on a different drive.

Often system files are physically stored on the root drive. User files are typically stored on a separate drive or drives, as shown in Figure 3-2.

FIGURE 3-2
The Linux file structure

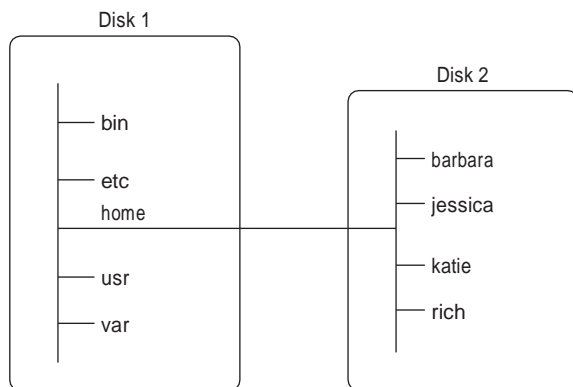


Figure 3-2 shows two hard drives on the computer. One hard drive is associated with the root of the virtual directory (indicated by a single forward slash). Other hard drives can be mounted anywhere in the virtual directory structure. In this example, the second hard drive is mounted at the location `/home`, which is where the user directories are located.

The Linux filesystem structure originally evolved from the Unix file structure. In a Linux filesystem, common directory names are used for common functions. Table 3-3 lists some of the more common Linux virtual top-level directory names and their contents.

TABLE 3-3 Common Linux Directory Names

Directory	Usage
<code>/</code>	root of the virtual directory, where normally, no files are placed
<code>/bin</code>	binary directory, where many GNU user-level utilities are stored
<code>/boot</code>	boot directory, where boot files are stored
<code>/dev</code>	device directory, where Linux creates device nodes
<code>/etc</code>	system configuration files directory
<code>/home</code>	home directory, where Linux creates user directories
<code>/lib</code>	library directory, where system and application library files are stored
<code>/media</code>	media directory, a common place for mount points used for removable media
<code>/mnt</code>	mount directory, another common place for mount points used for removable media
<code>/opt</code>	optional directory, often used to store third-party software packages and data files

<code>/proc</code>	process directory, where current hardware and process information is stored
<code>/root</code>	root home directory
<code>/sbin</code>	system binary directory, where many GNU admin-level utilities are stored
<code>/run</code>	run directory, where runtime data is held during system operation
<code>/srv</code>	service directory, where local services store their files
<code>/sys</code>	system directory, where system hardware information files are stored
<code>/tmp</code>	temporary directory, where temporary work files can be created and destroyed
<code>/usr</code>	user binary directory, where the bulk of GNU user-level utilities and data files are stored
<code>/var</code>	variable directory, for files that change frequently, such as log files

The common Linux directory names are based upon the Filesystem Hierarchy Standard (FHS). Many Linux distributions maintain compliance with FHS. Therefore, you should be able to easily find files on any FHS-compliant Linux systems.

NOTE

The FHS is occasionally updated. You may find that some Linux distributions are still using an older FHS standard, while other distributions only partially implement the current standard. To keep up to date on the FHS standard, visit its official home at <http://www.pathname.com/fhs/>.

When you log in to your system and reach a shell CLI prompt, your session starts in your home directory. Your home directory is a unique directory assigned to your user account. When a user account is created, the system normally assigns a unique directory for the account (see Chapter 7).

You can move around the virtual directory using a graphical interface. However, to move around the virtual directory from a CLI prompt, you need to learn to use the `cd` command.

Traversing directories

You use the change directory command (`cd`) to move your shell session to another directory in the Linux filesystem. The `cd` command syntax is pretty simplistic: `cd destination`.

The `cd` command may take a single parameter, *destination*, which specifies the directory name you want to go to. If you don't specify a destination on the `cd` command, it takes you to your home directory.

Part I: The Linux Command Line

The *destination* parameter can be expressed using two different methods. One method is using an absolute directory reference. The other method uses a relative directory reference.

The following sections describe each of these methods. The differences between these two methods are important to understand as you traverse the filesystem.

Using absolute directory references

You can reference a directory name within the virtual directory system using an *absolute directory reference*. The absolute directory reference defines exactly where the directory is in the virtual directory structure, starting at the root. Think of the absolute directory reference as the full name for a directory.

An absolute directory reference always begins with a forward slash (/), indicating the virtual directory system's root. Thus, to reference user binaries, contained within the `bin` directory stored within the `usr` directory, you would use an absolute directory reference as follows:

```
/usr/bin
```

With the absolute directory reference, there's no doubt as to exactly where you want to go. To move to a specific location in the filesystem using the absolute directory reference, you just specify the full pathname in the `cd` command:

```
christine@server01:~$ cd /usr/bin
christine@server01:/usr/bin$
```

Notice in the preceding example that the prompt originally had a tilde (~) in it. After the change to a new directory occurred, the tilde was replaced by `/usr/bin`. This is where a CLI prompt can help you keep track of where you are in the virtual directory structure. The tilde indicates that your shell session is located in your home directory. After you move out of your home directory, the absolute directory reference is shown in the prompt, if the prompt has been configured to do so.

NOTE

If your shell CLI prompt does not show your shell session's current location, then it has not been configured to do so. Chapter 6 shows you how to make configuration changes, if you desire modifications to your CLI prompt.

If your prompt has not been configured to show the shell session's current absolute directory location, then you can display the location via a shell command. The `pwd` command displays the shell session's current directory location, which is called the *present working directory*. An example of using the `pwd` command is shown here.

```
christine@server01:/usr/bin$ pwd
/usr/bin
christine@server01:/usr/bin$
```

TIP

It is a good habit to use the `pwd` command whenever you change to a new present working directory. Because many shell commands operate on the present working directory, you always want to make sure you are in the correct directory before issuing a command.

You can move to any level within the entire Linux virtual directory structure from any level using the absolute directory reference:

```
christine@server01:/usr/bin$ cd /var/log
christine@server01:/var/log$
christine@server01:/var/log$ pwd
/var/log
christine@server01:/var/log$
```

You can also quickly jump to your home directory from any level within the Linux virtual directory structure:

```
christine@server01:/var/log$ cd
christine@server01:~$
christine@server01:~$ pwd
/home/christine
christine@server01:~$
```

However, if you're just working within your own home directory structure, often using absolute directory references can get tedious. For example, if you're already in the directory `/home/christine`, it seems somewhat cumbersome to have to type the command:

```
cd /home/christine/Documents
```

just to get to your `Documents` directory. Fortunately, there's a simpler solution.

Using relative directory references

Relative directory references allow you to specify a destination directory reference relative to your current location. A relative directory reference doesn't start with a forward slash (`/`).

Instead, a relative directory reference starts with either a directory name (if you're traversing to a directory under your current directory) or a special character. For example, if you are in your home directory and want to move to your `Documents` subdirectory, you can use the `cd` command along with a relative directory reference:

```
christine@server01:~$ pwd
/home/christine
christine@server01:~$
christine@server01:~$ cd Documents
christine@server01:~/Documents$ pwd
/home/christine/Documents
christine@server01:~/Documents$
```

Part I: The Linux Command Line

In the preceding example, note that no forward slash (/) was used. Instead a relative directory reference was used and the present work directory was changed from `/home/christine` to `/home/christine/Documents`, with much less typing.

Also notice in the example that if the prompt is configured to display the present working directory, it keeps the tilde in the display. This shows that the present working directory is in a directory under the user's home directory.

TIP

If you are new to the command line and the Linux directory structure, it is recommended that you stick with absolute directory references for a while. After you become more familiar with the directory layout, switch to using relative directory references.

You can use a relative directory reference with the `cd` command in any directory containing subdirectories. You can also use a special character to indicate a relative directory location.

The two special characters used for relative directory references are:

- The single dot (`.`) to represent the current directory
- The double dot (`..`) to represent the parent directory

You *can* use the single dot, but it doesn't make sense to use it with the `cd` command. Later in the chapter, you will see how another command uses the single dot for relative directory references effectively.

The double dot character is extremely handy when trying to traverse a directory hierarchy. For example, if you are in the `Documents` directory under your home directory and need to go to your `Downloads` directory, also under your home directory, you can do this:

```
christine@server01:~/Documents$ pwd
/home/christine/Documents
christine@server01:~/Documents$ cd ../Downloads
christine@server01:~/Downloads$ pwd
/home/christine/Downloads
christine@server01:~/Downloads$
```

The double dot character takes you back up one level to your home directory; then the `/Downloads` portion of the command takes you back down into the `Downloads` directory. You can use as many double dot characters as necessary to move around. For example, if you are in your home directory (`/home/christine`) and want to go to the `/etc` directory, you could type the following:

```
christine@server01:~$ cd ../../etc
christine@server01:/etc$ pwd
/etc
christine@server01:/etc$
```

Of course, in a case like this, you actually have to do more typing rather than just typing the absolute directory reference, */etc*. Thus, use a relative directory reference only if it makes sense to do so.

NOTE

It's helpful to have a long informative shell CLI prompt, as used in this chapter section. However, for clarity purposes, a simple `$` prompt is used in the rest of the book's examples.

Now that you know how to traverse the directory system and confirm your present working directory, you can start to explore what's contained within the various directories. The next section takes you through the process of looking at files within the directory structure.

Listing Files and Directories

To see what files are available on the system, use the list command (`ls`). This section scribes the `ls` command and options available to format the information it can display.

Displaying a basic listing

The `ls` command at its most basic form displays the files and directories located in your current directory:

```
$ ls
Desktop  Downloads      Music      Pictures  Templates  Videos
Documents  examples.desktop  my_script  Public    test_file
$
```

Notice that the `ls` command produces the listing in alphabetical order (in columns rather than rows). If you're using a terminal emulator that supports color, the `ls` command may also show different types of entries in different colors. The `LS_COLORS` environment variable controls this feature. (Environment variables are covered in Chapter 6). Different Linux distributions set this environment variable depending on the capabilities of the terminal emulator.

If you don't have a color terminal emulator, you can use the `-F` parameter with the `ls` command to easily distinguish files from directories. Using the `-F` parameter produces the following output:

```
$ ls -F
Desktop/  Downloads/      Music/      Pictures/  Templates/  Videos/
Documents/  examples.desktop  my_script*  Public/    test_file
$
```

Part I: The Linux Command Line

The `-F` parameter flags the directories with a forward slash (/), to help identify them in the listing. Similarly, it flags executable files (like the `my_script` file in the preceding code) with an asterisk (*), to help you more easily find files that can be run on the system.

The basic `ls` command can be somewhat misleading. It shows the files and directories contained in the current directory, but not necessarily all of them. Linux often uses *hidden files* to store configuration information. In Linux, hidden files are files with filenames starting with a period (.). These files don't appear in the default `ls` listing. Thus, they are called hidden files.

To display hidden files along with normal files and directories, use the `-a` parameter. Here is an example of using the `-a` parameter with the `ls` command.

```
$ ls -a
.          .compiz  examples.desktop Music    test_file
..         .config  .gconf      my_script Videos
.bash_history Desktop  .gstreamer-0.10 Pictures .Xauthority
.bash_logout .dmrc    .ICEauthority .profile .xsession-errors
.bashrc      Documents .local      Public  .xsession-errors.old
.cache       Downloads .mozilla    Templates
$
```

All the files beginning with a period, hidden files, are now shown. Notice that three files begin with `.bash`. These are hidden files that are used by the bash shell environment. These features are covered in detail in Chapter 6.

The `-R` parameter is another option the `ls` command can use. Called the recursive option, it shows files that are contained within subdirectories in the current directory. If you have lots of subdirectories, this can be quite a long listing. Here's a simple example of what the `-R` parameter produces. The `-F` option was tacked on to help you see the file types:

```
$ ls -F -R
.:
Desktop/  Downloads/      Music/    Pictures/  Templates/  Videos/
Documents/ examples.desktop my_script* Public/    test_file

./Desktop:

./Documents:

./Downloads:

./Music:
ILoveLinux.mp3*

./Pictures:

./Public:
```

```

./Templates:

./Videos:
$

```

Notice that the `-R` parameter shows the contents of the current directory, which are the files from a user's home directory shown in earlier examples. It also shows each subdirectory in the user's home directory and their contents. The only subdirectory containing a file is the `Music` subdirectory, and it contains the executable file, `ILoveLinux.mp3`.

TIP

Option parameters don't have to be entered separately as shown in the nearby example: `ls -F -R`. They can often be combined as follows: `ls -FR`.

In the previous example, there were no subdirectories within subdirectories. If there had been further subdirectories, the `-R` parameter would have continued to traverse those as well. As you can see, for large directory structures, this can become quite a large output listing.

Displaying a long listing

In the basic listings, the `ls` command doesn't produce much information about each file. For listing additional information, another popular parameter is `-l`. The `-l` parameter produces a long listing format, providing more information about each file in the directory:

```

$ ls -l
total 48
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Desktop
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Documents
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Downloads
-rw-r--r-- 1 christine christine 8980 Apr 22 13:36 examples.desktop
-rw-rw-r-- 1 christine christine    0 May 21 13:44 fall
-rw-rw-r-- 1 christine christine    0 May 21 13:44 fell
-rw-rw-r-- 1 christine christine    0 May 21 13:44 fill
-rw-rw-r-- 1 christine christine    0 May 21 13:44 full
drwxr-xr-x 2 christine christine 4096 May 21 11:39 Music
-rw-rw-r-- 1 christine christine    0 May 21 13:25 my_file
-rw-rw-r-- 1 christine christine    0 May 21 13:25 my_script
-rwxrw-r-- 1 christine christine   54 May 21 11:26 my_script
-rw-rw-r-- 1 christine christine    0 May 21 13:42 new_file
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Pictures
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Public
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Templates
-rw-rw-r-- 1 christine christine    0 May 21 11:28 test_file
drwxr-xr-x 2 christine christine 4096 Apr 22 20:37 Videos
$

```


Part I: The Linux Command Line

The long listing format lists each file and subdirectory on a single line. In addition to the filename, the listing shows additional useful information. The first line in the output shows the total number of blocks contained within the directory. After that, each line contains the following information about each file (or directory):

- The file type — such as directory (d), file (-), linked file (l), character device (c), or block device (b)
- The file permissions (see Chapter 6)
- The number of file hard links (See the section “Linking Files” in Chapter 7.)
- The file owner username
- The file primary group name
- The file byte size
- The last time the file was modified
- The filename or directory name

The `-l` parameter is a powerful tool to have. Armed with this parameter, you can see most of the information you need for any file or directory.

The `ls` command has lots of parameters that can come in handy as you do file management. If you type at the shell prompt `man ls`, you see several pages of available parameters for you to use to modify the `ls` command output.

Don't forget that you can also combine many of the parameters. You can often find a parameter combination that not only displays the desired output, but also is easy to remember, such as `ls -alF`.

Filtering listing output

As you've seen in the examples, by default the `ls` command lists all the non-hidden directory files. Sometimes, this can be overkill, especially when you're just looking for information on a few files.

Fortunately, the `ls` command also provides a way for you to define a filter on the command line. It uses the filter to determine which files or directories it should display in the output.

The filter works as a simple text-matching string. Include the filter after any command line parameters you want to use:

```
$ ls -l my_script
-rwxrw-r-- 1 christine christine 54 May 21 11:26 my_script
$
```

When you specify the name of a specific file as the filter, the `ls` command only shows that file's information. Sometimes, you might not know the exact filename you're looking for.

The `ls` command also recognizes standard wildcard characters and uses them to match patterns within the filter:

- A question mark (?) to represent one character
- An asterisk (*) to represent any number of characters

The question mark can be used to replace exactly one character anywhere in the filter string. For example:

```
$ ls -l my_scr?pt
-rw-rw-r-- 1 christine christine  0 May 21 13:25 my_scrapt
-rwxrw-r-- 1 christine christine 54 May 21 11:26 my_script
$
```

The filter `my_scr?pt` matched two files in the directory. Similarly, the asterisk can be used to match zero or more characters:

```
$ ls -l my*
-rw-rw-r-- 1 christine christine  0 May 21 13:25 my_file
-rw-rw-r-- 1 christine christine  0 May 21 13:25 my_scrapt
-rwxrw-r-- 1 christine christine 54 May 21 11:26 my_script
$
```

Using the asterisk finds three different files, starting with the name `my`. As with the question mark, you can place the asterisks anywhere in the filter:

```
$ ls -l my_s*t
-rw-rw-r-- 1 christine christine  0 May 21 13:25 my_scrapt
-rwxrw-r-- 1 christine christine 54 May 21 11:26 my_script
$
```

Using the asterisk and question mark in the filter is called *file globbing*. File globbing is the processing of pattern matching using wildcards. The wildcards are officially called *metacharacter wildcards*. You can use more metacharacter wildcards for file globbing than just the asterisk and question mark. You can also use brackets:

```
$ ls -l my_scr[ai]pt
-rw-rw-r-- 1 christine christine  0 May 21 13:25 my_scrapt
-rwxrw-r-- 1 christine christine 54 May 21 11:26 my_script
$
```

In this example, we used the brackets along with two potential choices for a single character in that position, `a` or `i`. The brackets represent a single character position and give you multiple options for file globbing. You can list choices of characters, as shown in the preceding example, and you can specify a range of characters, such as an alphabetic range [`a - i`]:

```
$ ls -l f[a-i]ll
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fall
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fell
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fill
$
```

Also, you can specify what should not be included in the pattern match by using the exclamation point (!):

```
$ ls -l f[!a]ll
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fell
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fill
-rw-rw-r-- 1 christine christine 0 May 21 13:44 full
$
```

File globbing is a powerful feature when searching for files. It can also be used with other shell commands besides `ls`. You find out more about this later in the chapter.

Handling Files

The shell provides many file manipulation commands on the Linux filesystem. This section walks you through the basic shell commands you need to handle files.

Creating files

Every once in a while you run into a situation where you need to create an empty file. For example, sometimes applications expect a log file to be present before they can write to it. In these situations, you can use the `touch` command to easily create an empty file:

```
$ touch test_one
$ ls -l test_one
-rw-rw-r-- 1 christine christine 0 May 21 14:17 test_one
$
```

The `touch` command creates the new file you specify and assigns your username as the file owner. Notice in the preceding example that the file size is zero because the `touch` command just created an empty file.

The `touch` command can also be used to change the modification time. This is done without changing the file contents:

```
$ ls -l test_one
-rw-rw-r-- 1 christine christine 0 May 21 14:17 test_one
$ touch test_one
$ ls -l test_one
-rw-rw-r-- 1 christine christine 0 May 21 14:35 test_one
$
```

The modification time of `test_one` is now updated to 14:35 from the original time, 14:17. To change only the access time, use the `-a` parameter with the `touch` command:

```
$ ls -l test_one
-rw-rw-r-- 1 christine christine 0 May 21 14:35 test_one
$ touch -a test_one
```

```
$ ls -l test_one
-rw-rw-r-- 1 christine christine 0 May 21 14:35 test_one
$ ls -l --time=atime test_one
-rw-rw-r-- 1 christine christine 0 May 21 14:55 test_one
$
```

In the preceding example, notice that by using only the `ls -l` command, the access time does not display. This is because the modification time is shown by default. To see a file's access time, you need to add an additional parameter, `--time=atime`. After we add that parameter in the preceding example, the file's altered access time is displayed.

Creating empty files and altering file timestamps is not something you will do on a Linux system daily. However, copying files is an action you will do often while using the shell.

Copying files

Copying files and directories from one location in the filesystem to another is a common practice for system administrators. The `cp` command provides this feature.

In its most basic form, the `cp` command uses two parameters — the source object and the destination object: `cp source destination`.

When both the `source` and `destination` parameters are filenames, the `cp` command copies the source file to a new destination file. The new file acts like a brand new file, with an updated modification time:

```
$ cp test_one test_two
$ ls -l test_*
-rw-rw-r-- 1 christine christine 0 May 21 14:35 test_one
-rw-rw-r-- 1 christine christine 0 May 21 15:15 test_two
$
```

The new file `test_two` shows a different modification time than the `test_one` file. If the destination file already exists, the `cp` command may not prompt you to this fact. It is best to add the `-i` option to force the shell to ask whether you want to overwrite a file:

```
$ ls -l test_*
-rw-rw-r-- 1 christine christine 0 May 21 14:35 test_one
-rw-rw-r-- 1 christine christine 0 May 21 15:15 test_two
$
$ cp -i test_one test_two
cp: overwrite 'test_two'? n
$
```

If you don't answer `y`, the file copy does not proceed. You can also copy a file into a pre-existing directory:

```
$ cp -i test_one /home/christine/Documents/
$
```

Part I: The Linux Command Line

```
$ ls -l /home/christine/Documents
total 0
-rw-rw-r-- 1 christine christine 0 May 21 15:25 test_one
$
```

The new file is now under the `Documents` subdirectory, using the same filename as the original.

NOTE

The preceding example uses a trailing forward slash (/) on the destination directory name. Using the slash indicates `Documents` is a directory and not a file. This is helpful for clarity purposes and is important when copying single files. If the forward slash is not used and the subdirectory `/home/christine/Documents` does not exist, problems can occur. In this case, attempting to copy a single file to the `Documents` subdirectory creates a file named `Documents` instead, and no error messages display!

This last example used an absolute directory reference, but you can just as easily use a relative directory reference:

```
$ cp -i test_one Documents/
cp: overwrite 'Documents/test_one'? y
$
$ ls -l Documents
total 0
-rw-rw-r-- 1 christine christine 0 May 21 15:28 test_one
$
```

Earlier in this chapter, you read about the special symbols that can be used in relative directory references. One of them, the single dot (`.`), is great to use with the `cp` command. Remember that the single dot represents your present working directory. If you need to copy a file with a long source object name to your present working directory, the single dot can simplify the task:

```
$ cp -i /etc/NetworkManager/NetworkManager.conf .
$
$ ls -l NetworkManager.conf
-rw-r--r-- 1 christine christine 76 May 21 15:55 NetworkManager.conf
$
```

It's hard to see that single dot! If you look closely, you'll see it at the end of the first example code line. Using the single dot symbol is much easier than typing a full destination object name, when you have long source object names.

TIP

There are many more `cp` command parameters than those described here. Remember that you can see all the different available parameters available for the `cp` command, by typing `man cp`.

The `-R` parameter is a powerful `cp` command option. It allows you to recursively copy the contents of an entire directory in one command:

```
$ ls -Fd *Scripts
Scripts/
$ ls -l Scripts/
total 25
-rwxrw-r-- 1 christine christine 929 Apr  2 08:23 file_mod.sh
-rwxrw-r-- 1 christine christine 254 Jan  2 14:18 SGID_search.sh
-rwxrw-r-- 1 christine christine 243 Jan  2 13:42 SUID_search.sh
$
$ cp -R Scripts/ Mod_Scripts
$ ls -Fd *Scripts
Mod_Scripts/ Scripts/
$ ls -l Mod_Scripts
total 25
-rwxrw-r-- 1 christine christine 929 May 21 16:16 file_mod.sh
-rwxrw-r-- 1 christine christine 254 May 21 16:16 SGID_search.sh
-rwxrw-r-- 1 christine christine 243 May 21 16:16 SUID_search.sh
$
```

The directory `Mod_Scripts` did not exist prior to the `cp -R` command. It was created with the `cp -R` command, and the entire `Scripts` directory's contents were copied into it. Notice that all the files in the new `Mod_Scripts` directory have new dates associated with them. Now `Mod_Scripts` is a complete copy of the `Scripts` directory.

NOTE

In the preceding example, the options `-Fd` were added to the `ls` command. You read about the `-F` option earlier in this chapter. However, the `-d` option may be new to you. The `-d` option lists a directory's information but not its contents.

You can also use wildcard metacharacters in your `cp` commands:

```
$ cp *script Mod_Scripts/
$ ls -l Mod_Scripts
total 26
-rwxrw-r-- 1 christine christine 929 May 21 16:16 file_mod.sh
-rwxrw-r-- 1 christine christine 54  May 21 16:27 my_script
-rwxrw-r-- 1 christine christine 254 May 21 16:16 SGID_search.sh
-rwxrw-r-- 1 christine christine 243 May 21 16:16 SUID_search.sh
$
```

This command copied any files that ended with `script` to `Mod_Scripts`. In this case, only one file needed to be copied: `my_script`.

When copying files, another shell feature can help you besides the single dot and wildcard metacharacters. It is called tab auto-complete.

Using tab auto-complete

When working at the command line, you can easily mistype a command, directory name, or filename. In fact, the longer a directory reference or filename, the greater the chance that you will mistype it.

This is where *tab auto-complete* can be a lifesaver. Tab auto-complete allows you to start typing a filename or directory name and then press the tab key to have the shell complete it for you:

```
$ ls really*
really_ridiculously_long_file_name
$
$ cp really_ridiculously_long_file_name Mod_Scripts/
ls -l Mod_Scripts
total 26
-rwxrw-r-- 1 christine christine 929 May 21 16:16 file_mod.sh
-rwxrw-r-- 1 christine christine 54 May 21 16:27 my_script
-rw-rw-r-- 1 christine christine 0 May 21 17:08
really_ridiculously_long_file_name
-rwxrw-r-- 1 christine christine 254 May 21 16:16 SGID_search.sh
-rwxrw-r-- 1 christine christine 243 May 21 16:16 SUID_search.sh
$
```

In the preceding example, we typed the command **cp really** and pressed the tab key, and the shell auto-completed the rest of the filename! Of course, the destination directory had to be typed, but still tab auto-complete saved the command from several potential typographical errors.

The trick to using tab auto-complete is to give the shell enough filename characters so it can distinguish the desired file from other files. For example, if another filename started with `really`, pressing the tab key would not auto-complete the filename. Instead, you would hear a beep. If this happens, you can press the tab key again, and the shell shows you all the filenames starting with `really`. This feature allows you to see what needs to be typed for tab auto-complete to work properly.

Linking files

Linking files is a great option available in the Linux filesystem. If you need to maintain two (or more) copies of the same file on the system, instead of having separate physical copies, you can use one physical copy and multiple virtual copies, called *links*. A link is a placeholder in a directory that points to the real location of the file. Two types of file links are available in Linux:

- A symbolic link
- A hard link

A *symbolic link* is simply a physical file that points to another file somewhere in the virtual directory structure. The two symbolically linked together files do not share the same contents.

To create a symbolic link to a file, the original file must pre-exist. We can then use the `ln` command with the `-s` option to create the symbolic link:

```
$ ls -l data_file
-rw-rw-r-- 1 christine christine 1092 May 21 17:27 data_file
$
$ ln -s data_file sl_data_file
$
$ ls -l *data_file
-rw-rw-r-- 1 christine christine 1092 May 21 17:27 data_file
lrwxrwxrwx 1 christine christine    9 May 21 17:29 sl_data_file -> data_file
$
```

In the preceding example, notice that the name of the symbolic link, `sl_data_file`, is listed second in the `ln` command. The `->` symbol displayed after the symbolic link file's long listing shows that it is symbolically linked to the file `data_file`.

Also note the symbolic link's file size versus the data file's file size. The symbolic link, `sl_data_file`, is only 9 bytes, whereas the `data_file` is 1092 bytes. This is because `sl_data_file` is only pointing to `data_file`. They do not share contents and are two physically separate files.

Another way to tell that these linked files are separate physical files is by viewing their *inode* number. The inode number of a file or directory is a unique identification number that the kernel assigns to each object in the filesystem. To view a file or directory's inode number, add the `-i` parameter to the `ls` command:

```
$ ls -i *data_file
296890 data_file 296891 sl_data_file
$
```

The example shows that the data file's inode number is 296890, while the `sl_data_file` inode number is different. It is 296891. Thus, they are different files.

A *hard link* creates a separate virtual file that contains information about the original file and where to locate it. However, they are physically the same file. When you reference the hard link file, it's just as if you're referencing the original file. To create a hard link, again the original file must pre-exist, except that this time no parameter is needed on the `ln` command:

```
$ ls -l code_file
-rw-rw-r-- 1 christine christine 189 May 21 17:56 code_file
$
$ ln code_file hl_code_file
```


Part I: The Linux Command Line

```
$  
$ ls -li *code_file  
296892 -rw-rw-r-- 2 christine christine 189 May 21 17:56  
code_file  
296892 -rw-rw-r-- 2 christine christine 189 May 21 17:56  
hl_code_file  
$
```

In the preceding example, we used the `ls -li` command to show both the inode numbers and a long listing for the `*code_files`. Notice that both files, which are hard linked together, share the name inode number. This is because they are physically the same file. Also notice that the link count (the third item in the listing) now shows that both files have two links. In addition, their file size is exactly the same size as well.

NOTE

You can only create a hard link between files on the same physical medium. To create a link between files under separate physical mediums, you must use a symbolic link.

Be careful when copying linked files. If you use the `cp` command to copy a file that's linked to another source file, all you're doing is making another copy of the source file. This can quickly get confusing. Instead of copying the linked file, you can create another link to the original file. You can have many links to the same file with no problems. However, you also don't want to create soft links to other soft-linked files. This creates a chain of links that can be confusing — and easily broken — causing all sorts of problems.

You may find symbolic and hard links difficult concepts. Fortunately, renaming files in the next section is a great deal easier to understand.

Renaming files

In the Linux world, renaming files is called *moving files*. The `mv` command is available to move both files and directories to another location or a new name:

```
$ ls -li f?ll  
296730 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fall  
296717 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fell  
294561 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fill  
296742 -rw-rw-r-- 1 christine christine 0 May 21 13:44 full  
$  
$ mv fall fzll  
$  
$ ls -li f?ll  
296717 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fell  
294561 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fill  
296742 -rw-rw-r-- 1 christine christine 0 May 21 13:44 full  
296730 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fzll  
$
```

Notice that moving the file changed the name from `fall` to `fzll`, but it kept the same inode number and timestamp value. This is because `mv` affects only a file's name.

You can also use `mv` to change a file's location:

```
$ ls -li /home/christine/fzll
296730 -rw-rw-r-- 1 christine christine 0 May 21 13:44
/home/christine/fzll
$
$ ls -li /home/christine/Pictures/
total 0
$ mv fzll Pictures/
$
$ ls -li /home/christine/Pictures/
total 0
296730 -rw-rw-r-- 1 christine christine 0 May 21 13:44 fzll
$
$ ls -li /home/christine/fzll
ls: cannot access /home/christine/fzll: No such file or directory
$
```

In the preceding example, we moved the file `fzll` from `/home/christine` to `/home/christine/Pictures` using the `mv` command. Again, there were no changes to the file's inode number or timestamp value.

Tip

Like the `cp` command, you can use the `-i` option on the `mv` command. Thus, you are asked before the command attempts to overwrite any pre-existing files.

The only change was to the file's location. The `fzll` file no longer exists in `/home/christine`, because a copy of it was not left in its original location, as the `cp` command would have done.

You can use the `mv` command to move a file's location and rename it, all in one easy step:

```
$ ls -li Pictures/fzll
296730 -rw-rw-r-- 1 christine christine 0 May 21 13:44
Pictures/fzll
$
$ mv /home/christine/Pictures/fzll /home/christine/fall
$
$ ls -li /home/christine/fall
296730 -rw-rw-r-- 1 christine christine 0 May 21 13:44
/home/christine/fall
$
$ ls -li /home/christine/Pictures/fzll
ls: cannot access /home/christine/Pictures/fzll:
No such file or directory
```

Part I: The Linux Command Line

For this example, we moved the file `fz11` from a subdirectory, `Pictures`, to the home directory, `/home/christine`, and renamed it to `fall`. Neither the timestamp value nor the inode number changed. Only the location and name were altered.

You can also use the `mv` command to move entire directories and their contents:

```
$ ls -li Mod_Scripts
total 26
296886 -rwxrw-r-- 1 christine christine 929 May 21 16:16
file_mod.sh
296887 -rwxrw-r-- 1 christine christine 54 May 21 16:27
my_script
296885 -rwxrw-r-- 1 christine christine 254 May 21 16:16
SGID_search.sh
296884 -rwxrw-r-- 1 christine christine 243 May 21 16:16
SUID_search.sh
$
$ mv Mod_Scripts Old_Scripts
$
$ ls -li Mod_Scripts
ls: cannot access Mod_Scripts: No such file or directory
$
$ ls -li Old_Scripts
total 26
296886 -rwxrw-r-- 1 christine christine 929 May 21 16:16
file_mod.sh
296887 -rwxrw-r-- 1 christine christine 54 May 21 16:27
my_script
296885 -rwxrw-r-- 1 christine christine 254 May 21 16:16
SGID_search.sh
296884 -rwxrw-r-- 1 christine christine 243 May 21 16:16
SUID_search.sh
$
```

The directory's entire contents are unchanged. The only thing that changes is the name of the directory.

After you know how to rename...err...*move* files with the `mv` command, you realize how simple it is to accomplish. Another easy, but potentially dangerous, task is deleting files.

Deleting files

Most likely at some point you'll want to be able to delete existing files. Whether it's to clean up a filesystem or to remove a software package, you always have opportunities to delete files.

In the Linux world, deleting is called *removing*. The command to remove files in the bash shell is `rm`. The basic form of the `rm` command is simple:

```
$ rm -i fall
rm: remove regular empty file 'fall'? y
$
$ ls -l fall
ls: cannot access fall: No such file or directory
$
```

Notice that the `-i` command parameter prompts you to make sure that you're serious about removing the file. The shell has no recycle bin or trashcan. After you remove a file, it's gone forever. Therefore, a good habit is to always tack on the `-i` parameter to the `rm` command.

You can also use wildcard metacharacters to remove groups of files. However, again, use that `-i` option to protect yourself:

```
$ rm -i f?ll
rm: remove regular empty file 'fell'? y
rm: remove regular empty file 'fill'? y
rm: remove regular empty file 'full'? y
$
$ ls -l f?ll
ls: cannot access f?ll: No such file or directory
$
```

One other feature of the `rm` command, if you're removing lots of files and don't want to be bothered with the prompt, is to use the `-f` parameter to force the removal. Just be careful!

Managing Directories

Linux has a few commands that work for both files and directories (such as the `cp` command), and some that work only for directories. To create a new directory, you need to use a specific command, which is covered in this section. Removing directories can get interesting, so that is covered in this section as well.

Creating directories

Creating a new directory in Linux is easy — just use the `mkdir` command:

```
$ mkdir New_Dir
$ ls -ld New_Dir
drwxrwxr-x 2 christine christine 4096 May 22 09:48 New_Dir
$
```

The system creates a new directory named `New_Dir`. Notice in the new directory's long listing that the directory's record begins with a `d`. This indicates that `New_Dir` is not a file, but a directory.

Part I: The Linux Command Line

You can create directories and subdirectories in “bulk” if needed. However, if you attempt this with just the `mkdir` command, you get the following error message:

```
$ mkdir New_Dir/Sub_Dir/Under_Dir
mkdir: cannot create directory 'New_Dir/Sub_Dir/Under_Dir':
No such file or directory
$
```

To create several directories and subdirectories at the same time, you need to add the `-p` parameter:

```
$ mkdir -p New_Dir/Sub_Dir/Under_Dir
$
$ ls -R New_Dir
New_Dir:
Sub_Dir

New_Dir/Sub_Dir:
Under_Dir

New_Dir/Sub_Dir/Under_Dir:
$
```

The `-p` option on the `mkdir` command makes any missing *parent directories* as needed. A parent directory is a directory that contains other directories at the next level down the directory tree.

Of course, after you make something, you need to know how to delete it. This is especially useful if you created a directory in the wrong location.

Deleting directories

Removing directories can be tricky, and for good reason. There are lots of opportunities for bad things to happen when you start deleting directories. The shell tries to protect us from accidental catastrophes as much as possible. The basic command for removing a directory is `rmdir`:

```
$ touch New_Dir/my_file
$ ls -li New_Dir/
total 0
294561 -rw-rw-r-- 1 christine christine 0 May 22 09:52 my_file
$
$ rmdir New_Dir
rmdir: failed to remove 'New_Dir': Directory not empty
$
```

By default, the `rmdir` command works only for removing **empty** directories. Because we created a file, `my_file`, in the `New_Dir` directory, the `rmdir` command refuses to remove it.

To fix this, we must remove the file first. Then we can use the `rmdir` command on the now empty directory:

```
$ rm -i New_Dir/my_file
rm: remove regular empty file 'New_Dir/my_file'? y
$
$ rmdir New_Dir
$
$ ls -ld New_Dir
ls: cannot access New_Dir: No such file or directory
```

The `rmdir` has no `-i` option to ask if you want to remove the directory. This is one reason it is helpful that `rmdir` removes only empty directories.

You can also use the `rm` command on entire non-empty directories. Using the `-r` option allows the command to descend into the directory, remove the files, and then remove the directory itself:

```
$ ls -l My_Dir
total 0
-rw-rw-r-- 1 christine christine 0 May 22 10:02 another_file
$
$ rm -ri My_Dir
rm: descend into directory 'My_Dir'? y
rm: remove regular empty file 'My_Dir/another_file'? y
rm: remove directory 'My_Dir'? y
$
$ ls -l My_Dir
ls: cannot access My_Dir: No such file or directory
$
```

This also works for descending into multiple subdirectories and is especially useful when you have lots of directories and files to delete:

```
$ ls -FR Small_Dir
Small_Dir:
a_file b_file c_file Teeny_Dir/ Tiny_Dir/

Small_Dir/Teeny_Dir:
e_file

Small_Dir/Tiny_Dir:
d_file
$
$ rm -ir Small_Dir
rm: descend into directory 'Small_Dir'? y
rm: remove regular empty file 'Small_Dir/a_file'? y
rm: descend into directory 'Small_Dir/Tiny_Dir'? y
rm: remove regular empty file 'Small_Dir/Tiny_Dir/d_file'? y
```

Part I: The Linux Command Line

```
rm: remove directory 'Small_Dir/Tiny_Dir'? y
rm: descend into directory 'Small_Dir/Teeny_Dir'? y
rm: remove regular empty file 'Small_Dir/Teeny_Dir/e_file'? y
rm: remove directory 'Small_Dir/Teeny_Dir'? y
rm: remove regular empty file 'Small_Dir/c_file'? y
rm: remove regular empty file 'Small_Dir/b_file'? y
rm: remove directory 'Small_Dir'? y
$
$ ls -FR Small_Dir
ls: cannot access Small_Dir: No such file or directory
$
```

Although this works, it's somewhat awkward. Notice that you still must verify each and every file that gets removed. For a directory with lots of files and subdirectories, this can become tedious.

NOTE

For the `rm` command, the `-r` parameter and the `-R` parameter work exactly the same. When used with the `rm` command, the `-R` parameter also recursively traverses through the directory removing files. It is unusual for a shell command to have different cased parameters with the same function.

The ultimate solution for throwing caution to the wind and removing an entire directory, contents and all, is the `rm` command with both the `-r` and `-f` parameters:

```
$ tree Small_Dir
Small_Dir
├── a_file
├── b_file
├── c_file
├── Teeny_Dir
│   └── e_file
└── Tiny_Dir
    └── d_file

2 directories, 5 files
$
$ rm -rf Small_Dir
$
$ tree Small_Dir
Small_Dir [error opening dir]

0 directories, 0 files
$
```

The `rm -rf` command gives no warnings and no fanfare. This, of course, is an extremely dangerous tool to have, especially if you have superuser privileges. Use it sparingly, and only after triple checking to make sure that you're doing exactly what you want to do!

NOTE

Notice in the preceding example that we used the `tree` utility. It nicely displays directories, subdirectories, and their files. It's a useful utility when you need to understand a directory structure, especially before removing it. This utility may not be installed by default in your Linux distribution. See Chapter 9 for learning about installing software.

In the last few sections, you looked at managing both files and directories. So far we covered everything you need to know about files, except for how to peek inside of them.

Viewing File Contents

You can use several commands for looking inside files without having to pull out a text editor utility (see Chapter 10). This section demonstrates a few of the commands you have available to help you examine files.

Viewing the file type

Before you go charging off trying to display a file, try to get a handle on what type of file it is. If you try to display a binary file, you get lots of gibberish on your monitor and may even lock up your terminal emulator.

The `file` command is a handy little utility to have around. It can peek inside of a file and determine just what kind of file it is:

```
$ file my_file
my_file: ASCII text
$
```

The file in the preceding example is a `text` file. The `file` command determined not only that the file contains text but also the character code format of the text file, `ASCII`.

This following example shows a file that is simply a directory. Thus, the `file` command gives you another method to distinguish a directory:

```
$ file New_Dir
New_Dir: directory
$
```

This third `file` command example shows a file, which is a symbolic link. Note that the `file` command even tells you to which file it is symbolically linked:

```
$ file sl_data_file
sl_data_file: symbolic link to 'data_file'
$
```


Part I: The Linux Command Line

The following example shows what the `file` command returns for a script file. Although the file is ASCII text, because it's a script file, you can execute (run) it on the system:

```
$ file my_script
my_script: Bourne-Again shell script, ASCII text executable
$
```

The final example is a binary executable program. The `file` command determines the platform that the program was compiled for and what types of libraries it requires. This is an especially handy feature if you have a binary executable program from an unknown source:

```
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
[...]
$
```

Now that you know a quick method for viewing a file's type, you can start displaying and viewing files.

Viewing the whole file

If you have a large text file on your hands, you may want to be able to see what's inside of it. Linux has three different commands that can help you here.

Using the `cat` command

The `cat` command is a handy tool for displaying all the data inside a text file:

```
$ cat test1
hello

This is a test file.

That we'll use to      test the cat command.
$
```

Nothing too exciting, just the contents of the text file. However, the `cat` command has a few parameters that can help you out.

The `-n` parameter numbers all the lines for you:

```
$ cat -n test1
 1 hello
 2
 3 This is a test file.
 4
 5
 6 That we'll use to      test the cat command.
$
```

That feature will come in handy when you're examining scripts. If you just want to number the lines that have text in them, the `-b` parameter is for you:

```
$ cat -b test1
 1 hello

 2 This is a test file.

 3 That we'll use to      test the cat command.
$
```

Finally, if you don't want tab characters to appear, use the `-T` parameter:

```
$ cat -T test1
hello

This is a test file.

That we'll use to^Itest the cat command.
$
```

The `-T` parameter replaces any tabs in the text with the `^I` character combination.

For large files, the `cat` command can be somewhat annoying. The text in the file just quickly scrolls off the display without stopping. Fortunately, we have a simple way to solve this problem.

Using the `more` command

The main drawback of the `cat` command is that you can't control what's happening after you start it. To solve that problem, developers created the `more` command. The `more` command displays a text file, but stops after it displays each page of data. We typed the command `more /etc/bash.bashrc` to produce the sample `more` screen shown in Figure 3-3.

FIGURE 3-3

Using the `more` command to display a text file

```
shopt -s checkwinsize

# set variable identifying the chroot you work in (used in the prompt below)
if [ -z "${debian_chroot:-}" ] && [ -r /etc/debian_chroot ]; then
    debian_chroot=$(cat /etc/debian_chroot)
fi

# set a fancy prompt (non-color, overwrite the one in /etc/profile)
PS1='${debian_chroot:+($debian_chroot)}\u@h:\w\$ '

# Commented out, don't overwrite xterm -T "title" -n "icontitle" by default.
# If this is an xterm set the title to user@host:dir
#case "$TERM" in
#xterm*|rxvt*)
#    PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"'
#    ;;
#*)
#    ;;
#esac

# enable bash completion in interactive shells
#if ! shopt -oq posix; then
# if [ -f /usr/share/bash-completion/bash_completion ]; then
#     . /usr/share/bash-completion/bash_completion
# elif [ -f /etc/bash_completion ]; then
#     . /etc/bash_completion
# fi
#fi

--More-- (56%)
```

Notice at the bottom of the screen in Figure 3-3 that the `more` command displays a tag showing that you're still in the `more` application and how far along (56%) in the text file you are. This is the prompt for the `more` command.

The `more` command is a pager utility. Remember from earlier in this chapter a pager utility displays selected bash manual pages when you use the `man` command. Similarly to navigating through the man pages, you can use `more` to navigate through a text file by pressing the spacebar or you can go forward line by line using the Enter key. When you are finished navigating through the file using `more`, press the `q` key to quit.

The `more` command allows some rudimentary movement through the text file. For more advanced features, try the `less` command.

Using the `less` command

From its name, it sounds like it shouldn't be as advanced as the `more` command. However, the `less` command name is actually a play on words and is an advanced version of the `more` command (the `less` command name comes from the phrase "less is more"). It provides several very handy features for scrolling both forward and backward through a text file, as well as some pretty advanced searching capabilities.

The `less` command can also display a file's contents before it finishes reading the entire file. The `cat` and `more` commands cannot do this.

The `less` command operates much the same as the `more` command, displaying one screen of text from a file at a time. It supports the same command set as the `more` command, plus many more options.

TIP

To see all the options available for the `less` command, view its man pages by typing `man less`. You can do the same for the `more` command to see the reference material concerning its various options as well.

One set of features is that the `less` command recognizes the up and down arrow keys as well as the Page Up and Page Down keys (assuming that you're using a properly defined terminal). This gives you full control when viewing a file.

Viewing parts of a file

Often the data you want to view is located either right at the top or buried at the bottom of a text file. If the information is at the top of a large file, you still need to wait for the `cat` or `more` commands to load the entire file before you can view it. If the information is located at the bottom of a file (such as a log file), you need to wade through thousands of lines of text just to get to the last few entries. Fortunately, Linux has specialized commands to solve both of these problems.

Using the tail command

The `tail` command displays the last lines in a file (the file's "tail"). By default, it shows the last 10 lines in the file.

For these examples, we created a text file containing 20 text lines. It is displayed here in its entirety using the `cat` command:

```
$ cat log_file
line1
line2
line3
line4
line5
Hello World - line 6
line7
line8
line9
line10
line11
Hello again - line 12
line13
line14
line15
Sweet - line16
```

Part I: The Linux Command Line

```
line17
line18
line19
Last line - line20
$
```

Now that you have seen the entire text file, you can see the effect of using `tail` to view the file's last 10 lines:

```
$ tail log_file
line11
Hello again - line 12
line13
line14
line15
Sweet - line16
line17
line18
line19
Last line - line20
$
```

You can change the number of lines shown using `tail` by including the `-n` parameter. In this example, only the last two lines of the file are displayed, by adding `-n 2` to the `tail` command:

```
$ tail -n 2 log_file
line19
Last line - line20
$
```

The `-f` parameter is a pretty cool feature of the `tail` command. It allows you to peek inside a file as the file is being used by other processes. The `tail` command stays active and continues to display new lines as they appear in the text file. This is a great way to monitor the system log files in real-time mode.

Using the head command

The `head` command does what you'd expect; it displays a file's first group of lines (the file's "head"). By default, it displays the first 10 lines of text:

```
$ head log_file
line1
line2
line3
line4
line5
Hello World - line 6
```

```
line7
line8
line9
line10
$
```

Similar to the `tail` command, the `head` command supports the `-n` parameter so you can alter what's displayed. Both commands also allow you to simply type a dash along with the number of lines to display, as shown here:

```
$ head -5 log_file
line1
line2
line3
line4
line5
$
```

Usually the beginning of a file doesn't change, so the `head` command doesn't support the `-f` parameter feature as the `tail` command does. The `head` command is a handy way to just peek at the beginning of a file.

Summary

This chapter covered the basics of working with the Linux filesystem from a shell prompt. We began with a discussion of the bash shell and showed you how to interact with the shell. The command line interface (CLI) uses a prompt string to indicate when it's ready for you to enter commands.

The shell provides a wealth of utilities you can use to create and manipulate files. Before you start playing with files, you should understand how Linux stores them. This chapter discussed the basics of the Linux virtual directory and showed you how Linux references storage media devices. After describing the Linux filesystem, the chapter walked you through using the `cd` command to move around the virtual directory.

After showing you how to get to a directory, the chapter demonstrated how to use the `ls` command to list the files and subdirectories. Lots of parameters can customize the output of the `ls` command. You can obtain information on files and directories by using the `ls` command.

The `touch` command is useful for creating empty files and for changing the access or modification times on an existing file. The chapter also discussed using the `cp` command to copy existing files from one location to another. It walked you through the process of linking files instead of copying them, providing an easy way to have the same file in two locations without making a separate copy. The `ln` command provides this linking ability.

Part I: The Linux Command Line

Next, you learned how to rename files (called *moving*) in Linux using the `mv` command and saw how to delete files (called *removing*) using the `rm` command. This chapter also showed you how to perform the same tasks with directories, using the `mkdir` and `rmdir` commands.

Finally, the chapter closed with a discussion on viewing the contents of files. The `cat`, `more`, and `less` commands provide easy methods for viewing the entire contents of a file, while the `tail` and `head` commands are great for peeking inside a file to just see a small portion of it.

The next chapter continues the discussion on bash shell commands. We'll look at more advanced administrator commands that come in handy as you administer your Linux system.

Basic Script Building

IN THIS CHAPTER

Using multiple commands

Creating a script file

Displaying messages

Using variables

Redirecting input and output

Pipes

Performing math

Exiting the script

Now that we've covered the basics of the Linux system and the command line, it's time to start coding. This chapter discusses the basics of writing shell scripts. You need to know these basic concepts before you can start writing your own shell script masterpieces.

Using Multiple Commands

So far you've seen how to use the command line interface (CLI) prompt of the shell to enter commands and view the command results. The key to shell scripts is the ability to enter multiple commands and process the results from each command, even possibly passing the results of one command to another. The shell allows you to chain commands together into a single step.

If you want to run two commands together, you can enter them on the same prompt line, separated with a semicolon:

```
$ date ; who
Mon Feb 21 15:36:09 EST 2014
Christine tty2          2014-02-21 15:26
Samantha tty3          2014-02-21 15:26
Timothy tty1           2014-02-21 15:26
user      tty7          2014-02-19 14:03 (:0)
```


Part II: Shell Scripting Basics

```
user      pts/0          2014-02-21 15:21 (:0.0)
$
```

Congratulations, you just wrote a shell script! This simple script uses just two `bash` shell commands. The `date` command runs first, displaying the current date and time, followed by the output of the `who` command, showing who is currently logged on to the system. Using this technique, you can string together as many commands as you wish, up to the maximum command line character count of 255 characters.

Using this technique is fine for small scripts, but it has a major drawback: You must enter the entire command at the command prompt every time you want to run it. Instead of having to manually enter the commands onto a command line, you can combine the commands into a simple text file. When you need to run the commands, just simply run the text file.

Creating a Script File

To place shell commands in a text file, first you need to use a text editor (see Chapter 10) to create a file and then enter the commands into the file.

When creating a shell script file, you must specify the shell you are using in the first line of the file. Here's the format for this:

```
#!/bin/bash
```

In a normal shell script line, the pound sign (`#`) is used as a comment line. A comment line in a shell script isn't processed by the shell. However, the first line of a shell script file is a special case, and the pound sign followed by the exclamation point tells the shell what shell to run the script under (yes, you can be using a `bash` shell and run your script using another shell).

After indicating the shell, commands are entered onto each line of the file, followed by a carriage return. As mentioned, comments can be added by using the pound sign. An example looks like this:

```
#!/bin/bash
# This script displays the date and who's logged on
date
who
```

And that's all there is to it. You can use the semicolon and put both commands on the same line if you want to, but in a shell script, you can list commands on separate lines. The shell processes commands in the order in which they appear in the file.

Also notice that another line was included that starts with the pound symbol and adds a comment. Lines that start with the pound symbol (other than the first `#!` line) aren't

interpreted by the shell. This is a great way to leave comments for yourself about what's happening in the script, so when you come back to it two years later, you can easily remember what you did.

Save this script in a file called `test1`, and you are almost ready. You need to do a couple of things before you can run your new shell script file.

If you try running the file now, you'll be somewhat disappointed to see this:

```
$ test1
bash: test1: command not found
$
```

The first hurdle to jump is getting the bash shell to find your script file. If you remember from Chapter 6, the shell uses an environment variable called `PATH` to find commands. A quick look at the `PATH` environment variable demonstrates our problem:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin
:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin $
```

The `PATH` environment variable is set to look for commands only in a handful of directories. To get the shell to find the `test1` script, we need to do one of two things:

- Add the directory where our shell script file is located to the `PATH` environment variable.
- Use an absolute or relative file path to reference our shell script file in the prompt.

TIP

Some Linux distributions add the `$HOME/bin` directory to the `PATH` environment variable. This creates a place in every user's `HOME` directory to place files where the shell can find them to execute.

For this example, we use the second method to tell the shell exactly where the script file is located. Remember that to reference a file in the current directory, you can use the single dot operator in the shell:

```
$ ./test1
bash: ./test1: Permission denied
$
```

The shell found the shell script file just fine, but there's another problem. The shell indicated that you don't have permission to execute the file. A quick look at the file permissions should show what's going on here:

```
$ ls -l test1
-rw-rw-r-- 1 user user 73 Sep 24 19:56 test1
$
```

Part II: Shell Scripting Basics

When the new `test1` file was created, the `umask` value determined the default permission settings for the new file. Because the `umask` variable is set to `002` (see Chapter 7) in Ubuntu, the system created the file with only read/write permissions for the file's owner and group.

The next step is to give the file owner permission to execute the file, using the `chmod` command (see Chapter 7):

```
$ chmod u+x test1
$ ./test1
Mon Feb 21 15:38:19 EST 2014
Christine tty2      2014-02-21 15:26
Samantha tty3      2014-02-21 15:26
Timothy tty1       2014-02-21 15:26
user tty7          2014-02-19 14:03 (:0)
user pts/0         2014-02-21 15:21 (:0.0) $
```

Success! Now all the pieces are in the right places to execute the new shell script file.

Displaying Messages

Most shell commands produce their own output, which is displayed on the console monitor where the script is running. Many times, however, you will want to add your own text messages to help the script user know what is happening within the script. You can do this with the `echo` command. The `echo` command can display a simple text string if you add the string following the command:

```
$ echo This is a test
This is a test
$
```

Notice that by default you don't need to use quotes to delineate the string you're displaying. However, sometimes this can get tricky if you are using quotes within your string:

```
$ echo Let's see if this'll work
Lets see if thisll work
$
```

The `echo` command uses either double or single quotes to delineate text strings. If you use them within your string, you need to use one type of quote within the text and the other type to delineate the string:

```
$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
$
```

Now all the quotation marks appear properly in the output.

You can add `echo` statements anywhere in your shell scripts where you need to display additional information:

```
$ cat test1
#!/bin/bash
# This script displays the date and who's logged on
echo The time and date are:
date
echo "Let's see who's logged into the system:"
who
$
```

When you run this script, it produces the following output:

```
$ ./test1
The time and date are:
Mon Feb 21 15:41:13 EST 2014
Let's see who's logged into the system:
Christine tty2      2014-02-21 15:26
Samantha tty3      2014-02-21 15:26
Timothy tty1       2014-02-21 15:26
user tty7          2014-02-19 14:03 (:0)
user pts/0         2014-02-21 15:21 (:0.0)
$
```

That's nice, but what if you want to `echo` a text string on the same line as a command output? You can use the `-n` parameter for the `echo` statement to do that. Just change the first `echo` statement line to this:

```
echo -n "The time and date are: "
```

You need to use quotes around the string to ensure that there's a space at the end of the echoed string. The command output begins exactly where the string output stops. The output now looks like this:

```
$ ./test1
The time and date are: Mon Feb 21 15:42:23 EST 2014
Let's see who's logged into the system:
Christine tty2      2014-02-21 15:26
Samantha tty3      2014-02-21 15:26
Timothy tty1       2014-02-21 15:26
user tty7          2014-02-19 14:03 (:0)
user pts/0         2014-02-21 15:21 (:0.0)
$
```

Perfect! The `echo` command is a crucial piece of shell scripts that interact with users. You'll find yourself using it in many situations, especially when you want to display the values of script variables. Let's look at that next.

Using Variables

Just running individual commands from the shell script is useful, but this has its limitations. Often, you'll want to incorporate other data in your shell commands to process information. You can do this by using *variables*. Variables allow you to temporarily store information within the shell script for use with other commands in the script. This section shows how to use variables in your shell scripts.

Environment variables

You've already seen one type of Linux variable in action. Chapter 6 described the environment variables available in the Linux system. You can access these values from your shell scripts as well.

The shell maintains environment variables that track specific system information, such as the name of the system, the name of the user logged in to the system, the user's system ID (called UID), the default home directory of the user, and the search path used by the shell to find programs. You can display a complete list of active environment variables available by using the `set` command:

```
$ set
BASH=/bin/bash
[...]
HOME=/home/Samantha
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=Samantha
[...]
```

You can tap into these environment variables from within your scripts by using the environment variable's name preceded by a dollar sign. This is demonstrated in the following script:

```
$ cat test2
#!/bin/bash
# display user information from the system.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
$
```

The `$USER`, `$UID`, and `$HOME` environment variables are used to display the pertinent information about the logged-in user. The output should look something like this:

```
$chmod u+x test2
$ ./test2
User info for userid: Samantha
UID: 1001
HOME: /home/Samantha
$
```

Notice that the environment variables in the `echo` commands are replaced by their current values when the script runs. Also notice that we were able to place the `$USER` system variable within the double quotation marks in the first string, and the shell script still figured out what we meant. There is a drawback to using this method, however. Look at what happens in this example:

```
$ echo "The cost of the item is $15"
The cost of the item is 5
```

That is obviously not what was intended. Whenever the script sees a dollar sign within quotes, it assumes you're referencing a variable. In this example, the script attempted to display the variable `$1` (which was not defined) and then the number 5. To display an actual dollar sign, you must precede it with a backslash character:

```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

That's better. The backslash allowed the shell script to interpret the dollar sign as an actual dollar sign and not a variable. The next section shows how to create your own variables in your scripts.

NOTE

You may also see variables referenced using the format `${variable}`. The extra braces around the variable name are often used to help identify the variable name from the dollar sign.

User variables

In addition to the environment variables, a shell script allows you to set and use your own variables within the script. Setting variables allows you to temporarily store data and use it throughout the script, making the shell script more like a real computer program.

User variables can be any text string of up to 20 letters, digits, or an underscore character. User variables are case sensitive, so the variable `Var1` is different from the variable `var1`. This little rule often gets novice script programmers in trouble.

Part II: Shell Scripting Basics

Values are assigned to user variables using an equal sign. No spaces can appear between the variable, the equal sign, and the value (another trouble spot for novices). Here are a few examples of assigning values to user variables:

```
var1=10
var2=-57
var3=testing
var4="still more testing"
```

The shell script automatically determines the data type used for the variable value. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes.

Just like system variables, user variables can be referenced using the dollar sign:

```
$ cat test3
#!/bin/bash
# testing variables
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

Running the script produces the following output:

```
$ chmod u+x test3
$ ./test3
Katie checked in 10 days ago
Jessica checked in 5 days ago
$
```

Each time the variable is referenced, it produces the value currently assigned to it. It's important to remember that when referencing a variable value you use the dollar sign, but when referencing the variable to assign a value to it, you do not use the dollar sign. Here's an example of what I mean:

```
$ cat test4
#!/bin/bash
# assigning a variable value to another variable

value1=10
value2=$value1
echo The resulting value is $value2
$
```

When you use the value of the *value1* variable in the assignment statement, you must still use the dollar sign. This code produces the following output:

```
$ chmod u+x test4
$ ./test4
The resulting value is 10
$
```

If you forget the dollar sign and make the *value2* assignment line look like this:

```
value2=value1
```

you get the following output:

```
$ ./test4
The resulting value is value1
$
```

Without the dollar sign, the shell interprets the variable name as a normal text string, which is most likely not what you wanted.

Command substitution

One of the most useful features of shell scripts is the ability to extract information from the output of a command and assign it to a variable. After you assign the output to a variable, you can use that value anywhere in your script. This comes in handy when processing data in your scripts.

There are two ways to assign the output of a command to a variable:

- The backtick character (```)
- The `$()` format

Be careful with the backtick character; it is not the normal single quotation mark character you are used to using for strings. Because it is not used very often outside of shell scripts, you may not even know where to find it on your keyboard. You should become familiar with it because it's a crucial component of many shell scripts. Hint: On a U.S. keyboard, it is usually on the same key as the tilde symbol (`~`).

Command substitution allows you to assign the output of a shell command to a variable. Although this doesn't seem like much, it is a major building block in script programming.

You must either surround the entire command line command with two backtick characters:

```
testing=`date`
```

or use the `$()` format:

```
testing=$(date)
```


Part II: Shell Scripting Basics

The shell runs the command within the command substitution characters and assigns the output to the variable `testing`. Notice that there are no spaces between the assignment equal sign and the command substitution character. Here's an example of creating a variable using the output from a normal shell command:

```
$ cat test5
#!/bin/bash
testing=$(date)
echo "The date and time are: " $testing
$
```

The variable `testing` receives the output from the `date` command, and it is used in the `echo` statement to display it. Running the shell script produces the following output:

```
$ chmod u+x test5
$ ./test5
The date and time are: Mon Jan 31 20:23:25 EDT 2014
$
```

That's not all that exciting in this example (you could just as easily just put the command in the `echo` statement), but after you capture the command output in a variable, you can do anything with it.

Here's a popular example of how command substitution is used to capture the current date and use it to create a unique filename in a script:

```
#!/bin/bash
# copy the /usr/bin directory listing to a log file
today=$(date +%y%m%d)
ls /usr/bin -al > log.$today
```

The `today` variable is assigned the output of a formatted date command. This is a common technique used to extract date information for log filenames. The `+%y%m%d` format instructs the date command to display the date as a two-digit year, month, and day:

```
$ date +%y%m%d
140131
$
```

The script assigns the value to a variable, which is then used as part of a filename. The file itself contains the redirected output (discussed in the "Redirecting Input and Output" section) of a directory listing. After running the script, you should see a new file in your directory:

```
-rw-r--r--  1 user  user           769 Jan 31 10:15 log.140131
```

The log file appears in the directory using the value of the `$today` variable as part of the filename. The contents of the log file are the directory listing from the `/usr/bin` directory. If the script runs the next day, the log filename is `log.140201`, thus creating a new file for the new day.

CAUTION

Command substitution creates what's called a *subshell* to run the enclosed command. A subshell is a separate child shell generated from the shell that's running the script. Because of that, any variables you create in the script aren't available to the subshell command.

Subshells are also created if you run a command from the command prompt using the `./` path, but they aren't created if you just run the command without a path. However, if you use a built-in shell command, that doesn't generate a subshell. Be careful when running scripts from the command prompt!

Redirecting Input and Output

Sometimes, you want to save the output from a command instead of just having it displayed on the monitor. The bash shell provides a few different operators that allow you to *redirect* the output of a command to an alternative location (such as a file). Redirection can be used for input as well as output, redirecting a file to a command for input. This section describes what you need to do to use redirection in your shell scripts.

Output redirection

The most basic type of redirection is sending output from a command to a file. The bash shell uses the greater-than symbol (`>`) for this:

```
command > outputfile
```

Anything that would appear on the monitor from the command instead is stored in the output file specified:

```
$ date > test6
$ ls -l test6
-rw-r--r--  1 user  user          29 Feb 10 17:56 test6
$ cat test6
Thu Feb 10 17:56:58 EDT 2014
$
```

The redirect operator created the file `test6` (using the default `umask` settings) and redirected the output from the `date` command to the `test6` file. If the output file already exists, the redirect operator overwrites the existing file with the new file data:

```
$ who > test6
$ cat test6
user pts/0 Feb 10 17:55
$
```

Now the contents of the `test6` file contain the output from the `who` command.

Part II: Shell Scripting Basics

Sometimes, instead of overwriting the file's contents, you may need to append output from a command to an existing file — for example, if you're creating a log file to document an action on the system. In this situation, you can use the double greater-than symbol (`>>`) to append data:

```
$ date >> test6
$ cat test6
user      pts/0      Feb 10 17:55
Thu Feb 10 18:02:14 EDT 2014
$
```

The `test6` file still contains the original data from the `who` command processed earlier — and now it contains the new output from the `date` command.

Input redirection

Input redirection is the opposite of output redirection. Instead of taking the output of a command and redirecting it to a file, input redirection takes the content of a file and redirects it to a command.

The input redirection symbol is the less-than symbol (`<`):

```
command < inputfile
```

The easy way to remember this is that the command is always listed first in the command line, and the redirection symbol “points” to the way the data is flowing. The less-than symbol indicates that the data is flowing from the input file to the command.

Here's an example of using input redirection with the `wc` command:

```
$ wc < test6
      2      11      60
$
```

The `wc` command provides a count of text in the data. By default, it produces three values:

- The number of lines in the text
- The number of words in the text
- The number of bytes in the text

By redirecting a text file to the `wc` command, you can get a quick count of the lines, words, and bytes in the file. The example shows that there are 2 lines, 11 words, and 60 bytes in the `test6` file.

Another method of input redirection is called *inline input redirection*. This method allows you to specify the data for input redirection on the command line instead of in a file. This may seem somewhat odd at first, but a few applications are available for this process (such as those shown in the “Performing Math” section).

The inline input redirection symbol is the double less-than symbol (<<). Besides this symbol, you must specify a text marker that delineates the beginning and end of the data used for input. You can use any string value for the text marker, but it must be the same at the beginning of the data and the end of the data:

```
command << marker
data
marker
```

When using inline input redirection on the command line, the shell prompts for data using the secondary prompt, defined in the `PS2` environment variable (see Chapter 6). Here's how this looks when you use it:

```
$ wc << EOF
> test string 1
> test string 2
> test string 3
> EOF
      3      9     42
$
```

The secondary prompt continues to prompt for more data until you enter the string value for the text marker. The `wc` command performs the line, word, and byte counts of the data supplied by the inline input redirection.

Pipes

Sometimes, you need to send the output of one command to the input of another command. This is possible using redirection, but somewhat clunky:

```
$ rpm -qa > rpm.list
$ sort < rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686

[...]
```

Part II: Shell Scripting Basics

The `rpm` command manages the software packages installed on systems using the Red Hat Package Management system (RPM), such as the Fedora system as shown. When used with the `-qa` parameters, it produces a list of the existing packages installed, but not necessarily in any specific order. If you're looking for a specific package or group of packages, it can be difficult to find it using the output of the `rpm` command.

Using the standard output redirection, the output was redirected from the `rpm` command to a file, called `rpm.list`. After the command finished, the `rpm.list` file contained a list of all the installed software packages on my system. Next, input redirection was used to send the contents of the `rpm.list` file to the `sort` command to sort the package names alphabetically.

That was useful, but again, a somewhat clunky way of producing the information. Instead of redirecting the output of a command to a file, you can redirect the output to another command. This process is called *pipng*.

Like the command substitution backtick, the symbol for piping is not used often outside of shell scripting. The symbol is two vertical lines, one above the other. However, the `pipe` symbol often looks like a single vertical line in print (`|`). On a U.S. keyboard, it is usually on the same key as the backslash (`\`). The `pipe` is put between the commands to redirect the output from one to the other:

```
command1 | command2
```

Don't think of piping as running two commands back to back. The Linux system actually runs both commands at the same time, linking them together internally in the system. As the first command produces output, it's sent immediately to the second command. No intermediate files or buffer areas are used to transfer the data.

Now, using piping you can easily `pipe` the output of the `rpm` command directly to the `sort` command to produce your results:

```
$ rpm -qa | sort
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
```

```
[...]
```

Unless you're a (very) quick reader, you probably couldn't keep up with the output generated by this command. Because the piping feature operates in real time, as soon as the `rpm` command produces data, the `sort` command gets busy sorting it. By the time the `rpm` command finishes outputting data, the `sort` command already has the data sorted and starts displaying it on the monitor.

There's no limit to the number of pipes you can use in a command. You can continue piping the output of commands to other commands to refine your operation.

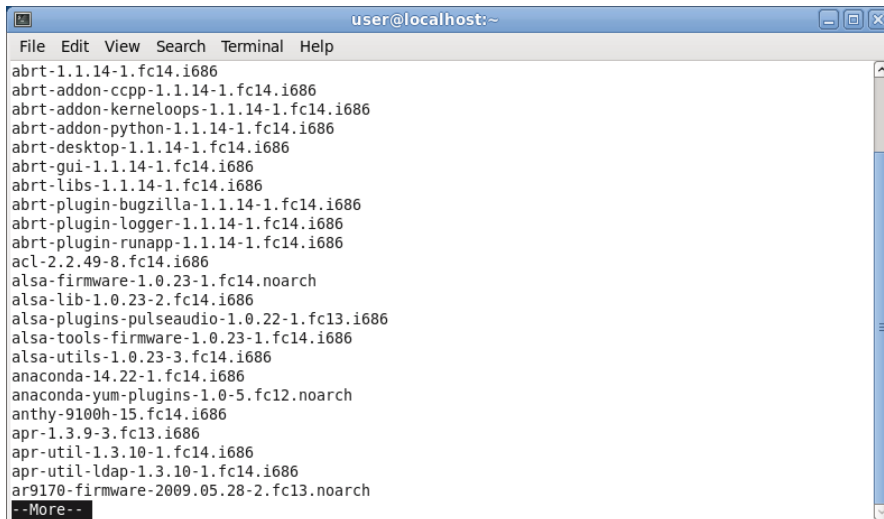
In this case, because the output of the `sort` command zooms by so quickly, you can use one of the text paging commands (such as `less` or `more`) to force the output to stop at every screen of data:

```
$ rpm -qa | sort | more
```

This command sequence runs the `rpm` command, pipes the output to the `sort` command, and then pipes that output to the `more` command to display the data, stopping after every screen of information. This now lets you pause and read what's on the display before continuing, as shown in Figure 11-1.

FIGURE 11-1

Using piping to send data to the `more` command



To get even fancier, you can use redirection along with piping to save your output to a file:

```
$ rpm -qa | sort > rpm.list
$ more rpm.list
```

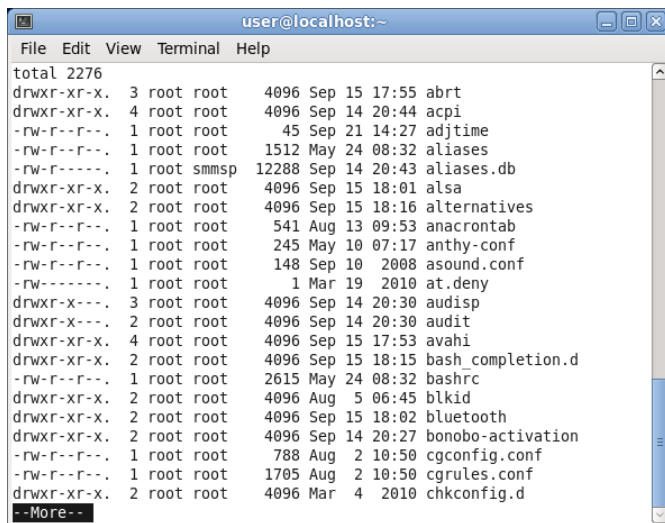
Part II: Shell Scripting Basics

```
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
[...]
```

As expected, the data in the `rpm.list` file is now sorted!

By far one of the most popular uses of piping is piping the results of commands that produce long output to the `more` command. This is especially common with the `ls` command, as shown in Figure 11-2.

FIGURE 11-2
Using the `more` command with the `ls` command



```
user@localhost:~
File Edit View Terminal Help
total 2276
drwxr-xr-x. 3 root root 4096 Sep 15 17:55 abrt
drwxr-xr-x. 4 root root 4096 Sep 14 20:44 acpi
-rw-r--r--. 1 root root 45 Sep 21 14:27 adjtime
-rw-r--r--. 1 root root 1512 May 24 08:32 aliases
-rw-r-----. 1 root smmsp 12288 Sep 14 20:43 aliases.db
drwxr-xr-x. 2 root root 4096 Sep 15 18:01 alsa
drwxr-xr-x. 2 root root 4096 Sep 15 18:16 alternatives
-rw-r--r--. 1 root root 541 Aug 13 09:53 anacrontab
-rw-r--r--. 1 root root 245 May 10 07:17 anthy-conf
-rw-r--r--. 1 root root 148 Sep 10 2008 asound.conf
-rw-----. 1 root root 1 Mar 19 2010 at.deny
drwxr-x---. 3 root root 4096 Sep 14 20:30 audisp
drwxr-x---. 2 root root 4096 Sep 14 20:30 audit
drwxr-xr-x. 4 root root 4096 Sep 15 17:53 avahi
drwxr-xr-x. 2 root root 4096 Sep 15 18:15 bash.completion.d
-rw-r--r--. 1 root root 2615 May 24 08:32 bashrc
drwxr-xr-x. 2 root root 4096 Aug 5 06:45 blkid
drwxr-xr-x. 2 root root 4096 Sep 15 18:02 bluetooth
drwxr-xr-x. 2 root root 4096 Sep 14 20:27 bonobo-activation
-rw-r--r--. 1 root root 788 Aug 2 10:50 cgconfig.conf
-rw-r--r--. 1 root root 1705 Aug 2 10:50 cgrules.conf
drwxr-xr-x. 2 root root 4096 Mar 4 2010 chkconfig.d
--More--
```

The `ls -l` command produces a long listing of all the files in the directory. For directories with lots of files, this can be quite a listing. By piping the output to the `more` command, you force the output to stop at the end of every screen of data.

Performing Math

Another feature crucial to any programming language is the ability to manipulate numbers. Unfortunately, for shell scripts this process is a bit awkward. There are two different ways to perform mathematical operations in your shell scripts.

The `expr` command

Originally, the Bourne shell provided a special command that was used for processing mathematical equations. The `expr` command allowed the processing of equations from the command line, but it is extremely clunky:

```
$ expr 1 + 5
6
```

The `expr` command recognizes a few different mathematical and string operators, shown in Table 11-1.

TABLE 11-1 The `expr` Command Operators

Operator	Description
<code>ARG1 ARG2</code>	Returns <code>ARG1</code> if neither argument is null or zero; otherwise, returns <code>ARG2</code>
<code>ARG1 & ARG2</code>	Returns <code>ARG1</code> if neither argument is null or zero; otherwise, returns 0
<code>ARG1 < ARG2</code>	Returns 1 if <code>ARG1</code> is less than <code>ARG2</code> ; otherwise, returns 0
<code>ARG1 <= ARG2</code>	Returns 1 if <code>ARG1</code> is less than or equal to <code>ARG2</code> ; otherwise, returns 0
<code>ARG1 = ARG2</code>	Returns 1 if <code>ARG1</code> is equal to <code>ARG2</code> ; otherwise, returns 0
<code>ARG1 != ARG2</code>	Returns 1 if <code>ARG1</code> is not equal to <code>ARG2</code> ; otherwise, returns 0
<code>ARG1 >= ARG2</code>	Returns 1 if <code>ARG1</code> is greater than or equal to <code>ARG2</code> ; otherwise, returns 0
<code>ARG1 > ARG2</code>	Returns 1 if <code>ARG1</code> is greater than <code>ARG2</code> ; otherwise, returns 0
<code>ARG1 + ARG2</code>	Returns the arithmetic sum of <code>ARG1</code> and <code>ARG2</code>
<code>ARG1 - ARG2</code>	Returns the arithmetic difference of <code>ARG1</code> and <code>ARG2</code>
<code>ARG1 * ARG2</code>	Returns the arithmetic product of <code>ARG1</code> and <code>ARG2</code>
<code>ARG1 / ARG2</code>	Returns the arithmetic quotient of <code>ARG1</code> divided by <code>ARG2</code>
<code>ARG1 % ARG2</code>	Returns the arithmetic remainder of <code>ARG1</code> divided by <code>ARG2</code>
<code>STRING : REGEXP</code>	Returns the pattern match if <code>REGEXP</code> matches a pattern in <code>STRING</code>

Continues

Part II: Shell Scripting Basics

TABLE 11-1 (continued)

Operator	Description
<code>match STRING REGEXP</code>	Returns the pattern match if <code>REGEXP</code> matches a pattern in <code>STRING</code>
<code>substr STRING POS LENGTH</code>	Returns the substring <code>LENGTH</code> characters in length, starting at position <code>POS</code> (starting at 1)
<code>index STRING CHARS</code>	Returns position in <code>STRING</code> where <code>CHARS</code> is found; otherwise, returns 0
<code>length STRING</code>	Returns the numeric length of the string <code>STRING</code>
<code>+ TOKEN</code>	Interprets <code>TOKEN</code> as a string, even if it's a keyword
<code>(EXPRESSION)</code>	Returns the value of <code>EXPRESSION</code>

Although the standard operators work fine in the `expr` command, the problem occurs when using them from a script or the command line. Many of the `expr` command operators have other meanings in the shell (such as the asterisk). Using them in the `expr` command produces odd results:

```
$ expr 5 * 2
expr: syntax error
$
```

To solve this problem, you need to use the shell escape character (the backslash) to identify any characters that may be misinterpreted by the shell before being passed to the `expr` command:

```
$ expr 5 \* 2
10
$
```

Now that's really starting to get ugly! Using the `expr` command in a shell script is equally cumbersome:

```
$ cat test6
#!/bin/bash
# An example of using the expr command
var1=10
var2=20
var3=$(expr $var2 / $var1)
echo The result is $var3
```

To assign the result of a mathematical equation to a variable, you have to use command substitution to extract the output from the `expr` command:

```
$ chmod u+x test6
$ ./test6
The result is 2
$
```

Fortunately, the bash shell has an improvement for processing mathematical operators as you shall see in the next section.

Using brackets

The bash shell includes the `expr` command to stay compatible with the Bourne shell; however, it also provides a much easier way of performing mathematical equations. In bash, when assigning a mathematical value to a variable, you can enclose the mathematical equation using a dollar sign and square brackets (`[operation]`):

```
$ var1=$((1 + 5))
$ echo $var1
6
$ var2=$(( $var1 * 2 ))
$ echo $var2
12
$
```

Using brackets makes shell math much easier than with the `expr` command. This same technique also works in shell scripts:

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$(( $var1 * ($var2 - $var3) ))
echo The final result is $var4
$
```

Running this script produces the output:

```
$ chmod u+x test7
$ ./test7
The final result is 500
$
```

Also, notice that when using the square brackets method for calculating equations, you don't need to worry about the multiplication symbol, or any other characters, being misinterpreted by the shell. The shell knows that it's not a wildcard character because it is within the square brackets.

There's one major limitation to performing math in the bash shell script. Look at this example:

```
$ cat test8
#!/bin/bash
var1=100
```

Part II: Shell Scripting Basics

```
var2=45
var3=${var1 / $var2}
echo The final result is $var3
$
```

Now run it and see what happens:

```
$ chmod u+x test8
$ ./test8
The final result is 2
$
```

The bash shell mathematical operators support only integer arithmetic. This is a huge limitation if you're trying to do any sort of real-world mathematical calculations.

NOTE

The z shell (zsh) provides full floating-point arithmetic operations. If you require floating-point calculations in your shell scripts, you might consider checking out the z shell (discussed in Chapter 23).

A floating-point solution

You can use several solutions for overcoming the bash integer limitation. The most popular solution uses the built-in bash calculator, called `bc`.

The basics of `bc`

The bash calculator is actually a programming language that allows you to enter floating-point expressions at a command line and then interprets the expressions, calculates them, and returns the result. The bash calculator recognizes these:

- Numbers (both integer and floating point)
- Variables (both simple variables and arrays)
- Comments (lines starting with a pound sign or the C language `/* */` pair)
- Expressions
- Programming statements (such as `if-then` statements)
- Functions

You can access the bash calculator from the shell prompt using the `bc` command:

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
```

```
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
$
```

The example starts out by entering the expression `12 * 5.4`. The bash calculator returns the answer. Each subsequent expression entered into the calculator is evaluated, and the result is displayed. To exit the bash calculator, you must enter `quit`.

The floating-point arithmetic is controlled by a built-in variable called `scale`. You must set this value to the desired number of decimal places you want in your answers, or you won't get what you were looking for:

```
$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
quit
$
```

The default value for the `scale` variable is zero. Before the `scale` value is set, the bash calculator provides the answer to zero decimal places. After you set the `scale` variable value to four, the bash calculator displays the answer to four decimal places. The `-q` command line parameter suppresses the lengthy welcome banner from the bash calculator.

In addition to normal numbers, the bash calculator also understands variables:

```
$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$
```

After a variable value is defined, you can use the variable throughout the bash calculator session. The `print` statement allows you to print variables and numbers.

Using `bc` in scripts

Now you may be wondering how the bash calculator is going to help you with floating-point arithmetic in your shell scripts. Do you remember command substitution? Yes, you can use

Part II: Shell Scripting Basics

the command substitution character to run a `bc` command and assign the output to a variable! The basic format to use is this:

```
variable=$(echo "options; expression" | bc)
```

The first portion, `options`, allows you to set variables. If you need to set more than one variable, separate them using the semicolon. The expression parameter defines the mathematical expression to evaluate using `bc`. Here's a quick example of doing this in a script:

```
$ cat test9
#!/bin/bash
var1=$(echo "scale=4; 3.44 / 5" | bc)
echo The answer is $var1
$
```

This example sets the scale variable to four decimal places and then specifies a specific calculation for the expression. Running this script produces the following output:

```
$ chmod u+x test9
$ ./test9
The answer is .6880
$
```

Now that's fancy! You aren't limited to just using numbers for the expression value. You can also use variables defined in the shell script:

```
$ cat test10
#!/bin/bash
var1=100
var2=45
var3=$(echo "scale=4; $var1 / $var2" | bc)
echo The answer for this is $var3
$
```

The script defines two variables, which are used within the expression sent to the `bc` command. Remember to use the dollar sign to signify the value for the variables and not the variables themselves. The output of this script is as follows:

```
$ ./test10
The answer for this is 2.2222
$
```

And of course, after a value is assigned to a variable, that variable can be used in yet another calculation:

```
$ cat test11
#!/bin/bash
var1=20
var2=3.14159
```

```
var3=$(echo "scale=4; $var1 * $var1" | bc)
var4=$(echo "scale=4; $var3 * $var2" | bc)
echo The final result is $var4
$
```

This method works fine for short calculations, but sometimes you need to get more involved with your numbers. If you have more than just a couple of calculations, it gets confusing trying to list multiple expressions on the same command line.

There's a solution to this problem. The `bc` command recognizes input redirection, allowing you to redirect a file to the `bc` command for processing. However, this also can get confusing, because you'd need to store your expressions in a file.

The best method is to use inline input redirection, which allows you to redirect data directly from the command line. In the shell script, you assign the output to a variable:

```
variable=$(bc << EOF
options
statements
expressions
EOF
)
```

The `EOF` text string indicates the beginning and end of the inline redirection data. Remember that the command substitution characters are still needed to assign the output of the `bc` command to the variable.

Now you can place all the individual bash calculator elements on separate lines in the script file. Here's an example of using this technique in a script:

```
$ cat test12
#!/bin/bash

var1=10.46
var2=43.67
var3=33.2
var4=71

var5=$(bc << EOF
scale = 4
a1 = ( $var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
)

echo The final answer for this mess is $var5
$
```

Part II: Shell Scripting Basics

Placing each option and expression on a separate line in your script makes things cleaner and easier to read and follow. The `EOF` string indicates the start and end of the data to redirect to the `bc` command. Of course, you must use the command substitution characters to indicate the command to assign to the variable.

You'll also notice in this example that you can assign variables within the bash calculator. It's important to remember that any variables created within the bash calculator are valid only within the bash calculator and can't be used in the shell script.

Exiting the Script

So far in our sample scripts, we terminated things pretty abruptly. When we were finished with our last command, we just ended the script. There's a more elegant way of completing things available to us.

Every command that runs in the shell uses an *exit status* to indicate to the shell that it's finished processing. The exit status is an integer value between 0 and 255 that's passed by the command to the shell when the command finishes running. You can capture this value and use it in your scripts.

Checking the exit status

Linux provides the `$?` special variable that holds the exit status value from the last command that executed. You must view or use the `$?` variable immediately after the command you want to check. It changes values to the exit status of the last command executed by the shell:

```
$ date
Sat Jan 15 10:01:30 EDT 2014
$ echo $?
0
$
```

By convention, the exit status of a command that successfully completes is zero. If a command completes with an error, then a positive integer value is placed in the exit status:

```
$ asdfg
-bash: asdfg: command not found
$ echo $?
127
$
```

The invalid command returns an exit status of 127. There's not much of a standard convention to Linux error exit status codes. However, you can use the guidelines shown in Table 11-2.

TABLE 11-2 Linux Exit Status Codes

Code	Description
0	Successful completion of the command
1	General unknown error
2	Misuse of shell command
126	The command can't execute
127	Command not found
128	Invalid exit argument
128+x	Fatal error with Linux signal x
130	Command terminated with Ctrl+C
255	Exit status out of range

127

An exit status value of 126 indicates that the user didn't have the proper permissions set to execute the command:

```
$ ./myprog.c
-bash: ./myprog.c: Permission denied
$ echo $?
126
$
```

Another common error you'll encounter occurs if you supply an invalid parameter to a command:

```
$ date %t
date: invalid date '%t'
$ echo $?
1
$
```

This generates the general exit status code of 1, indicating that an unknown error occurred in the command.

The exit command

By default, your shell script exits with the exit status of the last command in your script:

```
$ ./test6
The result is 2
$ echo $?
0
$
```


Part II: Shell Scripting Basics

You can change that to return your own exit status code. The `exit` command allows you to specify an exit status when your script ends:

```
$ cat test13
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$((var1 + var2))
echo The answer is $var3
exit 5
$
```

When you check the exit status of the script, you get the value used as the parameter of the `exit` command:

```
$ chmod u+x test13
$ ./test13
The answer is 40
$ echo $?
5
$
```

You can also use variables in the `exit` command parameter:

```
$ cat test14
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$((var1 + var2))
exit $var3
$
```

When you run this command, it produces the following exit status:

```
$ chmod u+x test14
$ ./test14
$ echo $?
40
$
```

You should be careful with this feature, however, because the exit status codes can only go up to 255. Watch what happens in this example:

```
$ cat test14b
#!/bin/bash
# testing the exit status
var1=10
```

```
var2=30
var3=${var1 * $var2}
echo The value is $var3
exit $var3
$
```

Now when you run it, you get the following:

```
$ ./test14b
The value is 300
$ echo $?
44
$
```

The exit status code is reduced to fit in the 0 to 255 range. The shell does this by using modulo arithmetic. The *modulo* of a value is the remainder after a division. The resulting number is the remainder of the specified number divided by 256. In the case of 300 (the result value), the remainder is 44, which is what appears as the exit status code.

In Chapter 12, you'll see how you can use the *if-then* statement to check the error status returned by a command to see whether the command was successful.

Summary

The bash shell script allows you to string commands together into a script. The most basic way to create a script is to separate multiple commands on the command line using a semicolon. The shell executes each command in order, displaying the output of each command on the monitor.

You can also create a shell script file, placing multiple commands in the file for the shell to execute in order. The shell script file must define the shell used to run the script. This is done in the first line of the script file, using the *#!* symbol, followed by the full path of the shell.

Within the shell script you can reference environment variable values by using a dollar sign in front of the variable. You can also define your own variables for use within the script, and assign values and even the output of a command by using the backtick character or the *\$()* format. The variable value can be used within the script by placing a dollar sign in front of the variable name.

The bash shell allows you to redirect both the input and output of a command from the standard behavior. You can redirect the output of any command from the monitor display to a file by using the greater-than symbol, followed by the name of the file to capture the output. You can append output data to an existing file by using two greater-than symbols.

Part II: Shell Scripting Basics

The less-than symbol is used to redirect input to a command. You can redirect input from a file to a command.

The Linux `pipe` command (the broken bar symbol) allows you to redirect the output of a command directly to the input of another command. The Linux system runs both commands at the same time, sending the output of the first command to the input of the second command without using any redirect files.

The bash shell provides a couple of ways for you to perform mathematical operations in your shell scripts. The `expr` command is a simple way to perform integer math. In the bash shell, you can also perform basic math calculations by enclosing equations in square brackets, preceded by a dollar sign. To perform floating-point arithmetic, you need to utilize the `bc` calculator command, redirecting input from inline data and storing the output in a user variable.

Finally, the chapter discussed how to use the exit status in your shell script. Every command that runs in the shell produces an exit status. The exit status is an integer value between 0 and 255 that indicates if the command completed successfully, and if not, what the reason may have been. An exit status of 0 indicates that the command completed successfully. You can use the `exit` command in your shell script to declare a specific exit status upon the completion of your script.

So far in your shell scripts, things have proceeded in an orderly fashion from one command to the next. In the next chapter, you'll see how you can use some logic flow control to alter which commands are executed within the script.

Using Structured Commands

IN THIS CHAPTER

Working with the if-then statement

Nesting ifs

Understanding the test command

Testing compound conditions

Using double brackets and parentheses

Looking at case

In the shell scripts presented in Chapter 11, the shell processed each individual command in the shell script in the order it appeared. This works out fine for sequential operations, where you want all the commands to process in the proper order. However, this isn't how all programs operate.

Many programs require some sort of logic flow control between the commands in the script. There is a whole command class that allows the script to skip over executed commands based on tested conditions. These commands are generally referred to as *structured commands*.

The structured commands allow you to alter the operation flow of a program. Quite a few structured commands are available in the bash shell, so we'll look at them individually. In this chapter, we look at `if-then` and `case` statements.

Working with the if-then Statement

The most basic type of structured command is the `if-then` statement. The `if-then` statement has the following format:

```
if command
then
    commands
fi
```

If you're using `if-then` statements in other programming languages, this format may be somewhat confusing. In other programming languages, the object after the `if` statement is an equation that is evaluated for a `TRUE` or `FALSE` value. That's not how the bash shell `if` statement works.

Part II: Shell Scripting Basics

The bash shell `if` statement runs the command defined on the `if` line. If the exit status of the command (see Chapter 11) is zero (the command completed successfully), the commands listed under the `then` section are executed. If the exit status of the command is anything else, the `then` commands aren't executed, and the bash shell moves on to the next command in the script. The `fi` statement delineates the `if-then` statement's end.

Here's a simple example to demonstrate this concept:

```
$ cat test1.sh
#!/bin/bash
# testing the if statement
if pwd
then
    echo "It worked"
fi
$
```

This script uses the `pwd` command on the `if` line. If the command completes successfully, the `echo` statement should display the text string. When you run this script from the command line, you get the following results:

```
$ ./test1.sh
/home/Christine
It worked
$
```

The shell executed the `pwd` command listed on the `if` line. Because the exit status was zero, it also executed the `echo` statement listed in the `then` section.

Here's another example:

```
$ cat test2.sh
#!/bin/bash
# testing a bad command
if IamNotaCommand
then
    echo "It worked"
fi
echo "We are outside the if statement"
$
$ ./test2.sh
./test2.sh: line 3: IamNotaCommand: command not found
We are outside the if statement
$
```

In this example, we deliberately used a command, `IamNotaCommand`, that does not work in the `if` statement line. Because this is a bad command, it produces an exit status that's non-zero, and the bash shell skips the `echo` statement in the `then` section. Also notice that the error message generated from running the command in the `if` statement still

appears in the script's output. There may be times when you don't want an error statement to appear. Chapter 15 discusses how this can be avoided.

NOTE

You might see an alternative form of the `if-then` statement used in some scripts:

```
if command; then
    commands
fi
```

By putting a semicolon at the end of the command to evaluate, you can include the `then` statement on the same line, which looks closer to how `if-then` statements are handled in some other programming languages.

You are not limited to just one command in the `then` section. You can list commands just as in the rest of the shell script. The `bash` shell treats the commands as a block, executing all of them when the command in the `if` statement line returns a zero exit status or skipping all of them when the command returns a non-zero exit status:

```
$ cat test3.sh
#!/bin/bash
# testing multiple commands in the then section
#
testuser=Christine
#
if grep $testuser /etc/passwd
then
    echo "This is my first command"
    echo "This is my second command"
    echo "I can even put in other commands besides echo:"
    ls -a /home/$testuser/.b*
fi
$
```

The `if` statement line uses the `grep` comment to search the `/etc/passwd` file to see if a specific username is currently used on the system. If there's a user with that logon name, the script displays some text and then lists the `bash` files in the user's *HOME* directory:

```
$ ./test3.sh
Christine:x:501:501:Christine B:/home/Christine:/bin/bash
This is my first command
This is my second command
I can even put in other commands besides echo:
/home/Christine/.bash_history /home/Christine/.bash_profile
/home/Christine/.bash_logout /home/Christine/.bashrc
$
```

However, if you set the `testuser` variable to a user that doesn't exist on the system, nothing happens:

```
$ cat test3.sh
#!/bin/bash
# testing multiple commands in the then section
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "This is my first command"
    echo "This is my second command"
    echo "I can even put in other commands besides echo:"
    ls -a /home/$testuser/.b*
fi
$
$ ./test3.sh
$
```

It's not all that exciting. It would be nice if we could display a little message saying that the username wasn't found on the system. Well, we can, using another feature of the `if-then` statement.

Exploring the if-then-else Statement

In the `if-then` statement, you have only one option for whether a command is successful. If the command returns a non-zero exit status code, the bash shell just moves on to the next command in the script. In this situation, it would be nice to be able to execute an alternate set of commands. That's exactly what the `if-then-else` statement is for.

The `if-then-else` statement provides another group of commands in the statement:

```
if command
then
    commands
else
    commands
fi
```

When the command in the `if` statement line returns with a zero exit status code, the commands listed in the `then` section are executed, just as in a normal `if-then` statement. When the command in the `if` statement line returns a non-zero exit status code, the bash shell executes the commands in the `else` section.

Now you can copy and modify the test script to include an `else` section:

```
$ cp test3.sh test4.sh
$
$ nano test4.sh
$
```

```
$ cat test4.sh
#!/bin/bash
# testing the else section
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "The bash files for user $testuser are:"
    ls -a /home/$testuser/.b*
    echo
else
    echo "The user $testuser does not exist on this system."
    echo
fi
$
$ ./test4.sh
The user NoSuchUser does not exist on this system.

$
```

That's more user-friendly. Just like the `then` section, the `else` section can contain multiple commands. The `fi` statement delineates the end of the `else` section.

Nesting ifs

Sometimes, you must check for several situations in your script code. For these situations, you can nest the `if-then` statements:

To check if a logon name is not in the `/etc/passwd` file and yet a directory for that user still exists, use a nested `if-then` statement. In this case, the nested `if-then` statement is within the primary `if-then-else` statement's `else` code block:

```
$ ls -d /home/NoSuchUser/
/home/NoSuchUser/
$
$ cat test5.sh
#!/bin/bash
# Testing nested ifs
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system."
else
    echo "The user $testuser does not exist on this system."
```


Part II: Shell Scripting Basics

```
    if ls -d /home/$testuser/
    then
        echo "However, $testuser has a directory."
    fi
fi
$
$ ./test5.sh
The user NoSuchUser does not exist on this system.
/home/NoSuchUser/
However, NoSuchUser has a directory.
$
```

The script correctly finds that although the login name has been removed from the `/etc/passwd` file, the user's directory is still on the system. The problem with using this manner of nested `if-then` statements in a script is that the code can get hard to read, and the logic flow becomes difficult to follow.

Instead of having to write separate `if-then` statements, you can use an alternative version of the `else` section, called `elif`. The `elif` continues an `else` section with another `if-then` statement:

```
if command1
then
    commands
elif command2
then
    more commands
fi
```

The `elif` statement line provides another command to evaluate, similar to the original `if` statement line. If the exit status code from the `elif` command is zero, `bash` executes the commands in the second `then` statement section. Using this method of nesting provides cleaner code with an easier-to-follow logic flow:

```
$ cat test5.sh
#!/bin/bash
# Testing nested ifs - use elif
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system."
#
elif ls -d /home/$testuser
then
    echo "The user $testuser does not exist on this system."
    echo "However, $testuser has a directory."
#
```

```
fi
$
$ ./test5.sh
/home/NoSuchUser
The user NoSuchUser does not exist on this system.
However, NoSuchUser has a directory.
$
```

You can even take this script a step further and have it check for both a non-existent user with a directory and a non-existent user without a directory. This is accomplished by adding an `else` statement within the nested `elif`:

```
$ cat test5.sh
#!/bin/bash
# Testing nested ifs - use elif & else
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system."
#
elif ls -d /home/$testuser
then
    echo "The user $testuser does not exist on this system."
    echo "However, $testuser has a directory."
#
else
    echo "The user $testuser does not exist on this system."
    echo "And, $testuser does not have a directory."
fi
$
$ ./test5.sh
/home/NoSuchUser
The user NoSuchUser does not exist on this system.
However, NoSuchUser has a directory.
$
$ sudo rmdir /home/NoSuchUser
[sudo] password for Christine:
$
$ ./test5.sh
ls: cannot access /home/NoSuchUser: No such file or directory
The user NoSuchUser does not exist on this system.
And, NoSuchUser does not have a directory.
$
```

Before the `/home/NoSuchUser` directory was removed and the test script executed the `elif` statement, a zero exit status was returned. Thus, the statements within the `elif`'s `then` code block were executed. After the `/home/NoSuchUser` directory was removed, a

non-zero exit status was returned for the `elif` statement. This caused the statements in the `else` block within the `elif` block to be executed.

TIP

Keep in mind that, with an `elif` statement, any `else` statements immediately following it are for that `elif` code block. They are not part of a preceding `if-then` statement code block.

You can continue to string `elif` statements together, creating one huge `if-then-elif` conglomeration:

```
if command1
then
    command set 1
elif command2
then
    command set 2
elif command3
then
    command set 3
elif command4
then
    command set 4
fi
```

Each block of commands is executed depending on which command returns the zero exit status code. Remember that the bash shell executes the `if` statements in order, and only the first one that returns a zero exit status results in the `then` section being executed.

Even though the code looks cleaner with `elif` statements, it still can be confusing to follow the script's logic. Later in the "Considering the case Command" section, you'll see how to use the `case` command instead of having to nest lots of `if-then` statements.

Trying the test Command

So far, all you've seen in the `if` statement line are normal shell commands. You might be wondering if the bash `if-then` statement has the ability to evaluate any condition other than a command's exit status code.

The answer is no, it can't. However, there's a neat utility available in the bash shell that helps you evaluate other things, using the `if-then` statement.

The `test` command provides a way to test different conditions in an `if-then` statement. If the condition listed in the `test` command evaluates to `TRUE`, the `test` command exits with a zero exit status code. This makes the `if-then` statement behave in much the same

way that `if-then` statements work in other programming languages. If the condition is `FALSE`, the `test` command exits with a non-zero exit status code, which causes the `if-then` statement to exit.

The format of the `test` command is pretty simple:

```
test condition
```

The *condition* is a series of parameters and values that the `test` command evaluates. When used in an `if-then` statement, the `test` command looks like this:

```
if test condition
then
    commands
fi
```

If you leave out the *condition* portion of the `test` command statement, it exits with a non-zero exit status code and triggers any `else` block statements:

```
$ cat test6.sh
#!/bin/bash
# Testing the test command
#
if test
then
    echo "No expression returns a True"
else
    echo "No expression returns a False"
fi
$
$ ./test6.sh
No expression returns a False
$
```

When you add in a condition, it is tested by the `test` command. For example, using the `test` command, you can determine whether a variable has content. A simple condition expression is needed to determine whether a variable has content:

```
$ cat test6.sh
#!/bin/bash
# Testing the test command
#
my_variable="Full"
#
if test $my_variable
then
    echo "The $my_variable expression returns a True"
#
else
```

Part II: Shell Scripting Basics

```
        echo "The $my_variable expression returns a False"
    fi
$
$ ./test6.sh
The Full expression returns a True
$
```

The variable `my_variable` contains content (`Full`), so when the `test` command checks the condition, the exit status returns a zero. This triggers the statement in the `then` code block.

As you would suspect, the opposite occurs when the variable does not contain content:

```
$ cat test6.sh
#!/bin/bash
# Testing the test command
#
my_variable=""
#
if test $my_variable
then
    echo "The $my_variable expression returns a True"
#
else
    echo "The $my_variable expression returns a False"
fi
$
$ ./test6.sh
The expression returns a False
$
```

The bash shell provides an alternative way of testing a condition without declaring the `test` command in an `if-then` statement:

```
if [ condition ]
then
    commands
fi
```

The square brackets define the test condition. Be careful; you *must* have a space after the first bracket and a space before the last bracket, or you'll get an error message.

The `test` command and test conditions can evaluate three classes of conditions:

- Numeric comparisons
- String comparisons
- File comparisons

The next sections describe how to use each of these test classes in your `if-then` statements.

Using numeric comparisons

The most common test evaluation method is to perform a comparison of two numeric values. Table 12-1 shows the list of condition parameters used for testing two values.

TABLE 12-1 The test Numeric Comparisons

Comparison	Description
<code>n1 -eq n2</code>	Checks if <code>n1</code> is equal to <code>n2</code>
<code>n1 -ge n2</code>	Checks if <code>n1</code> is greater than or equal to <code>n2</code>
<code>n1 -gt n2</code>	Checks if <code>n1</code> is greater than <code>n2</code>
<code>n1 -le n2</code>	Checks if <code>n1</code> is less than or equal to <code>n2</code>
<code>n1 -lt n2</code>	Checks if <code>n1</code> is less than <code>n2</code>
<code>n1 -ne n2</code>	Checks if <code>n1</code> is not equal to <code>n2</code>

The numeric test conditions can be used to evaluate both numbers and variables. Here's an example of doing that:

```
$ cat numeric_test.sh
#!/bin/bash
# Using numeric test evaluations
#
value1=10
value2=11
#
if [ $value1 -gt 5 ]
then
    echo "The test value $value1 is greater than 5"
fi
#
if [ $value1 -eq $value2 ]
then
    echo "The values are equal"
else
    echo "The values are different"
fi
#
$
```

The first test condition:

```
if [ $value1 -gt 5 ]
```

tests if the value of the variable *value1* is greater than 5. The second test condition:

```
if [ $value1 -eq $value2 ]
```

tests if the value of the variable *value1* is equal to the value of the variable *value2*. Both numeric test conditions evaluate as expected:

```
$ ./numeric_test.sh
The test value 10 is greater than 5
The values are different
$
```

There is a limitation to the test numeric conditions concerning floating-point values:

```
$ cat floating_point_test.sh
#!/bin/bash
# Using floating point numbers in test evaluations
#
value1=5.555
#
echo "The test value is $value1"
#
if [ $value1 -gt 5 ]
then
    echo "The test value $value1 is greater than 5"
fi
#
$ ./floating_point_test.sh
The test value is 5.555
./floating_point_test.sh: line 8:
[: 5.555: integer expression expected
$
```

This example uses a floating-point value, stored in the *value1* variable. Next, it evaluates the value. Something obviously went wrong.

Remember that the only numbers the bash shell can handle are integers. This works perfectly fine if all you need to do is display the result, using an `echo` statement. However, this doesn't work in numeric-oriented functions, such as our numeric test condition. The bottom line is that you cannot use floating-point values for test conditions.

Using string comparisons

Test conditions also allow you to perform comparisons on string values. Performing comparisons on strings can get tricky, as you'll see. Table 12-2 shows the comparison functions you can use to evaluate two string values.

TABLE 12-2 The test String Comparisons

Comparison	Description
<code>str1 = str2</code>	Checks if <code>str1</code> is the same as string <code>str2</code>
<code>str1 != str2</code>	Checks if <code>str1</code> is not the same as <code>str2</code>
<code>str1 < str2</code>	Checks if <code>str1</code> is less than <code>str2</code>
<code>str1 > str2</code>	Checks if <code>str1</code> is greater than <code>str2</code>
<code>-n str1</code>	Checks if <code>str1</code> has a length greater than zero
<code>-z str1</code>	Checks if <code>str1</code> has a length of zero

The following sections describe the different string comparisons available.

Looking at string equality

The equal and not equal conditions are fairly self-explanatory with strings. It's pretty easy to know when two string values are the same or not:

```
$ cat test7.sh
#!/bin/bash
# testing string equality
testuser=rich
#
if [ $USER = $testuser ]
then
    echo "Welcome $testuser"
fi
$
$ ./test7.sh
Welcome rich
$
```

Also, using the not equals string comparison allows you to determine if two strings have the same value or not:

```
$ cat test8.sh
#!/bin/bash
# testing string equality
testuser=baduser
#
if [ $USER != $testuser ]
then
    echo "This is not $testuser"
else
    echo "Welcome $testuser"
fi
```



```
$
$ ./test8.sh
This is not baduser
$
```

Keep in mind that the test comparison takes all punctuation and capitalization into account when comparing strings for equality.

Looking at string order

Trying to determine if one string is less than or greater than another is where things start getting tricky. Two problems often plague shell programmers when trying to use the greater-than or less-than features of test conditions:

- The greater-than and less-than symbols must be escaped, or the shell uses them as redirection symbols, with the string values as filenames.
- The greater-than and less-than order is not the same as that used with the `sort` command.

The first item can result in a huge problem that often goes undetected when programming your scripts. Here's an example of what sometimes happens to novice shell script programmers:

```
$ cat badtest.sh
#!/bin/bash
# mis-using string comparisons
#
val1=baseball
val2=hockey
#
if [ $val1 > $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./badtest.sh
baseball is greater than hockey
$ ls -l hockey
-rw-r--r--  1 rich      rich          0 Sep 30 19:08 hockey
$
```

By just using the greater-than symbol itself in the script, no errors are generated, but the results are wrong. The script interpreted the greater-than symbol as an output redirection (see Chapter 15). Thus, it created a file called `hockey`. Because the redirection completed successfully, the test condition returns a zero exit status code, which the `if` statement evaluates as though things completed successfully!

To fix this problem, you need to properly escape the greater-than symbol:

```
$ cat test9.sh
#!/bin/bash
# mis-using string comparisons
#
vall=baseball
val2=hockey
#
if [ $vall \> $val2 ]
then
    echo "$vall is greater than $val2"
else
    echo "$vall is less than $val2"
fi
$
$ ./test9.sh
baseball is less than hockey
$
```

Now that answer is more along the lines of what you would expect from the string comparison.

The second issue is a little more subtle, and you may not even run across it unless you are working with uppercase and lowercase letters. The `sort` command handles uppercase letters opposite to the way the test conditions consider them:

```
$ cat test9b.sh
#!/bin/bash
# testing string sort order
vall=Testing
val2=testing
#
if [ $vall \> $val2 ]
then
    echo "$vall is greater than $val2"
else
    echo "$vall is less than $val2"
fi
$
$ ./test9b.sh
Testing is less than testing
$
$ sort testfile
testing
Testing
$
```

Part II: Shell Scripting Basics

Capitalized letters are treated as less than lowercase letters in test comparisons. However, the `sort` command does the opposite. When you put the same strings in a file and use the `sort` command, the lowercase letters appear first. This is due to different ordering techniques.

Test comparisons use standard ASCII ordering, using each character's ASCII numeric value to determine the sort order. The `sort` command uses the sorting order defined for the system locale language settings. For the English language, the locale settings specify that lowercase letters appear before uppercase letters in sorted order.

NOTE

The `test` command and test expressions use the standard mathematical comparison symbols for string comparisons and text codes for numerical comparisons. This is a subtle feature that many programmers manage to get reversed. If you use the mathematical comparison symbols for numeric values, the shell interprets them as string values and may not produce the correct results.

Looking at string size

The `-n` and `-z` comparisons are handy when trying to evaluate whether a variable contains data:

```
$ cat test10.sh
#!/bin/bash
# testing string length
val1=testing
val2=''
#
if [ -n $val1 ]
then
    echo "The string '$val1' is not empty"
else
    echo "The string '$val1' is empty"
fi
#
if [ -z $val2 ]
then
    echo "The string '$val2' is empty"
else
    echo "The string '$val2' is not empty"
fi
#
if [ -z $val3 ]
then
    echo "The string '$val3' is empty"
else
    echo "The string '$val3' is not empty"
fi
```

```

$
$ ./test10.sh
The string 'testing' is not empty
The string '' is empty
The string '' is empty
$

```

This example creates two string variables. The `val1` variable contains a string, and the `val2` variable is created as an empty string. The following comparisons are made as shown below:

```
if [ -n $val1 ]
```

The preceding code determines whether the `val1` variable is non-zero in length, which it is, so its `then` section is processed.

```
if [ -z $var2 ]
```

This preceding code determines whether the `val2` variable is zero in length, which it is, so its `then` section is processed.

```
if [ -z $val3 ]
```

The preceding determines whether the `val3` variable is zero in length. This variable was never defined in the shell script, so it indicates that the string length is still zero, even though it wasn't defined.

TIP

Empty and uninitialized variables can have catastrophic effects on your shell script tests. If you're not sure of the contents of a variable, it's always best to test if the variable contains a value using `-n` or `-z` before using it in a numeric or string comparison.

Using file comparisons

The last category of test comparisons is quite possibly the most powerful and most used comparisons in shell scripting. This category allows you to test the status of files and directories on the Linux filesystem. Table 12-3 lists these comparisons.

TABLE 12-3 The test File Comparisons

Comparison	Description
<code>-d file</code>	Checks if <code>file</code> exists and is a directory
<code>-e file</code>	Checks if <code>file</code> exists
<code>-f file</code>	Checks if <code>file</code> exists and is a file

Continues

TABLE 12.3 (continued)

Comparison	Description
<code>-r file</code>	Checks if <code>file</code> exists and is readable
<code>-s file</code>	Checks if <code>file</code> exists and is not empty
<code>-w file</code>	Checks if <code>file</code> exists and is writable
<code>-x file</code>	Checks if <code>file</code> exists and is executable
<code>-O file</code>	Checks if <code>file</code> exists and is owned by the current user
<code>-G file</code>	Checks if <code>file</code> exists and the default group is the same as the current user
<code>file1 -nt file2</code>	Checks if <code>file1</code> is newer than <code>file2</code>
<code>file1 -ot file2</code>	Checks if <code>file1</code> is older than <code>file2</code>

These conditions give you the ability to check filesystem files within shell scripts. They are often used in scripts that access files. Because they're used so often, let's look at each of these individually.

Checking directories

The `-d` test checks to see if a specified directory exists on the system. This is usually a good thing to do if you're trying to write a file to a directory or before you try to change to a directory location:

```
$ cat test11.sh
#!/bin/bash
# Look before you leap
#
jump_directory=/home/arthur
#
if [ -d $jump_directory ]
then
    echo "The $jump_directory directory exists"
    cd $jump_directory
    ls
else
    echo "The $jump_directory directory does not exist"
fi
#
$
$ ./test11.sh
The /home/arthur directory does not exist
$
```

The `-d` test condition checks to see if the `jump_directory` variable's directory exists. If it does, it proceeds to use the `cd` command to change to the current directory and performs a directory listing. If it does not, the script emits a warning message and exits the script.

Checking whether an object exists

The `-e` comparison allows you to check if either a file or directory object exists before you attempt to use it in your script:

```
$ cat test12.sh
#!/bin/bash
# Check if either a directory or file exists
#
location=$HOME
file_name="sentinel"
#
if [ -e $location ]
then #Directory does exist
    echo "OK on the $location directory."
    echo "Now checking on the file, $file_name."
    #
    if [ -e $location/$file_name ]
    then #File does exist
        echo "OK on the filename"
        echo "Updating Current Date..."
        date >> $location/$file_name
    #
    else #File does not exist
        echo "File does not exist"
        echo "Nothing to update"
    fi
#
else #Directory does not exist
    echo "The $location directory does not exist."
    echo "Nothing to update"
fi
#
$
$ ./test12.sh
OK on the /home/Christine directory.
Now checking on the file, sentinel.
File does not exist
Nothing to update
$
$ touch sentinel
$
$ ./test12.sh
OK on the /home/Christine directory.
Now checking on the file, sentinel.
OK on the filename
Updating Current Date...
$
```

The first check uses the `-e` comparison to determine whether the user has a `$HOME` directory. If so, the next `-e` comparison checks to determine whether the `sentinel` file exists in the `$HOME` directory. If the file doesn't exist, the shell script notes that the file is missing and that there is nothing to update.

To ensure that the update will work, the `sentinel` file was created and the shell script was run a second time. This time when the conditions are tested, both the `$HOME` and the `sentinel` file are found, and the current date and time is appended to the file.

Checking for a file

The `-e` comparison works for both files and directories. To be sure that the object specified is a file and not a directory, you must use the `-f` comparison:

```
$ cat test13.sh
#!/bin/bash
# Check if either a directory or file exists
#
item_name=$HOME
echo
echo "The item being checked: $item_name"
echo
#
if [ -e $item_name ]
then #Item does exist
    echo "The item, $item_name, does exist."
    echo "But is it a file?"
    echo
    #
    if [ -f $item_name ]
    then #Item is a file
        echo "Yes, $item_name is a file."
    #
    else #Item is not a file
        echo "No, $item_name is not a file."
    fi
fi
#
else #Item does not exist
    echo "The item, $item_name, does not exist."
    echo "Nothing to update"
fi
#
$ ./test13.sh

The item being checked: /home/Christine

The item, /home/Christine, does exist.
But is it a file?

No, /home/Christine is not a file.
$
```

This little script does lots of checking! First, it uses the `-e` comparison to test whether `$HOME` exists. If it does, it uses `-f` to test whether it's a file. If it isn't a file (which of course it isn't), a message is displayed stating that it is not a file.

A slight modification to the variable, `item_name`, replacing the directory `$HOME` with a file, `$HOME/sentinel`, causes a different outcome:

```
$ nano test13.sh
$
$ cat test13.sh
#!/bin/bash
# Check if either a directory or file exists
#
item_name=$HOME/sentinel
[...]
$
$ ./test13.sh

The item being checked: /home/Christine/sentinel

The item, /home/Christine/sentinel, does exist.
But is it a file?

Yes, /home/Christine/sentinel is a file.
$
```

The `test13.sh` script listing is snipped, because the only item changed in the shell script was the `item_name` variable's value. Now when the script is run, the `-f` test on `$HOME/sentinel` exits with a zero status, triggering the `then` statement, which in turn outputs the message `Yes, /home/Christine/sentinel is a file.`

Checking for read access

Before trying to read data from a file, it's usually a good idea to test whether you can read from the file first. You do this with the `-r` comparison:

```
$ cat test14.sh
#!/bin/bash
# testing if you can read a file
pwfile=/etc/shadow
#
# first, test if the file exists, and is a file
if [ -f $pwfile ]
then
    # now test if you can read it
    if [ -r $pwfile ]
```



```
        then
            tail $pwfile
        else
            echo "Sorry, I am unable to read the $pwfile file"
        fi
    else
        echo "Sorry, the file $file does not exist"
    fi
fi
$
$ ./test14.sh
Sorry, I am unable to read the /etc/shadow file
$
```

The `/etc/shadow` file contains the encrypted passwords for system users, so it's not readable by normal users on the system. The `-r` comparison determined that read access to the file wasn't allowed, so the test command failed and the bash shell executed the `else` section of the `if-then` statement.

Checking for empty files

You should use `-s` comparison to check whether a file is empty, especially if you don't want to remove a non-empty file. Be careful because when the `-s` comparison succeeds, it indicates that a file has data in it:

```
$ cat test15.sh
#!/bin/bash
# Testing if a file is empty
#
file_name=$HOME/sentinel
#
if [ -f $file_name ]
then
    if [ -s $file_name ]
    then
        echo "The $file_name file exists and has data in it."
        echo "Will not remove this file."
    #
    else
        echo "The $file_name file exists, but is empty."
        echo "Deleting empty file..."
        rm $file_name
    fi
else
    echo "File, $file_name, does not exist."
fi
#
$ ls -l $HOME/sentinel
-rw-rw-r--. 1 Christine Christine 29 Jun 25 05:32 /home/Christine/sentinel
```

```
$
$ ./test15.sh
The /home/Christine/sentinel file exists and has data in it.
Will not remove this file.
$
```

First, the `-f` comparison tests whether the file exists. If it does exist, the `-s` comparison is triggered to determine whether the file is empty. An empty file will be deleted. You can see from the `ls -l` that the `sentinel` file is not empty, and therefore the script does not delete it.

Checking whether you can write to a file

The `-w` comparison determines whether you have permission to write to a file. The `test16.sh` script is simply an update of the `test13.sh` script. Now instead of just checking whether the `item_name` exists and is a file, the script also checks to see whether it has permission to write to the file:

```
$ cat test16.sh
#!/bin/bash
# Check if a file is writable.
#
item_name=$HOME/sentinel
echo
echo "The item being checked: $item_name"
echo
[...]
    echo "Yes, $item_name is a file."
    echo "But is it writable?"
    echo
    #
    if [ -w $item_name ]
    then #Item is writable
        echo "Writing current time to $item_name"
        date +%H%M >> $item_name
    #
    else #Item is not writable
        echo "Unable to write to $item_name"
    fi
#
else #Item is not a file
    echo "No, $item_name is not a file."
fi
[...]
$
$ ls -l sentinel
-rw-rw-r--. 1 Christine Christine 0 Jun 27 05:38 sentinel
$
```

Part II: Shell Scripting Basics

```
$ ./test16.sh

The item being checked: /home/Christine/sentinel

The item, /home/Christine/sentinel, does exist.
But is it a file?

Yes, /home/Christine/sentinel is a file.
But is it writable?

Writing current time to /home/Christine/sentinel
$
$ cat sentinel
0543
$
```

The `item_name` variable is set to `$HOME/sentinel`, and this file allows user write access (see Chapter 7 for more information on file permissions). Thus, when the script is run, the `-w` test expression returns a non-zero exit status and the `then` code block is executed, which writes a time stamp into the `sentinel` file.

When the `sentinel` file user's write access is removed via `chmod`, the `-w` test expression returns a non-zero status, and a time stamp is not written to the file:

```
$ chmod u-w sentinel
$
$ ls -l sentinel
-r--rw-r--. 1 Christine Christine 5 Jun 27 05:43 sentinel
$
$ ./test16.sh

The item being checked: /home/Christine/sentinel

The item, /home/Christine/sentinel, does exist.
But is it a file?

Yes, /home/Christine/sentinel is a file.
But is it writable?

Unable to write to /home/Christine/sentinel
$
```

The `chmod` command could be used again to grant the write permission back for the user. This would make the write test expression return a zero exit status and allow a write attempt to the file.

Checking whether you can run a file

The `-x` comparison is a handy way to determine whether you have execute permission for a specific file. Although this may not be needed for most commands, if you run lots of scripts from your shell scripts, it could be useful:

```
$ cat test17.sh
#!/bin/bash
# testing file execution
#
if [ -x test16.sh ]
then
    echo "You can run the script: "
    ./test16.sh
else
    echo "Sorry, you are unable to execute the script"
fi
$
$ ./test17.sh
You can run the script:
[...]
$
$ chmod u-x test16.sh
$
$ ./test17.sh
Sorry, you are unable to execute the script
$
```

This example shell script uses the `-x` comparison to test whether you have permission to execute the `test16.sh` script. If so, it runs the script. After successfully running the `test16.sh` script the first time, the permissions were changed. This time, the `-x` comparison failed, because execute permission had been removed for the `test16.sh` script.

Checking ownership

The `-O` comparison allows you to easily test whether you're the owner of a file:

```
$ cat test18.sh
#!/bin/bash
# check file ownership
#
if [ -O /etc/passwd ]
then
    echo "You are the owner of the /etc/passwd file"
else
    echo "Sorry, you are not the owner of the /etc/passwd file"
fi
```

Part II: Shell Scripting Basics

```
$
$ ./test18.sh
Sorry, you are not the owner of the /etc/passwd file
$
```

The script uses the `-o` comparison to test whether the user running the script is the owner of the `/etc/passwd` file. The script is run under a normal user account, so the test fails.

Checking default group membership

The `-G` comparison checks the default group of a file, and it succeeds if it matches the group of the default group for the user. This can be somewhat confusing because the `-G` comparison checks the default groups only and not all the groups to which the user belongs. Here's an example of this:

```
$ cat test19.sh
#!/bin/bash
# check file group test
#
if [ -G $HOME/testing ]
then
    echo "You are in the same group as the file"
else
    echo "The file is not owned by your group"
fi
$
$ ls -l $HOME/testing
-rw-rw-r-- 1 rich rich 58 2014-07-30 15:51 /home/rich/testing
$
$ ./test19.sh
You are in the same group as the file
$
$ chgrp sharing $HOME/testing
$
$ ./test19
The file is not owned by your group
$
```

The first time the script is run, the `$HOME/testing` file is in the `rich` group, and the `-G` comparison succeeds. Next, the group is changed to the `sharing` group, of which the user is also a member. However, the `-G` comparison failed, because it compares only the default groups, not any additional group memberships.

Checking file date

The last set of comparisons deal with comparing the creation times of two files. This comes in handy when writing scripts to install software. Sometimes, you don't want to install a file that is older than a file already installed on the system.

The `-nt` comparison determines whether a file is newer than another file. If a file is newer, it has a more recent file creation time. The `-ot` comparison determines whether a file is older than another file. If the file is older, it has an older file creation time:

```
$ cat test20.sh
#!/bin/bash
# testing file dates
#
if [ test19.sh -nt test18.sh ]
then
    echo "The test19 file is newer than test18"
else
    echo "The test18 file is newer than test19"
fi
if [ test17.sh -ot test19.sh ]
then
    echo "The test17 file is older than the test19 file"
fi
$
$ ./test20.sh
The test19 file is newer than test18
The test17 file is older than the test19 file
$
$ ls -l test17.sh test18.sh test19.sh
-rwxrw-r-- 1 rich rich 167 2014-07-30 16:31 test17.sh
-rwxrw-r-- 1 rich rich 185 2014-07-30 17:46 test18.sh
-rwxrw-r-- 1 rich rich 167 2014-07-30 17:50 test19.sh
$
```

The file paths used in the comparisons are relative to the directory from which you run the script. This can cause problems if the files being checked are moved around. Another problem is that neither of these comparisons checks whether the file exists first. Try this test:

```
$ cat test21.sh
#!/bin/bash
# testing file dates
#
if [ badfile1 -nt badfile2 ]
then
    echo "The badfile1 file is newer than badfile2"
else
    echo "The badfile2 file is newer than badfile1"
fi
$
$ ./test21.sh
The badfile2 file is newer than badfile1
$
```

This little example demonstrates that if the files don't exist, the `-nt` comparison just returns a failed condition. It's imperative to ensure that the files exist before trying to use them in the `-nt` or `-ot` comparison.

Considering Compound Testing

The `if-then` statement allows you to use Boolean logic to combine tests. You can use these two Boolean operators:

- `[condition1] && [condition2]`
- `[condition1] || [condition2]`

The first Boolean operation uses the AND Boolean operator to combine two conditions. Both conditions must be met for the `then` section to execute.

TIP

Boolean logic is a method that reduces the potential returned values to be either `TRUE` or `FALSE`.

The second Boolean operation uses the OR Boolean operator to combine two conditions. If either condition evaluates to a `TRUE` condition, the `then` section is executed.

The following shows the AND Boolean operator in use:

```
$ cat test22.sh
#!/bin/bash
# testing compound comparisons
#
if [ -d $HOME ] && [ -w $HOME/testing ]
then
    echo "The file exists and you can write to it"
else
    echo "I cannot write to the file"
fi
$
$ ./test22.sh
I cannot write to the file
$
$ touch $HOME/testing
$
$ ./test22.sh
The file exists and you can write to it
$
```

Using the AND Boolean operator, both of the comparisons must be met. The first comparison checks to see if the `$HOME` directory exists for the user. The second comparison checks to

see if there's a file called `testing` in the user's `$HOME` directory, and if the user has write permissions for the file. If either of these comparisons fails, the `if` statement fails and the shell executes the `else` section. If both of the comparisons succeed, the `if` statement succeeds, and the shell executes the `then` section.

Working with Advanced if-then Features

Two additions to the bash shell provide advanced features that you can use in `if-then` statements:

- Double parentheses for mathematical expressions
- Double square brackets for advanced string handling functions

The following sections describe each of these features in more detail.

Using double parentheses

The *double parentheses* command allows you to incorporate advanced mathematical formulas in your comparisons. The `test` command allows for only simple arithmetic operations in the comparison. The double parentheses command provides more mathematical symbols, which programmers who have used other programming languages may be familiar with using. Here's the format of the double parentheses command:

```
(( expression ))
```

The *expression* term can be any mathematical assignment or comparison expression. Besides the standard mathematical operators that the `test` command uses, Table 12-4 shows the list of additional operators available for use in the double parentheses command.

TABLE 12-4 The Double Parentheses Command Symbols

Symbol	Description
<code>val++</code>	Post-increment
<code>val--</code>	Post-decrement
<code>++val</code>	Pre-increment
<code>--val</code>	Pre-decrement
<code>!</code>	Logical negation
<code>~</code>	Bitwise negation
<code>**</code>	Exponentiation

Continues

TABLE 12.4 (continued)

Symbol	Description
<<	Left bitwise shift
>>	Right bitwise shift
&	Bitwise Boolean AND
	Bitwise Boolean OR
&&	Logical AND
	Logical OR

You can use the double parentheses command in an `if` statement, as well as in a normal command in the script for assigning values:

```
$ cat test23.sh
#!/bin/bash
# using double parenthesis
#
val1=10
#
if (( $val1 ** 2 > 90 ))
then
    (( val2 = $val1 ** 2 ))
    echo "The square of $val1 is $val2"
fi
$
$ ./test23.sh
The square of 10 is 100
$
```

Notice that you don't need to escape the greater-than symbol in the expression within the double parentheses. This is yet another advanced feature besides the double parentheses command.

Using double brackets

The *double bracket* command provides advanced features for string comparisons. Here's the double bracket command format:

```
[[ expression ]]
```

The double bracketed *expression* uses the standard string comparison used in the test evaluations. However, it provides an additional feature that the test evaluations don't — *pattern matching*.

NOTE

Double brackets work fine in the bash shell. Be aware, however, that not all shells support double brackets.

In pattern matching, you can define a regular expression (discussed in detail in Chapter 20) that's matched against the string value:

```
$ cat test24.sh
#!/bin/bash
# using pattern matching
#
if [[ $USER == r* ]]
then
    echo "Hello $USER"
else
    echo "Sorry, I do not know you"
fi
$
$ ./test24.sh
Hello rich
$
```

Notice in the preceding script that double equal signs (==) are used. These double equal signs designate the string to the right (`r*`) as a pattern, and pattern matching rules are applied. The double bracket command matches the `$USER` environment variable to see whether it starts with the letter `r`. If so, the comparison succeeds, and the shell executes the `then` section commands.

Considering the case Command

Often, you'll find yourself trying to evaluate a variable's value, looking for a specific value within a set of possible values. In this scenario, you end up having to write a lengthy `if-then-else` statement, like this:

```
$ cat test25.sh
#!/bin/bash
# looking for a possible value
#
if [ $USER = "rich" ]
then
    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = "barbara" ]
then
    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = "testing" ]
then
    echo "Special testing account"
elif [ $USER = "jessica" ]
```

Part II: Shell Scripting Basics

```
then
    echo "Do not forget to logout when you're done"
else
    echo "Sorry, you are not allowed here"
fi
$
$ ./test25.sh
Welcome rich
Please enjoy your visit
$
```

The `elif` statements continue the `if-then` checking, looking for a specific value for the single comparison variable.

Instead of having to write all the `elif` statements to continue checking the same variable value, you can use the `case` command. The `case` command checks multiple values of a single variable in a list-oriented format:

```
case variable in
    pattern1 | pattern2) commands1;;
    pattern3) commands2;;
    *) default commands;;
esac
```

The `case` command compares the variable specified against the different patterns. If the variable matches the pattern, the shell executes the commands specified for the pattern. You can list more than one pattern on a line, using the bar operator to separate each pattern. The asterisk symbol is the catch-all for values that don't match any of the listed patterns. Here's an example of converting the `if-then-else` program to using the `case` command:

```
$ cat test26.sh
#!/bin/bash
# using the case command
#
case $USER in
    rich | barbara)
        echo "Welcome, $USER"
        echo "Please enjoy your visit";;
    testing)
        echo "Special testing account";;
    jessica)
        echo "Do not forget to log off when you're done";;
    *)
        echo "Sorry, you are not allowed here";;
esac
$
$ ./test26.sh
```

```
Welcome, rich
Please enjoy your visit
$
```

The `case` command provides a much cleaner way of specifying the various options for each possible variable value.

Summary

Structured commands allow you to alter the normal flow of shell script execution. The most basic structured command is the `if-then` statement. This statement provides a command evaluation and performs other commands based on the evaluated command's output.

You can expand the `if-then` statement to include a set of commands the bash shell executes if the specified command fails as well. The `if-then-else` statement executes commands only if the command being evaluated returns a non-zero exit status code.

You can also link `if-then-else` statements together, using the `elif` statement. The `elif` is equivalent to using an `else if` statement, providing for additional checking of whether the original command that was evaluated failed.

In most scripts, instead of evaluating a command, you'll want to evaluate a condition, such as a numeric value, the contents of a string, or the status of a file or directory. The `test` command provides an easy way for you to evaluate all these conditions. If the condition evaluates to a `TRUE` condition, the `test` command produces a zero exit status code for the `if-then` statement. If the condition evaluates to a `FALSE` condition, the `test` command produces a non-zero exit status code for the `if-then` statement.

The square bracket is a special bash command that is a synonym for the `test` command. You can enclose a test condition in square brackets in the `if-then` statement to test for numeric, string, and file conditions.

The double parentheses command provides advanced mathematical evaluations using additional operators. The double square bracket command allows you to perform advanced string pattern-matching evaluations.

Finally, the chapter discussed the `case` command, which is a shorthand way of performing multiple `if-then-else` commands, checking the value of a single variable against a list of values.

The next chapter continues the discussion of structured commands by examining the shell looping commands. The `for` and `while` commands let you create loops that iterate through commands for a given period of time.