CHAPTER 12

Creating Scripts

A script is a program written in an interpreted language, typically associated with a shell or other program whose primary purpose is something other than as an interpreted language. In Linux, many scripts are *shell scripts*, which are associated with Bash or another shell. (If you're familiar with *batch files* in DOS or Windows, scripts serve a similar purpose.) You can write shell scripts to help automate tedious repetitive tasks or to perform new and complex tasks. Many of Linux's startup functions are performed by scripts, so mastering scripting will help you manage the startup process.

This chapter covers Bash shell scripts, beginning with the task of creating a new script file. I then describe several important scripting features that help you to perform progressively more complex scripting tasks.

- **Beginning a shell script**
- **Using commands**
- **Using arguments**
- **Using variables**
- **Using conditional expressions**
- **Using loops**
- **Using functions**
- **Setting the script's exit value**

Learning More

Like any programming task, shell scripting can be quite complex. Consequently, this chapter barely scratches the surface of what you can accomplish through shell scripting. Consult a book on the topic, such as Cameron Newham's *Learning the Bash Shell, 3rd Edition* (O'Reilly, 2005) or Richard Blum's *Linux Command Line and Shell Scripting Bible* (Wiley, 2008), for more information.

Beginning a Shell Script

Certification Objective

Shell scripts are plain-text files, so you create them in text editors such as Vi, nano, or pico, as described in Chapter 11, "Editing Files." A shell script begins with a line that identifies the shell that's used to run it, such as the following:

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

Certification Objective

This chapter describes Bash shell scripts. Simple Bash scripts can run in other shells, but more complex scripts are more likely to be shell-specific.

The first two characters are a special code that tells the Linux kernel that this is a script and to use the rest of the line as a pathname to the program that's to interpret the script. (This line is sometimes called the *shebang*, *hashbang*, *hashpling*, or *pound bang* line.) Shell scripting languages use a hash mark (#) as a comment character, so the script utility ignores this line, although the kernel doesn't. On most systems, `/bin/sh` is a symbolic link that points to `/bin/bash`, but it can point to some other shell. Specifying the script as using `/bin/sh` guarantees that any Linux system will have a shell program to run the script, but if the script uses any features specific to a particular shell, you should specify that shell instead—for instance, use `/bin/bash` or `/bin/tcsh` instead of `/bin/sh`.

When you're done writing the shell script, you should modify it so that it's executable. You do this with the `chmod` command, which is described in more detail in Chapter 15, "Setting Ownership and Permissions." For now know that you use the `a+x` option to add execute permissions for all users. For instance, to make a file called `my-script` executable, you should issue the following command:

```
$ chmod a+x my-script
```

You'll then be able to execute the script by typing its name, possibly preceded by `./` to tell Linux to run the script in the current directory rather than searching the current path. If you fail to make the script executable, you can still run the script by running the shell program followed by the script name (as in **`bash my-script`**), but it's generally better to make the script executable. If the script is one you run regularly, you may want to move it to a location on your path, such as `/usr/local/bin`. When you do that, you won't have to type the complete path or move to the script's directory to execute it; you can just type **`my-script`**.

Using Commands

One of the most basic features of shell scripts is the ability to run commands. You can use both commands that are built into the shell and external commands—that is, you can run other programs as commands. Most of the

commands you type in a shell prompt are external commands—they're programs located in `/bin`, `/usr/bin`, and other directories on your path. You can run such programs, as well as internal commands, by including their names in the script. You can also specify parameters to such programs in a script. For instance, suppose you want a script that launches two `xterm` windows and the KMail mail reader program. Listing 12.1 presents a shell script that accomplishes this goal.

Listing 12.1: A simple script that launches three programs

```
#!/bin/bash
/usr/bin/xterm &
/usr/bin/xterm &
/usr/bin/kmail &
```

Aside from the first line that identifies it as a script, the script looks just like the commands you might type to accomplish the task manually, except for one fact: The script lists the complete paths to each program. This is usually not strictly necessary, but listing the complete path ensures that the script will find the programs even if the `PATH` environment variable changes. On the other hand, if the program files move (say, because you upgrade the package from which they're installed and the packager decides to move them), scripts that use complete paths will break. If a script produces a `No such file or directory` error for a command, typing **`which`** ***`command`***, where *command* is the offending command, should help you locate it.

Fedora plans to rearrange its directory tree starting with Fedora 17. Once this is done, most program files will reside in `/usr/bin`.

Each program-launch line in Listing 12.1 ends in an ampersand (`&`). This character tells the shell to go on to the next line without waiting for the first to finish. If you omit the ampersands in Listing 12.1, the effect will be that the first `xterm` will open but the second won't open until the first is closed. Likewise, KMail won't start until the second `xterm` terminates.

Although launching several programs from one script can save time in startup scripts and some other situations, scripts are also frequently used to run a series of programs that manipulate data in some way. Such scripts typically do *not* include the ampersands at the ends of the commands because one command must run after another or may even rely on output from the first. A comprehensive list of such commands is impossible because you can run any program you can install in Linux as a command in a script—even another script. A few commands that are commonly used in scripts include the following:

Normal file manipulation commands The file manipulation commands, such as `ls`, `mv`, `cp`, and `rm`, are often used in scripts. You can use these commands to help automate repetitive file maintenance tasks.

grep This command is described in Chapter 10, "Searching, Extracting, and Archiving Data." It locates files that contain the string you specify, or displays the lines that contain those strings in a single file.

find Where grep searches for patterns within the contents of files, find does so based on filenames, ownership, and similar characteristics. Chapter 10 covers this command.

cut This command extracts text from fields in a file. It's frequently used to extract variable information from a file whose contents are highly patterned. To use it, you pass it one or more options that specify what information you want, followed by one or more filenames. For instance, users' home directories appear in the sixth colon-delimited field of the /etc/passwd file. You can therefore type **cut -f 6 -d ":" /etc/passwd** to extract this information. The same command in a script will extract this information, which you'll probably save to a variable or pass to a subsequent command.

sed This program provides many of the capabilities of a conventional text editor (such as search-and-replace operations) but via commands that can be typed at a command prompt or entered in a script.

Certification Objective

echo Sometimes a script must provide a message to the user; echo is the tool to accomplish this goal. You can pass various options to echo or just a string to be shown to the user. For instance, `echo "Press the Enter key"` causes a script to display the specified string. You can also use echo to display the value of variables (described later, in "Using Variables").

mail The mail command can be used to send e-mail from within a script. Pass it the `-s` *subject* parameter to specify a subject line, and give it an e-mail address as the last argument. If it's used at the command line, you then type a message and terminate it with a Ctrl+D keystroke. If it's used from a script, you might omit the subject entirely or pass it an external file as the message using input redirection. You might want to use this command to send mail to the superuser about the actions of a startup script or a script that runs on an automated basis.

Chapter 10 describes input redirection.

Many of these commands are extremely complex, and completely describing them is beyond the scope of this chapter. You can consult these commands' man pages for more information. A few of them are described elsewhere in this book, as noted in their descriptions.

Even if you have a full grasp of how to use some key external commands, simply executing commands as you might when typing them at a command prompt is of limited utility. Many administrative tasks require you to modify what you type at a command, or even what commands you enter, depending on information from other commands. For this reason, scripting languages include additional features to help you make your scripts useful.

Using Arguments

Certification Objective

Variables can help you expand the utility of scripts. A variable is a placeholder in a script for a value that will be determined when the script runs. Variables' values can be passed as parameters to a script, generated internally to a script, or extracted from a script's environment. (An *environment* is a set of variables that any program can access. The environment includes things like the current directory and the search path for running programs.)

Variables that are passed to the script are frequently called *parameters* or *arguments*. They're represented in the script by a dollar sign ($) followed by a number from 0 up—$0 stands for the name of the script, $1 is the first parameter to the script, $2 is the second parameter, and so on. To understand how this might be useful, consider the task of adding a user. As described in Chapter 14, "Creating Users and Groups," creating an account for a new user typically involves running at least two commands—`useradd` and `passwd`. You may also need to run additional site-specific commands, such as commands that create unusual user-owned directories aside from the user's home directory.

As an example of how a script with an argument variable can help in such situations, consider Listing 12.2. This script creates an account and changes the account's password (the script prompts you to enter the password when you run the script). It creates a directory in the `/shared` directory tree corresponding to the account, and it sets a symbolic link to that directory from the new user's home directory. It also adjusts ownership and permissions in a way that may be useful, depending on your system's ownership and permissions policies.

Listing 12.2: A script that reduces account-creation tedium


```
#!/bin/bash
useradd -m $1
passwd $1
mkdir -p /shared/$1
chown $1.users /shared/$1
chmod 775 /shared/$1
ln -s /shared/$1 /home/$1/shared
chown $1.users /home/$1/shared
```

If you use Listing 12.2, you need to type only three things: the script name with the desired username and the password (twice). For instance, if the script is called `mkuser`, you can use it like this:

```
# mkuser ajones
Changing password for user ajones
New password:
Retype new password:
passwd: all authentication tokens updated successfully
```

Most of the script's programs operate silently unless they encounter problems, so the interaction (including typing the passwords, which don't echo to the screen) is a result of just the `passwd` command. In effect, Listing 12.2's script replaces seven lines of commands with one. Every one of those lines uses the username, so by running this script, you also reduce the chance of a typo causing problems.

Using Variables

Certification Objective

Another type of variable is assigned within scripts—for instance, such variables can be set from the output of a command. These variables are also identified by leading dollar signs, but they're typically given names that at least begin with a letter, such as `$Addr` or `$Name`. (When values are assigned to variables, the dollar sign is omitted, as illustrated shortly.) You can then use these variables in conjunction with normal commands as if they were command parameters, but the value of the variable is passed to the command.

For instance, consider Listing 12.3, which checks to see if the computer's router is up with the help of the `ping` utility. This script uses two variables. The first is `$ip`, which is extracted from the output of `route` using the `grep`, `tr`, and `cut` commands. When you're assigning a value to a variable from the output of a command, that command should be enclosed in backtick characters (`), which appear on the same key as the tilde (~) on most keyboards. These are *not* ordinary single quotes, which appear on the same key as the regular quote character (") on most keyboards. The second variable, `$ping`, simply points to the `ping` program. It can just as easily be omitted, with subsequent uses of `$ping` replaced by the full path to the program or simply by `ping` (relying on the `$PATH` environment variable to find the program). Variables like this are sometimes used to make it easier to modify the script in the future. For instance, if you move the `ping` program, you need to modify only one line of the script. Variables can also be used in conjunction with conditionals to ensure that the script works on more systems—for instance, if `ping` were called something else on some systems.

In addition to several commands, the `ip=` line uses backticks (`) to assign the output of that command chain to `ip`. Chapter 10 describes this technique.

Listing 12.3: Script demonstrating assignment and use of variables

```
#!/bin/bash
ip=`route -n | grep UG | tr -s " " | cut -f 2 -d " "`
ping="/bin/ping"
echo "Checking to see if $ip is up..."
$ping -c 5 $ip
```

In practice, you use Listing 12.3 by typing the script's name. The result should be the message `Checking to see if` *`192.168.1.1`* `is up` (with

192.168.1.1 replaced by the computer's default gateway system) and the output from the `ping` command, which should attempt to send five packets to the router. If the router is up and is configured to respond to pings, you'll see five return packets and summary information, similar to the following:

```
$ routercheck
Checking to see if 192.168.1.1 is up...
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=63 time=23.0 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=63 time=0.176 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=63 time=0.214 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=63 time=0.204 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=63 time=0.191 ms

--- 192.168.1.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.176/4.758/23.005/9.123 ms
```


If the router is down, you'll see error messages to the effect that the host was unreachable.

Listing 12.3 is of limited practical use and contains bugs. For instance, the script identifies the computer's gateway merely by the presence of the string `UG` in the router's output line from `route`. If a computer has two routers defined, this won't work correctly, and the result is likely to be a script that misbehaves. The purpose of Listing 12.3 is to illustrate how variables can be assigned and used, not to be a flawless working script.

Scripts like Listing 12.3, which obtain information from running one or more commands, are useful in configuring features that rely on system-specific information or information that varies with time. You can use a similar approach to obtain the current hostname (using the `hostname` command), the current time (using `date`), the total time the computer's been running (using `uptime`), free disk space (using `df`), and so on. When combined with conditional expressions (described shortly), variables become even more powerful because then your script can perform one action when one condition is met and another in some other case. For instance, a script that installs software can check free disk space and abort the installation if insufficient disk space is available.

Certification Objective

In addition to assigning variables with the assignment operator (=), you can read variables from standard input using `read`, as in `read response` to read input for subsequent access as `$response`. This method of variable assignment is useful for scripts that must interact with users. For instance, instead of reading the username from the command line, Listing 12.2 may be modified to prompt the user for the username. Listing 12.4 shows the result. To use this script, you type its name *without* typing a username on the command line. The

script will then prompt for a username, and after you enter one, the script will attempt to create an account with that name.

Listing 12.4: Modified version of Listing 12.2 that employs user interaction

```
#!/bin/bash
echo -n "Enter a username: "
read name
useradd -m $name
passwd $name
mkdir -p /shared/$name
chown $name.users /shared/$name
chmod 775 /shared/$name
ln -s /shared/$name /home/$name/shared
chown $name.users /home/$name/shared
```


One special type of variable is an environment variable, which is assigned and accessed just like a shell script variable. The difference is that the script or command that sets an environment variable uses Bash's `export` command to make the value of the variable accessible to programs launched from the shell or shell script that made the assignment. In other words, you can set an environment variable in one script and use it in another script that the first script launches. Environment variables are most often set in shell startup scripts, but the scripts you use can access them. For instance, if your script calls X programs, it might check for the presence of a valid `$DISPLAY` environment variable and abort if it finds that this variable isn't set. By convention, environment variable names are all uppercase, whereas non-environment shell script variables are all lowercase or mixed case.

One special variable deserves mention: `$?`. This variable holds the *exit status* (or *return value*) of the most recently executed command. Most programs return a value of 0 when they terminate normally and return another value to specify errors. You can display this value with `echo` or use it in a conditional expression (described next) to have your script perform special error handling.

Consult a program's `man` page to learn the meanings of its return values.

Using Conditional Expressions

Certification Objective

Scripting languages support several types of *conditional expressions*. These enable a script to perform one of several actions contingent on some condition—typically the value of a variable. One common command that uses conditional expressions is `if`, which allows the system to take one of two actions depending on whether some condition is true. The `if` keyword's conditional expression appears in brackets after the `if` keyword and can take many forms.

For instance, `-f file` is true if *file* exists and is a regular file; `-s file` is true if *file* exists and has a size greater than 0; and *string1* == *string2* is true if the two strings have the same values.

Conditionals may be combined together with the logical and (&&) or logical or (||) operators. When conditionals are combined with &&, both sides of the operator must be true for the condition as a whole to be true. When || is used, if either side of the operator is true, the condition as a whole is true.

Certification Objective

To better understand the use of conditionals, consider the following code fragment:

```
if [ -s /tmp/tempstuff ]
   then
      echo "/tmp/tempstuff found; aborting!"
      exit
fi
```

This fragment causes the script to exit if the file `/tmp/tempstuff` is present. The `then` keyword marks the beginning of a series of lines that execute only if the conditional is true, and `fi` (`if` backward) marks the end of the `if` block. Such code may be useful if the script creates and then later deletes this file, because its presence indicates that a previous run of the script didn't succeed or is still underway.

Certification Objective

An alternative form for a conditional expression uses the `test` keyword rather than square brackets around the conditional:

```
if test -s /tmp/tempstuff
```

You can also test a command's return value by using the command as the condition:

```
if [ command ]
   then
      additional-commands
fi
```

In this example, the *additional-commands* will be run only if *command* completes successfully. If *command* returns an error code, the *additional-commands* won't be run.

Conditional expressions may be expanded by use of the `else` clause:

```
if [ conditional-expression ]
   then
      commands
   else
      other-commands
fi
```

Code of this form causes either *commands* or *other-commands* to execute, depending on the evaluation of *conditional-expression*. This is useful if *something* should happen in a part of the program, but precisely what should happen depends on some condition. For instance, you may want to launch one of two different file archiving programs depending on a user's input.

Certification Objective

What do you do if more than two outcomes are possible—for instance, if a user may provide any one of four possible inputs? You can nest several `if/then/else` clauses, but this gets awkward quickly. A cleaner approach is to use `case`:

```
case word in
   pattern1) command(s) ;;
   pattern2) command(s) ;;
   ...
esac
```

Certification Objective

For a `case` statement, a *word* is likely to be a variable, and each *pattern* is a possible value of that variable. The patterns can be expanded much like filenames, using the same wildcards and expansion rules (* to stand for any string, for instance). You can match an arbitrary number of patterns in this way. Each set of commands must end with a double semicolon (`;;`), and the `case` statement as a whole ends in the string `esac` (`case` backward).

Filename expansion using asterisks (*), question marks (?), and so on is sometimes called *globbing*.

Upon execution, `bash` executes the commands associated with the first pattern to match the *word*. Execution then jumps to the line following the `esac` statement; any intervening commands don't execute. If no patterns match the word, no code within the `case` statement executes. If you want to have a default condition, use * as the final *pattern*; this pattern matches any *word*, so its commands will execute if no other *pattern* matches.

Using Loops

Conditional expressions are sometimes used in *loops*. Loops are structures that tell the script to perform the same task repeatedly until some condition is met (or until some condition is no longer met). For instance, Listing 12.5 shows a loop that plays all the `.wav` audio files in a directory.

The `aplay` command is a basic audio file player. On some systems, you may need to use `play` or some other command instead of `aplay`.

Listing 12.5: A script that executes a command on every matching file in a directory

```
#!/bin/bash
for d in `ls *.wav` ; do
   aplay $d
done
```

The `for` loop as used here executes once for every item in the list generated by `ls *.wav`. Each of those items (filenames) is assigned in turn to the `$d` variable and so is passed to the `aplay` command.

The `seq` command can be useful in creating `for` loops (and in other ways, too): This command generates a list of numbers starting from its first argument and continuing to its last one. For instance, typing **`seq 1 10`** generates 10 lines, each with a number between 1 and 10. You can use a `for` loop beginning `for x in` `` `seq 1 10` `` to have the loop execute 10 times, with the value of `x` incrementing with each iteration. If you pass just one parameter to `seq`, it interprets that number as an ending point, with the starting point being `1`. If you pass three values to `seq`, it interprets them as a starting value, an increment amount, and an ending value.

Another type of loop is the `while` loop, which executes for as long as its condition is true. The basic form of this loop type is like this:

Certification Objective

```
while [ condition ]
do
   commands
done
```

The `until` loop is similar in form, but it continues execution for as long as its condition is *false*—that is, until the condition becomes true.

Using Functions

A *function* is a part of a script that performs a specific sub-task and that can be called by name from other parts of the script. Functions are defined by placing parentheses after the function name and enclosing the lines that make up the function within curly braces:


```
myfn() {
   commands
}
```

The keyword `function` may optionally precede the function name. In either event, the function is called by name as if it were an ordinary internal or external command.

Functions are very useful in helping to create modular scripts. For instance, if your script needs to perform half a dozen distinct computations, you can place each computation in a function and then call them all in sequence. Listing 12.6 demonstrates the use of functions in a simple program that copies a file but aborts with an error message if the target file already exists. This script accepts a target and a destination filename and must pass those filenames to the functions.

Listing 12.6: A script demonstrating the use of functions

```
#/bin/bash

doit() {
   cp $1 $2
}

function check() {
  if [ -s $2 ]
     then
        echo "Target file exists! Exiting!"
        exit
  fi
}

check $1 $2
doit $1 $2
```

If you enter Listing 12.6 and call it `safercp`, you can use it like this, assuming the file `original.txt` exists and `dest.txt` doesn't:

```
$ ./safercp original.txt dest.txt
$ ./safercp original.txt dest.txt
Target file exists! Exiting!
```

The first run of the script succeeded because `dest.txt` didn't exist. On the second run, though, the destination file did exist, so the script terminated with the error message.

Note that the functions aren't run directly and in the order in which they appear in the script. They're run only when called in the main body of the script—which in Listing 12.6 consists of just two lines, each corresponding to one function call, at the very end of the script.

Setting the Script's Exit Value

Certification Objective

Ordinarily, a script's return value is the same as the last command the script called—that is, the script returns `$?`. You can control the exit value, however, or exit from the script at any point, by using the `exit` command. Used without any options, `exit` causes immediate termination of the script, with the usual exit value of `$?`. This can be useful in error handling or in aborting an ongoing operation for any reason—if the script detects an error or if the user selects an option to terminate, you can call `exit` to quit.

If you pass a numeric value between `0` and `255` to `exit`, the script terminates and returns the specified value as the script's own exit value. You can use this feature to signal errors to other scripts that might call your own script. You may

have to include extra code to keep track of the causes of abnormal termination, though. For instance, you might set aside a variable (say, `$termcause`) to hold the cause of the script's termination. Set it to `0` at the start of the script and then, if the script detects a problem that will cause termination, reset `$termcause` to some non-`0` value. (You can use any numeric codes you like; there's no set meaning for such codes.) On exit, be sure to pass `$termcause` to `exit`:

```
exit $termcause
```

The Essentials and Beyond

Serious Linux users and administrators must have at least a basic understanding of shell scripts. Many configuration and startup files are in fact shell scripts, and being able to read them, and perhaps modify them, will help you administer your system. Being able to create new shell scripts is also important, because doing so will help you simplify tedious tasks and create site-specific tools by gluing together multiple programs to accomplish your goals.

Suggested Exercises

- Write a script that copies a file by prompting the user to enter the source and destination filenames rather than by accepting them as arguments on the command line, as `cp` does.
- Some text editors leave backup files with filenames that end in tildes (~). Write a script that, when you pass it a directory name as an argument, locates all such files in that directory. The script should then ask the user whether to delete each file individually and do so if and only if the user responds by typing `Y`.

Review Questions

1. After using a text editor to create a shell script, what step should you take before trying to use the script by typing its name?

   A. Set one or more executable bits using `chmod`.

   B. Copy the script to the `/usr/bin/scripts` directory.

   C. Compile the script by typing **`bash`** ***`scriptname`***, where *`scriptname`* is the script's name.

   D. Run a virus checker on the script to be sure it contains no viruses.

   E. Run a spell checker on the script to ensure it contains no bugs.

(Continues)

The Essentials and Beyond *(Continued)*

2. Describe the effect of the following short script, `cp1`, if it's called as **`cp1 big.c big.cc`**:

```
#!/bin/bash
cp $2 $1
```

   A. It has the same effect as the `cp` command—copying the contents of `big.c` to `big.cc`.

   B. It compiles the C program `big.c` and calls the result `big.cc`.

   C. It copies the contents of `big.cc` to `big.c`, eliminating the old `big.c`.

   D. It converts the C program `big.c` into a C++ program called `big.cc`.

   E. The script's first line is invalid, so it won't work.

3. What is the purpose of conditional expressions in shell scripts?

   A. They prevent scripts from executing if license conditions aren't met.

   B. They display information about the script's computer environment.

   C. They enable the script to take different actions in response to variable data.

   D. They enable scripts to learn in a manner reminiscent of Pavlovian conditioning.

   E. They cause scripts to run only at specified times of day.

4. True or false: A user types **`myscript laser.txt`** to run a script called `myscript`. Within `myscript`, the `$0` variable holds the value `laser.txt`.

5. True or false: Valid looping statements in Bash include `for`, `while`, and `until`.

6. True or false: The following script launches three simultaneous instances of the `terminal` program.

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

7. You've written a simple shell script that does nothing but launch programs. In order to ensure that the script works with most user shells, what should its first line read?

8. What command can you use to display prompts for a user in a shell script?

9. What Bash scripting command can you use to control the program flow based on a variable that can take many values (such as all the letters of the alphabet)?

CHAPTER 13

Understanding Users and Groups

Linux is a multi-user OS, meaning that it provides features to help multiple individuals use the computer. Collectively, these features constitute *accounts*. Previous chapters of this book have referred to accounts in passing but haven't covered them in detail. This chapter changes that; it describes important account principles and a few commands you can use to begin investigating accounts. Related to accounts are *groups*, which are collections of accounts that can be given special permissions on the computer, so this chapter also describes groups. One account, known as `root`, has special privileges on the computer. You use this account to perform most system administration tasks, so you should understand this account before you tackle the administrative tasks described in the last few chapters of this book.

- **Understanding accounts**
- **Using account tools**
- **Working as `root`**

Understanding Accounts

Even a single-user workstation uses multiple accounts. Such a computer may have just one *user account*, but several *system accounts* help keep the computer running.

Accounts enable multiple users to share a single computer without causing each other too much trouble. They also enable system administrators to track who is using system resources and, sometimes, who is doing things they shouldn't be doing. Thus, account features help users use a computer and administrators administer it. Understanding these features is the basis for enabling you to manage accounts.

Some account features help you identify accounts and the files and resources associated with them. Knowing how to use these features will help you track down account-related problems and manage users of a computer.