

INTRODUCTION TO SHELL SCRIPT PROGRAMMING

After reading this chapter and completing the exercises, you will be able to:

- ◆ Understand the program development cycle
- ◆ Compare UNIX/Linux shells for creating scripts
- ◆ Use shell variables, operators, and wildcard characters
- ◆ Use shell logic structures
- ◆ Employ shell scripting to create a menu
- ◆ Use commands to help debug shell scripts
- ◆ Explain ways to customize your personal environment
- ◆ Use the *trap* command
- ◆ Develop a menu-based application

Shell script programming is a greatly valued ability among UNIX and Linux users, programmers, and administrators because it gives flexibility in creating applications of all kinds. Some users create scripts to generate reports from data files. Others use scripts to create and maintain data files, such as for tracking projects, finances, or people. Still others perform system maintenance tasks through scripts, including monitoring who is logged in or backing up files.

The focus of this chapter is to develop your shell script programming skills and to show you how to build a menu-based application. You begin by getting an overview of the application you will build and of the program development cycle. Next, you learn about shell script programming tools that include using variables, operators, and logic structures. Finally, in the Hands-on Projects, you put to work what you've learned by building a menu-based application.

PREVIEWING THE APPLICATION

As you learned in Chapters 4 and 5 (“UNIX/Linux File Processing” and “Advanced File Processing”), commands such as *grep*, *cut*, *paste*, and *awk* are powerful commands for manipulating data. Although these commands are powerful, they can be difficult for nontechnical users, in part because they often must be combined in long sequences to achieve the results you want. Repeatedly executing these command sequences can be cumbersome, even for experienced technical users. You’ve discovered in earlier chapters that shell scripts can help eliminate these problems.

One advantage of shell scripts is that you can create them to present user-friendly screens—for example, screens that automatically issue commands such as *grep* and *awk* to extract, format, and display information. This gives nontechnical users access to powerful features of UNIX/Linux. For your own use, shell scripts save time by automating long command sequences that you must perform often.

The shell script application you develop in this chapter and enhance in Chapter 7, “Advanced Shell Programming,” is a simulated employee information system that stores and displays employee data—such as you might commonly find in a human resources system in an organization. It presents a menu of operations from which the user can choose. Among other tasks, these operations automate the process of inputting, searching for, formatting, and displaying employee records. For preliminary testing, you create and use a data file that contains a sampling of employee records, similar to one that an experienced shell programmer might use for testing.

As you learn the tools needed to develop your application in this chapter, you gain experience with the following scripting and programming features of the UNIX/Linux shell:

- *Shell variables*—Your scripts often need to keep values in memory for later use. **Shell variables** temporarily store values in memory for use by a shell script. They use symbolic names that can access the values stored in memory. In this case, a **symbolic name** is a name consisting of letters, numbers, or characters and is used to reference the contents of a variable; often the name reflects a variable’s purpose or contents.
- *Shell script operators*—Shell scripts support many **shell script operators**, including those for assigning the contents of a shell variable, for evaluating information, for performing mathematical operations, and for piping or redirection of input/output.
- *Logic or control structures*—Shell scripts support **logic structures** (also called **control structures**), including sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to a different point), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives).

In addition, you learn special commands for formatting screen output and positioning the cursor. Before you begin writing your application, it is important to understand more about the program development cycle and the basic elements of programming.

THE PROGRAM DEVELOPMENT CYCLE

The process of developing an application is known as the **program development cycle**. The steps involved in the cycle are the same whether you are writing shell scripts or high-level language programs.

The process begins by creating program specifications—the requirements the application must meet. The specifications determine what data the application takes as input, the processes that must be performed on the data, and the correct output.

After you determine the specifications, the design process begins. During this process, programmers create file formats, screen layouts, and algorithms. An **algorithm** is a sequence of procedures, programming code, or commands that result in a program or that can be used as part of a program. Programmers use a variety of tools to design complex applications. You learn about some of the tools in this chapter and about additional tools in Chapter 7.

After the design process is complete, programmers begin writing the actual code, which they must then test and debug. **Debugging** is the process of going through program code to locate errors and then fix them. When programmers find errors, they correct them and begin the testing process again. This procedure continues until the application performs satisfactorily.

Figure 6-1 illustrates the program development cycle.

Using High-Level Languages

Computer programs are instructions often written using a high-level language, such as COBOL, Visual Basic, C, or C++. A **high-level language** is a computer language that uses English-like expressions. For example, the following COBOL statement instructs the computer to add 1 to the variable COUNTER:

```
ADD 1 TO COUNTER.
```

Here is a similar statement, written in C++:

```
counter = counter + 1;
```

A program's high-level language statements are stored in a file called the **source file**. This is the file that the programmer creates with an editor such as vi or Emacs. The source file cannot execute, however, because the computer can only process instructions written in low-level machine language. As you recall from Chapter 3, "Mastering Editors," machine-language instructions are cryptic codes expressed in binary numbers. Therefore, the high-level source file must be converted into a low-level machine language file, as described next.

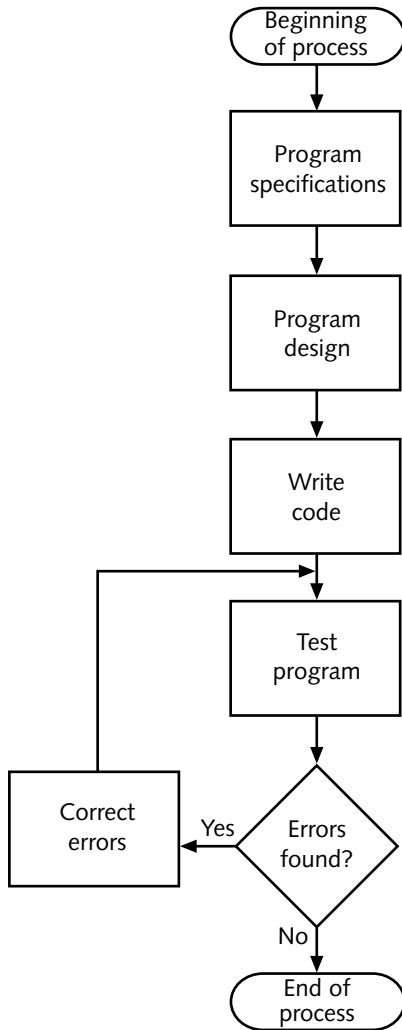


Figure 6-1 Program development cycle

The source file is converted into an executable machine-language file by a program called a compiler. The **compiler** reads the lines of code that the programmer wrote in the source file and converts them to the appropriate machine-language instructions. For example, the Linux C and C++ compilers are named *gcc* and *g++*. The following command illustrates how to compile the C++ source code file, *datecalc.C*, so that you can run it as the program *datecalc*:

```
g++ datecalc.C -o datecalc
```

In this sample command, the *-o* option followed by *datecalc* instructs the compiler to create an executable file, *datecalc*. The source file is *datecalc.C*. The command causes the compiler

to translate the C++ program `datecalc.C` into an executable machine-language program, which is stored in the file `datecalc`. You learn more about C and C++ programming in Chapter 10, “Developing UNIX/Linux Applications in C and C++.”



As you learn in Chapter 10, some important differences exist between C and C++ source code, and therefore it is necessary to use the correct compiler (`gcc` versus `g++`). Remember, when you invoke one of these compilers in Linux, the `gcc` compiler expects C files to have the `.c` extension, whereas the `g++` compiler expects C++ files to have the `.C` extension.

If a source file contains **syntax errors** (grammatical mistakes in program language use), it cannot be converted into an executable file. The compiler locates and reports any syntax errors, which the programmer must correct.



After compiling, the executable program might still contain fatal run-time errors or logic errors. Fatal run-time errors cause the program to abort, for example, due to an invalid memory location specified in the program code. Logic errors cause the program to produce invalid results because of problems such as flawed mathematical statements.

Another way to accomplish programming tasks is to develop UNIX/Linux shell scripts, which you learn in this chapter.

Using UNIX/Linux Shell Scripts

First introduced in Chapter 4, UNIX/Linux shell scripts are text files that contain sequences of UNIX/Linux commands. Like high-level source files, a programmer creates shell scripts with a text editor. Unlike high-level language programs, shell scripts do not have to be converted into machine language by a compiler. This is because the UNIX/Linux shell acts as an **interpreter** when reading script files. As this interpreter reads the statements in a script file, it immediately translates them into executable instructions, and causes them to run. No executable file is produced because the interpreter translates and executes the scripted statements in one step. If a syntax error is encountered, the execution of the shell script halts.

After you create a shell script, you tell the operating system that the file can be executed. This is accomplished by using the `chmod` (“change mode”) command that you learned in Chapters 2 and 4 (“Exploring the UNIX/Linux File Systems and File Security” and “UNIX/Linux File Processing”) to change the file’s mode. The mode determines how the file can be used. Recall that modes can be denoted by single-letter codes: *r* (read), *w* (write), and *x* (execute). Further, the `chmod` command tells the computer who is allowed to use the file: the user or owner (*u*), the group (*g*), or all other users (*o*). For a description of the `chmod` command, see Appendix B, “Syntax Guide to UNIX/Linux Commands.”

Recall from Chapter 4 that you can change the mode of a file so that UNIX/Linux recognize it as an executable program (mode x) that everyone (user, group, and others) can use. In the following example, the user is the owner of the file:

```
$ chmod ugo+x filename <Enter>
```

Alternatively, you can make a file executable for all users by entering either:

```
$ chmod a+x filename <Enter>
```

or

```
$ chmod 755 filename <Enter>
```

In *chmod a+x*, the *a* stands for all and is the same as *ugo*. Also, remember from Chapter 2 that *chmod 755* gives owner (in the first position) read, write, and execute permissions (7). It also gives group (in the second position) read and execute permissions (5), and gives others (in the third position) read and execute permissions (5).

After you make the file executable, you can run it in one of several ways:

- You can simply type the name of the script at the system command prompt. However, before this method can work, you must modify your default directory path to include the directory in which the script resides. The directory might be the source or bin directory under your home directory. If you use this method, before any script or program can be run it must be retrieved from a path identified in the **PATH variable**, which provides a list of directory locations where UNIX or Linux looks to find executable scripts or programs. You learn how to temporarily modify the PATH variable in the “Variables” section later in this chapter; you learn how to permanently modify the PATH variable in Chapter 7.
- If the script resides in your current directory, which is not in the PATH variable, you can run the script by preceding the name with a dot slash (./) to tell UNIX/Linux to look in the current directory to find it, as follows:

```
$ ./filename <Enter>
```

- If the script does not reside in your current directory and is not in the PATH variable, you can run it by specifying the absolute path to the script. For example, if the script is in the data directory under your home directory, you can type either of the following (using Tom’s home directory as an example):

```
$ /home/tom/data/filename <Enter>
```

or

```
$ ~/data/filename <Enter>
```

Shell scripts run less quickly than compiled programs because the shell must interpret each UNIX/Linux command inside the executable script file before it is executed. Whether a programmer uses a script or a compiled program (such as a C++ program) is often related to several factors:

- Whether the programmer is more proficient in writing scripts than source code for a compiler

- Whether there is a need for the script or program to execute as quickly as possible, such as to reduce the load on the computer's resources when there are multiple users
- Whether the job is relatively complex; if so, a compiled program might offer more flexible options or features

Prototyping an Application

A **prototype** is a running model of your application, which lets you review the final results before committing to the design. Using a shell script to create a prototype is often the quickest and most efficient method because prototyping logic and design capabilities reside within UNIX/Linux.

After the working prototype is approved, the script can be rewritten to run faster using a compiled language such as C++. If the shell script performs well, however, you might not need to convert it to a compiled program.

Using Comments

In Chapter 5, you were introduced to using comments to provide documentation about a script. Plan to use comments in all of your scripts and programs, so that later it is easier to remember how they work.

Comment lines begin with a pound (#) symbol, such as in the following example from the pact script you created in Hands-on Project 5-16 in Chapter 5:

```
# =====
# Script Name:  pact
# By:           Your initials
# Date:        November 2009
# Purpose:     Create temporary file, pnum, to hold the
#              count of the number of projects each
#              programmer is working on. The pnum file
#              consists of:
#              prog_num and count fields
# =====
cut -d: -f4 project | sort | uniq -c | awk '{printf "%s:
%s\n", $2, $1}' > pnum
# cut prog_num, pipe output to sort to remove duplicates
# and get count for prog/projects.
# output file with prog_number followed by count
```

In this example, comment lines appear at the beginning of the script and after the *cut* command. You can place comment lines anywhere in a script to provide documentation. For example, in the Hands-on Projects for this chapter, you typically place comments at the beginning of a script to show the script name, the script's author, the date the script was written, and the script's purpose. As you write code in this and later chapters, insert any

additional comment lines that you believe might be helpful for later reference. Some examples of what you might comment include:

- Script name, author(s), creation date, and purpose
- Modification date(s) and the purpose of each modification
- The purpose and types of variables used (You learn about variables in this chapter.)
- Files that are accessed, created, or modified
- How logic structures work (You create logic structures in this chapter.)
- The purpose of shell functions (You create shell functions in Chapter 7.)
- How complex lines of code work
- The reasons for including specific commands

Although writing comments might take a little extra time, in the long run the comments can save you much more time when you need to modify that script or incorporate it in an application with other scripts.

THE PROGRAMMING SHELL

Before you create a script, choose the shell in which to run the script. As you learned in Chapter 1, UNIX/Linux versions support different shells and each shell has different capabilities. Also, recall that all Linux versions use the Bash shell (Bourne Again Shell) as the default shell. Table 6-1 lists the three shells that come with most Linux distributions, their derivations, and distinguishing features in relation to shell programming.

Table 6-1 Linux shells

Shell Name	Original Shell from Which Derived	Description in Terms of Shell Programming
Bash	Bourne and Korn shells	Offers strong scripting and programming language features, such as shell variables, logic structures, and math/logic expressions; combines the best features of the Bourne and Korn shells
csh/tcsh	C shell	Conforms to a scripting and programming language format; shell expressions use operators similar to those found in the C programming language
ksh/zsh	Korn shell	Is similar to the Bash shell in many respects, but also has syntax similar to that of C programming; useful if you are familiar with older Korn shell scripts

The Bash shell offers improved features over the older Bourne and Korn shells and is fully backward compatible with the Bourne shell. In addition, the Bash shell, when compared to the other shells, has a more powerful programming interface. For these reasons, you use the Bash shell for shell scripts in this book.

**TIP**

The manual pages in Fedora, Red Hat Enterprise Linux, and SUSE contain a generous amount of documentation about the Bash shell. Just enter *man bash* to access the documentation.

Now that you have selected the shell, it is important to learn about several basic features used by shell scripts, including variables, shell operators, and special characters.

6

VARIABLES

Variables use symbolic names that represent values stored in memory. The three types of variables discussed in this section are configuration variables, environment variables, and shell variables. **Configuration variables** are used to store information about the setup of the operating system, and after they are set up, you typically do not change them.

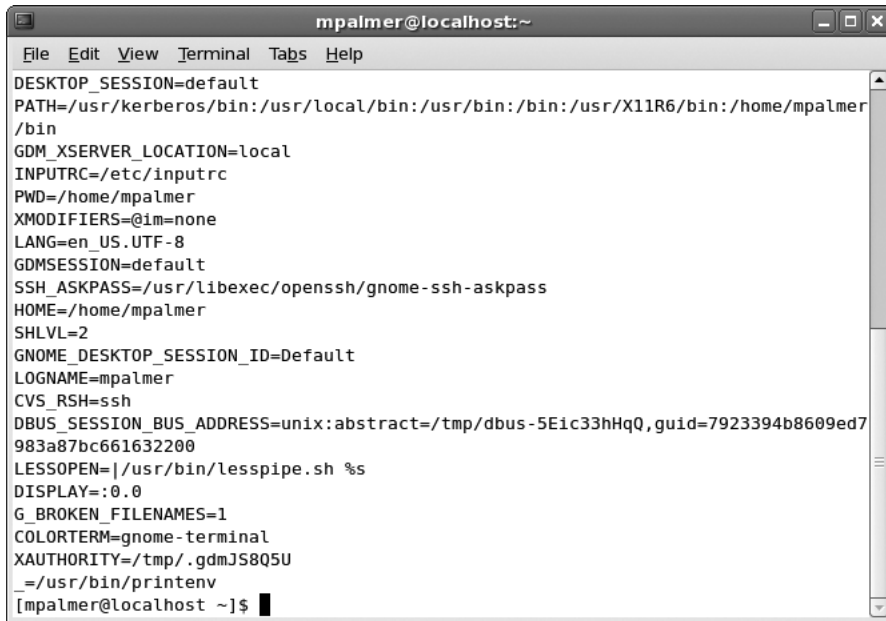
You can set up **environment variables** with initial values that you can change as needed. These variables, which UNIX/Linux read when you log in, determine many characteristics of your login session. For example, in Chapter 2 you learned about the `PS1` environment variable, which determines the way your prompt appears. In addition, UNIX/Linux use environment variables to determine such things as where it should look for programs, which shell to use, and the path of your home directory.

Shell variables (defined earlier) are those you create at the command line or in a shell script. They are very useful in shell scripts for temporarily storing information.

Environment and Configuration Variables

Environment and configuration variables bear standard names, such as `PS1`, `HOME`, `PATH`, `SHELL`, `USERNAME`, and `PWD`. (Configuration and environment variables are capitalized to distinguish them from user variables.) A script file in your home directory sets the initial values of environment variables. You can use these variables to set up and personalize your login sessions. For example, you can set your `PATH` variable to search for the location of shell scripts that other users have created, so you can more easily execute those scripts. Table 6-2 lists standard Bash shell environment and configuration variables.

You can, at any time, use the *printenv* command to view a list of your current environment and configuration variables, which you should typically do before you change any. (See Figure 6-2.) Hands-on Project 6-1 enables you to view your environment variables.



```
mpalmer@localhost:~
File Edit View Terminal Tabs Help
DESKTOP_SESSION=default
PATH=/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/mpalmer
/bin
GDM_XSERVER_LOCATION=local
INPUTRC=/etc/inputrc
PWD=/home/mpalmer
XMODIFIERS=@im=none
LANG=en_US.UTF-8
GDMSESSION=default
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HOME=/home/mpalmer
SHLVL=2
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=mpalmer
CVS_RSH=ssh
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-5Eic33hHqQ,guid=7923394b8609ed7
983a87bc661632200
LESSOPEN=|/usr/bin/lesspipe.sh %s
DISPLAY=:0.0
G_BROKEN_FILENAMES=1
COLORTERM=gnome-terminal
XAUTHORITY=/tmp/.gdmJS8Q5U
_=/usr/bin/printenv
[mpalmer@localhost ~]$
```

Figure 6-2 Viewing the environment variable listing

Syntax **printenv** [-options] [*variable name*]

Dissection

- Prints a listing of environment and configuration variables
 - Specifies one or more variables as arguments to view information only about those variables
-



TIP

Besides the *printenv* command, consider using the *set* command (discussed later in this chapter) with no arguments to view your current Bash shell environment, including environment variables, shell script variables, and shell functions. (You learn about shell functions in Chapter 7.) To learn more about the environment and configuration variables used on your system, type *man bash* at the command line. Scroll to the section, Shell Variables.

Table 6-2 Standard Bash shell environment and configuration variables

Name	Variable Contents	Determined by
HOME	Identifies the path name for user's home directory	System
LOGNAME	Holds the account name of the user currently logged in	System
PPID	Refers to the parent ID of the shell	System
TZ	Holds the time zone set for use by the system	System
IFS	Enables the user to specify a default delimiter for use in working with files	Redefinable
LINEND	Holds the current line number of a function or script	Redefinable
MAIL	Identifies the name of the mail file checked by the mail utility for received messages	Redefinable
MAILCHECK	Identifies the interval for checking and received mail (example: 60)	Redefinable
PATH	Holds the list of path names for directories searched for executable commands	Redefinable
PS1	Holds the primary shell prompt	Redefinable
PS2	Contains the secondary shell prompt	Redefinable
PS3 and PS4	Holds prompts used by the <i>set</i> and <i>select</i> commands	Redefinable
SHELL	Holds the path name of the program for the type of shell you are using	Redefinable
BASH	Contains the absolute path to the Bash shell, such as <code>/bin/bash</code>	User defined
BASH_VERSION	Holds the version number of Bash	User defined
CDPATH	Identifies the path names for directories searched by the <i>cd</i> command for subdirectories	User defined
ENV	Contains the file name containing commands to initialize the shell, as in <code>.bashrc</code> or <code>.tcshrc</code>	User defined
EUID	Holds the user identification number (UID) of the currently logged in user	User defined
EXINIT	Contains the initialization commands for the vi editor	User defined
FCEDIT	Enables you to access a range of commands in the command history file; FCEDIT is a Bash shell utility and is the variable used to specify which editor (vi by default) is used when you invoke the FC command	User defined
FIGORE	Specifies file name suffixes to ignore when working with certain files	User defined

Table 6-2 Standard Bash shell environment and configuration variables (continued)

Name	Variable Contents	Determined by
FUNCNAME	Contains the name of the function that is running, or is empty if there is no shell function running	User defined
GROUPS	Identifies the current user's group memberships	User defined
HISTCMD	Contains the sequence number that the currently active command is assigned in the history index of commands that already have been used	User defined
HISTFILE	Identifies the file in which the history of the previously executed commands is stored	User defined
HISTFILESIZE	Sets the upward limit of command lines that can be stored in the file specified by the HISTFILE variable	User defined
HISTSIZE	Establishes the upward limit of commands that the Bash shell can recall	User defined
HOSTFILE	Holds the name of the file that provides the Bash shell with information about its network host name (such as <i>localhost.localdomain</i>) and IP address (such as <i>129.0.0.24</i>); if the HOSTFILE variable is empty, the system uses the file <i>/etc/hosts</i> by default	User defined
HOSTTYPE	Contains information about the type of computer that is hosting the Bash shell, such as <i>i386</i> for an Intel-based processor	User defined
INPUTRC	Identifies the file name for the Readline start-up file overriding the default of <i>/etc/inputrc</i>	User defined
MACHTYPE	Identifies the type of system, including CPU, operating system, and desktop	User defined
MAILPATH	Contains a list of mail files to be checked by mail for received messages	User defined
MAILWARNING	Enables (when set) the user to determine if she has already read the mail currently in the mail file	User defined
OLDPWD	Identifies the directory accessed just before the current directory	User defined
OPTIND	Shows the index number of the argument to be processed next, when a command is run using one or more option arguments	User defined
OPTARG	Contains the last option specified when a command is run using one or more option arguments	User defined

Table 6-2 Standard Bash shell environment and configuration variables (continued)

Name	Variable Contents	Determined by
OPTERR	Enables Bash to display error messages associated with command-option arguments, if set to 1 (which is the default established each time the Bash shell is invoked)	User defined
OSTYPE	Identifies the type of operating system on which Bash is running, such as linux-gnu	User defined
PROMPT_COMMAND	Holds the command to be executed prior to displaying a primary prompt	User defined
PWD	Holds the name of the directory that is currently accessed	User defined
RANDOM	Yields a random integer each time it is called, but you must first assign a value to the RANDOM variable to properly initialize random number generation	User defined
REPLY	Specifies the line to read as input, when there is no input argument passed to the built-in shell command, which is read	User defined
SHLVL	Contains the number of times Bash is invoked plus one, such as the value 3 when there are two Bash (terminal) sessions currently running	User defined
TERM	Contains the name of the terminal type in use by the Bash shell	User defined
TIMEFORMAT	Contains the timing for pipelines	User defined
TMOUT	Enables Bash to stop or close due to inactivity at the command prompt, after waiting the number of seconds specified in the TMOUT variable (TMOUT is empty by default so that Bash does not automatically stop due to inactivity.)	User defined
UID	Holds the user identification number of the currently logged in user	User defined

Shell Variables

Shell variables are variables that you can define and manipulate for use with program commands that you employ in a shell. These are variables that are temporarily stored in memory and that you can display on the screen or use to perform specific actions in a shell script. For example, you might define the shell variable **TODAY** to store today's date so you can later recall it and then print it on a report generated from a shell script.

When you work with shell variables, keep in mind guidelines for handling them and for naming them. Some basic guidelines for handling shell variables are:

- Omit spaces when you assign a variable without using single or double quotation marks around its value, such as when assigning a numerical value—use `x=5` and not `x = 5`. (This type of assignment also enables you to perform mathematical operations on the assigned value.)
- To assign a variable that must contain spaces, such as a string variable, enclose the value in double or single quotation marks—use `fname="Thomas F. Berentino"` and not `fname=Thomas F. Berentino`. A **string** variable is a nonnumeric field of information treated simply as a group of characters. Numbers in a string are considered characters rather than digits.
- To reference a variable, use a dollar sign (\$) in front of it or enclose it in curly brackets ({ }).
- If the variable consists of an array (a set of values), use square brackets ([]) to refer to a specific position of a value in an array—use `myarray[0]=value1` for the first value in the array, for example.
- Export a shell variable to make the variable available to other shell scripts (as discussed in the following section).
- After you create a shell variable, you can configure it so that it cannot be changed by entering the *readonly* command with the variable name as the argument, such as *readonly fname*.

Sample guidelines for naming shell variables are:

- Avoid using the dollar sign in a variable name, because this can create confusion with using the dollar sign to reference the shell variable.
- Use names that are descriptive of the contents or purpose of the shell variable—use *lname* for a variable to contain a person's last name, instead of *var* or *x*, for example.
- Use capitalization appropriately and consistently—for instance, if you are defining address information, use variable names such as *city*, *state*, *zip* and not *City*, *STATE*, *ZIP*. Note that some programmers like to use all lowercase letters or all uppercase letters for variable names. For example, many script, C, and C++ programmers prefer using all lowercase letters when possible.
- If a variable name is to consist of two or more words, use underscores between the words—use *last_name* and not *last name*, for example.

In the next section, you learn about shell operators, which are used to define and evaluate variables, such as environment and shell variables.

SHELL OPERATORS

Bash shell operators are divided into four groups:

- Defining operators
- Evaluating operators
- Arithmetic and relational operators
- Redirection operators

You learn about each of these groups of operators in the following sections. You also learn how to use the *export* command to make a variable you have defined available to a shell script. Finally, you look at how to modify the *PATH* environment variable to make it easier to run shell scripts.

6

Defining Operators

Defining operators are used to assign a value to a variable. Evaluating operators are used for actions such as determining the contents of a variable. The equal sign (=) is one of the most common operators used to define a variable. For example, assume that you want to create a variable called *NAME* and assign *Becky* as the value to be contained in the variable. You would set the variable as follows:

```
NAME=Becky
```

The variable names or values that appear to the left and right of an operator are its **operands**. The name of the variable you are setting must appear to the left of the = operator. The value of the variable you are setting must appear to the right.



NOTE

Notice there are no spaces between the = operator and its operands.

Sometimes, it is necessary to assign to a variable's contents a string of characters that contain spaces, such as *Becky J. Zubrow*. To make this kind of assignment, you surround the variable contents with double quotation marks as follows:

```
NAME="Becky J. Zubrow"
```

Another way to assign a value to a shell variable is by using the back quote (`) operator. (The back quote is not the same as the apostrophe or single quotation mark; see the following Tip.) This operator is used to tell the shell to execute the command inside the back quotes and then store the result in the variable. For example, in the following:

```
LIST=`ls`
```

The *ls* command is executed and a listing of the current working directory is stored in the *LIST* variable.



On many standard keyboards, the key for the back quote operator is located in the upper-left corner under the Esc key and is combined with the tilde (~) on that key.

Evaluating Operators

When you assign a value to a variable, you might want to evaluate it by displaying its contents via an **evaluating operator**. You can use the dollar sign (\$) in front of the variable along with the *echo* command to view the contents. For example, if you enter:

```
echo $NAME
```

you see the contents of the NAME variable you created earlier. You can also use the format `echo "$NAME"` to view the variable's contents. However, if you enter:

```
echo '$NAME'
```

using single quotation marks, the contents of NAME are suppressed and all you see is \$NAME echoed on the screen.

Try Hands-on Project 6-2 to use the defining and evaluation operators.

Arithmetic and Relational Operators

Arithmetic operators consist of the familiar plus (+) for addition, minus (-) for subtraction, asterisk (*) for multiplication, and slash (/) for division. **Relational operators** compare the relationship between two values or arguments, such as greater than (>), less than (<), equal to (=), and others. Table 6-3 explains the arithmetic and relational operators. For a complete listing of arithmetic operators enter *man bash* and go to the section, ARITHMETIC EVALUATION.

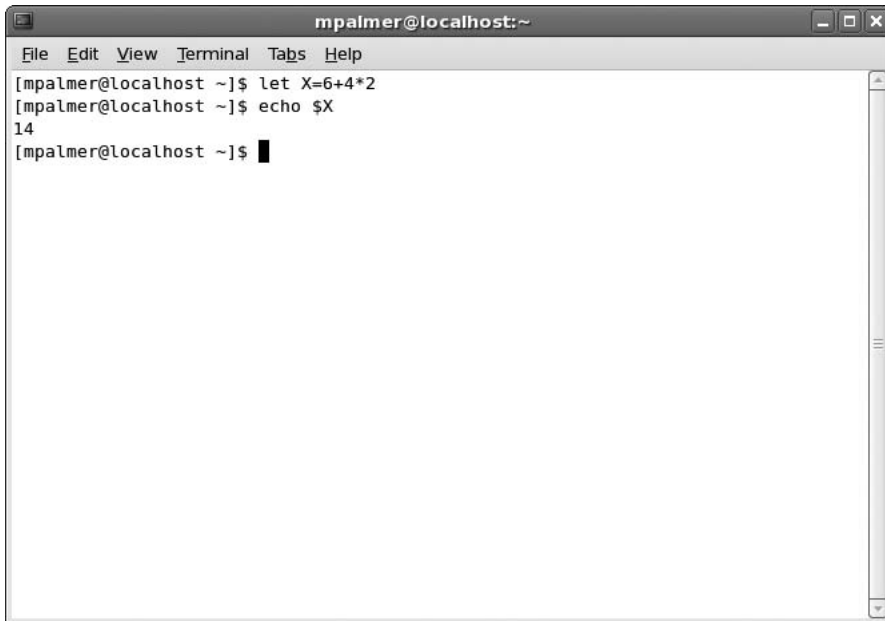
Table 6-3 Examples of the shell's arithmetic and relational operators

Operator	Description	Example
-, +	Unary minus and plus	+R (denotes positive R) -R (denotes negative R)
!, ~	Logical and bitwise negation	!Y (returns 0 if Y is nonzero, returns 1 if Y is zero) ~X (reverses the bits in X)
*, /, %	Multiplication, division, and remainder	A * B (returns A times B) A / B (returns A divided by B) A % B (returns the remainder of A divided by B)
+, -	Addition, subtraction	X + Y (returns X plus Y) X - Y (returns X minus Y)
>, <	Greater than and less than	M > N (Is M greater than N?) M < N (Is M less than N?)
=, !=	Equality and inequality	Q = R (Is Q equal to R?) Q != R (Is Q not equal to R?)

When using arithmetic operators, the usual mathematical precedence rules apply: Multiplication and division are performed before addition and subtraction. For example, the value of the expression $6 + 4 * 2$ is 14, not 20. Precedence can be overridden, however, by using parentheses. For example, the value of the expression $(6 + 4) * 2$ is 20, not 14. Other mathematical rules also apply; for example, division by zero is treated as an error.

To store arithmetic values in a variable, use the *let* statement. For example, the following command stores 14 in the variable X (See Figure 6-3 using the *echo* command to show the contents of X after using the *let* command.):

```
let X=6+4*2
```

A screenshot of a terminal window titled 'mpalmer@localhost:~'. The window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The terminal shows the following commands and output:

```
[mpalmer@localhost ~]$ let X=6+4*2
[mpalmer@localhost ~]$ echo $X
14
[mpalmer@localhost ~]$
```

Figure 6-3 Using *let* to set the contents of a shell variable

Notice in the preceding example that there is one space between *let* and the expression that follows it. Also, there are no spaces in the arithmetic equation following a *let* statement. In this example, there are no spaces on either side of the equal (=), plus (+), and multiplication (*) operators.

You can use shell variables as operands to arithmetic operators. Assuming the variable X has the value 14, the following command stores 18 in the variable Y:

```
let Y=X+4
```

Syntax `let` *expression with operators*

Dissection

- Performs a given action on numbers that is specified by operators and stores the result in a shell variable
 - Parentheses are used around specific expressions if you want to alter the mathematical precedence rules or to simply ensure the result is what you intend.
-

`let` is a built-in command for the Bash shell. For documentation about `let`, enter `man bash`, and scroll down to the section SHELL BUILTIN COMMANDS. Try Hands-on Project 6-3 to learn how to use the `let` command.

REDIRECTION OPERATORS

Recall that the `>` **redirection operator** overwrites an existing file. For example, in `cat file1 > file2`, the contents of `file1` overwrite the contents of `file2`. If you write a shell script that uses the `>` operator to create a file, you might want to prevent it from overwriting important information. You can use the `set` command with the `-o noclobber` option to prevent a file from being overwritten, as in the following example:

```
$ set -o noclobber <Enter>
```

Syntax `set` `[-options]` `[arguments]`

Dissection

- With no options, displays the current listing of Bash environment and shell script variables
 - Useful options include:
 - a exports all variables after they are defined
 - n takes commands without executing them, so you can debug errors without affecting data (Also see the `sh -n` command later in this chapter.)
 - o sets a particular shell mode—when used with `noclobber` as the argument, it prevents files from being overwritten by use of the `>` operator
 - u shows an error when there is an attempt to use an undefined variable
 - v displays command lines as they are executed
-

`set` is another built-in command for the Bash shell. For documentation about `set`, enter `man bash`, and scroll down to the section SHELL BUILTIN COMMANDS.



If you want to save time and automatically export all shell script variables you have defined, use `set` with the `-a` option.

However, you can choose to overwrite a file anyway by placing a pipe character (`|`) after the redirection operator:

```
$ set -o noclobber      <Enter>
$ cat new_file > old_file <Enter>
bash: old_file: cannot overwrite existing file
$ cat new_file >| old_file <Enter>
```



Avoid employing the `-o noclobber` option if you are using the Bash shell in the X Window interface with the KDE desktop. On some distributions, using the option in this manner can unexpectedly terminate the command-line session.

Exporting Shell Variables to the Environment

Shell scripts cannot automatically access variables created and assigned on the command line or by other shell scripts. To make a variable available to a shell script, you must use the `export` command to give it a global meaning so that it is viewed by the shell as an environment variable.

Syntax `export` [-options] [*variable names*]

Dissection

- Makes a shell variable global so that it can be accessed by other shell scripts or programs, such as shell scripts or programs called within a shell script
 - Useful options include:
 - `-n` undoes the export, so the variable is no longer global
 - `-p` lists exported variables
-

`export` is a built-in Bash shell command, which means you can find help documentation by entering `man bash` and scrolling to the SHELL BUILTIN COMMANDS section. Try Hands-on Project 6-4 to use the `export` command in the Bash shell.

Modifying the PATH Variable

Just as shell variables are not universally recognized until you export them, the same is true for executing a shell script. Up to this point, you have used `./` to run a shell script. This is because the shell looks for programs in the directories specified by the `PATH` variable. If you

are developing a shell script in a directory that is not specified in your `PATH` environment variable, you must type `./` in front of the shell name. If you just type the name of the shell by itself, the script doesn't run because it is not in your currently defined path—which means that the shell interpreter cannot find it to run. You need to type `./` to tell the shell interpreter to look in your current working directory.

For example, in some UNIX/Linux systems, such as Fedora, Red Hat Enterprise Linux, and SUSE, your home directory is not automatically defined in your current path. You can verify this by typing the following to see what directories are in your path:

```
echo $PATH
```

In Fedora or Red Hat Enterprise Linux, for example, you will likely see a path such as the following:

```
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/username/bin
```

Notice that each directory in the path is separated by a colon. Your home directory, as represented by `/home/username` is not in the path by default, but another directory, `/home/username/bin`, in which programs might be stored, is in the path by default—although the actual directory, `/home/username/bin`, might not be created by default on your system even though it is in the path.

Because new shell scripts are most often kept in the current directory while they are being tested, you should add the current working directory to the `PATH` variable. Here is an example command for how you can do this quickly:

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

Remember, the shell interprets `$PATH` as the contents of the `PATH` variable. This sample command sets the `PATH` variable to its current contents. The colon and dot (.) add the current directory to the search path so that the shell program can locate the new program.

After you type this command, you can execute new shell scripts by simply using the name of the script without prefacing it with the `./` characters. Hands-on Project 6-5 enables you to try this.



TIP

When you type `PATH=$PATH:.`, the current working directory is temporarily stored as part of your path only for the duration of your login session.



Configuring to have the current directory set in your path does involve some risk if a hacker gains access to your account while you are logged in. For example, a hacker might gain access through an open port (communication path in a network protocol). If you choose to put your current working directory in the PATH variable, be certain you have secured access to your account, such as through closing unused ports. For more information about operating system security, including for UNIX/Linux systems, see *Guide to Operating Systems Security* (Course Technology, ISBN 0-619-16040-3).

MORE ABOUT WILDCARD CHARACTERS

Shell scripts frequently use the asterisk (*) and other wildcard characters (such as ? and []), which help to locate information containing only a portion of a matching pattern. For example, to list all program files with names that contain a .c extension, use the following command:

```
ls *.c
```

Wildcard characters are also known as **glob** characters. If an unquoted argument contains one or more glob characters, the shell processes the argument for file name generation. Glob characters are part of **glob patterns**, which are intended to match file names and words. Special constructions that might appear in glob patterns are:

- The question mark (?) matches exactly one character, except for the backslash and period.
- The asterisk (*) matches zero or more characters in a file name.
- *[chars]* defines a class of characters. The glob pattern matches any single character in the class. A class can contain a range of characters, as in *[a-z]*.

For example, assume the working directory contains files chap1, chap2, and chap3. The following command displays the contents of all three files:

```
more chap[1-3] <Enter>
```

The commands and variables used in shell scripts are organized into different logic structures. In the next sections, you learn how to use logic structures for effective script handling.

SHELL LOGIC STRUCTURES

Logic structures are techniques for structuring program code and affect the order in which the code is executed or how it is executed, such as looping back through the code from a particular point or jumping from one point in the code to another. The four basic logic structures needed for program development are:

- Sequential logic
- Decision logic

- Looping logic
- Case logic

Each of these logic structures is discussed in the following sections.

Sequential Logic

Sequential logic works so that commands are executed in the order in which they appear in the script or program.

For example, consider a sales manager in a company who begins each week by tallying sales information. First, she runs the `tally_all` program to compute the year's gross sales statistics up to the current date. Then she runs the `profit_totals` program to tally the profit statistics. Next, she runs the `sales_breakdown` report program that shows the sales performance of the 40 salespeople she manages. Finally, she runs the `management_statistics` report program to provide the sales statistics needed by her management. The sales manager can automate her work by creating a script that uses sequential logic, first running the `tally_all` program, then the `profit_totals`, the `sales_breakdown`, and the `management_statistics` programs. Her shell script with sequential logic would have the following sequence of commands (note that her system is set up so that all she needs to do is to enter the programs names at the commandline):

```
tally_all
profit_totals
sales_breakdown
management_statistics
```

The only break in sequence logic comes when a branch instruction changes the flow of execution by redirecting to another location in the script or program. A **branch instruction** is one that tells the program to go to a different section of code.

Many scripts are simple, straightforward command sequences. An example is the Programmer Activity Status Report script you wrote in Chapter 5, and is listed next. The shell executes the script's commands in the order they appear in the file. You use sequential logic to write this type of application.

```
#=====
# Script Name:      practivity
# By:              MP
# Date:            November 2009
# Purpose: Generate Programmer Activity Status Report
#=====
cut -d: -f4 project | sort | uniq -c | awk '{printf "%s:
%s \n", $2, $1}' > pnum
cut -d: -f1-4 programmer | sort -t: +0 -1 | uniq > pnn
join -t: -a1 -j1 1 -j2 1 pnn pnum > pactrep
```

```
# Print the report
awk '
BEGIN {
    { FS = ":" }
    { print "\tProgrammer Activity Status Report\n" }
    { "date" | getline d }
    { printf "\t    %s\n",d }
    { print "Prog# \t*--Name--*                Projects\n" }
    { print "=====\n" }
    { printf "%-s\t%-12.12s %-12.12s %s\t%d\n",
        $1, $2, $3, $4, $5 } ' pactrep
# remove all the temporary files
rm pnum pnn pactrep
```

Hands-on Project 6-6 enables you to build a short script to demonstrate sequential logic as well as practice the *let* command, and to build expressions using constants, variables, and arithmetic operators.

Decision Logic

Decision logic enables your script or program to execute a statement or series of statements only if a certain condition exists. In this usage, a **statement** is another name for a line of code that performs an action in your program. The *if* statement is a primary decision-making logic structure in this type of logic.

In decision logic, the script is programmed to make decisions as it runs. By using the *if* statement, you can specify conditions for the script to evaluate before it makes a decision. For example, if *a* occurs, then the script does *b*; but if *x* occurs instead, then the script does *y*. Consider a situation in which a magazine publisher gives the reader two subscription choices based on price. If the reader sends in \$25, then the magazine publisher gives him 12 weeks of the publication, but if the reader sends in \$50 dollars, then the magazine publisher gives him 24 weeks of the publication.

Another way to use a decision structure is as a simple yes or no situation. For example, you might use a portion of a script to enable the user to continue updating a file or to close the file. When the script asks: “Do you want to continue updating (*y* or *n*)?”, if you answer *y* then the script gives you a blank screen form in which to enter more information to put in the file. If instead you answer *n*, then the script closes the file and stops.

Consider, for example, the following lines of code. In this shell script, the user is asked to enter a favorite vegetable. If the user enters “broccoli,” the decision logic of the program displays “Broccoli is a healthy choice.” If any other vegetable is entered, the decision logic displays the line “Don’t forget to eat your broccoli also.”

```
echo -n "What is your favorite vegetable? "
read veg_name
if [ "$veg_name" = "broccoli" ]
then
```

```
    echo "Broccoli is a healthy choice."
else
    echo "Don't forget to eat your broccoli also."
fi
```

**NOTE**

Throughout the sample scripts, variables are always enclosed in double quotation marks, as in "\$veg_name", "\$choice", "\$looptest", "\$yesno", "\$guess", and "\$myfavorite", because of how the shell interprets variables. All shell variables, unless declared otherwise, are strings, which are arrays of alphanumeric characters. If you do not enter data in the string variables, the variables are treated as blank strings, which result in an invalid test. The enclosing double quotation marks, therefore, maintain the validity of strings, with or without data, and the test is carried out without producing an error condition.

You create and run this script in Hands-on Project 6-7. However, before you attempt the project, let's examine the contents of the script. The first statement uses the *echo* command to display a message on the screen. The *-n* option suppresses the line feed that normally appears after the message. The second statement uses the *read* command, which waits for the user to type a line of keyboard input. The input is stored in the variable specified as the *read* command's argument. The line in the script reads the user's input into the *veg_name* variable.

The next line begins an *if* statement. The word "if" is followed by an expression inside a set of brackets ([]). (The spaces on either side that separate the [and] characters from the enclosed expression are necessary.) The expression, which is tested to determine if it is true or false, compares the contents of the *veg_name* variable with the string *broccoli*. (When you use the = operator in an *if* statement's test expression, it tests its two operands for equality. In this case, the operands are the variable *\$veg_name* and the string *broccoli*. If the operands are equal, the expression is true—otherwise, it is false. If the contents of the *\$veg_name* variable are equal to *broccoli*, the statement that follows the word "then" is executed. In this script, it is the *echo* statement, "Broccoli is a healthy choice."

If the *if* statement's expression is false (if the contents of the *\$veg_name* variable do not equal *broccoli*), the statement that follows the word "else" is executed. That statement reads, "Don't forget to eat your broccoli also." In this script, it is a different *echo* statement.

Notice the last statement, which consists of the characters "fi." *fi* ("if" spelled backward) always marks the end of an *if* or an *if...else* statement.

**NOTE**

When you evaluate the contents of a variable using a logic structure, such as the *if* statement, you need to define the variable first, such as through a *read* statement.

You can nest a control structure, such as an *if* statement, inside another control structure. To **nest** means that you layer statements at two or more levels under an original statement

structure. For example, a script can have an *if* statement inside another *if* statement. The first *if* statement controls when the second *if* statement is executed, as in the following code sample:

```
echo -n "What is your favorite vegetable? "
read veg_name
if [ "$veg_name" = "broccoli" ]
then
    echo "Broccoli is a healthy choice."
else
    if [ "$veg_name" = "carrots" ]
    then
        echo "Carrots are great for you."
    else
        echo "Don't forget to eat your broccoli also."
    fi
fi
```

As you can see, the second *if* statement is located in the first *if* statement's *else* section. It is only executed when the first *if* statement's expression is false.

Decision logic structures, such as the *if* statement, are used in applications in which different courses of action are required, depending on the result of a command or comparison.

Looping Logic

In **looping logic**, a control structure (or loop) repeats until a specific condition exists or some action occurs. The basic idea of looping logic is to keep repeating an action until some condition is met. For example, the logic might keep printing a list of people's names until it reaches the final name on the list, prints the final name, and stops. In another example, a script might open an inventory file of mountain bike models, print the model and quantity for the first bike, do the same for the second bike, and so on until there are no more bikes listed in the file.

You learn two looping mechanisms in the sections that follow: the *for* and the *while* loop.

The For Loop

Use the *for* command to loop through a range of values. It causes a variable to take on each value in a specified set, one at a time, and perform some action while the variable contains each individual value. The loop stops after the variable has taken on the last value in the set and has performed the specified action with that value.

An example of a *for* loop is as follows:

```
for USERS in john ellen tom becky eli jill
do
    echo $USERS
done
```

In this *for* loop structure, the first line specifies the values that will be assigned, one at a time, to the `USERS` shell variable. Because six values are in the set, the loop repeats six times. Each

time it repeats, `USERS` contains a different value from the set, and the statement between the *do* and *done* statements is executed. The first time around the loop, “john” is assigned to the `USERS` variable and is then displayed on the screen via the *echo \$USERS* command. Next, “ellen” is assigned to the `USERS` variable and displayed. The loop continues through “tom,” “becky,” “eli,” and “jill.” After “jill” is assigned to `USERS` and displayed on the screen, the looping comes to an end and the *done* command is executed to end the looping logic.

Hands-on Project 6-8 enables you to use a *for* loop in a shell script.

Executing Control Structures at the Command Line

Most shell script control structures, such as the *if* and *for* statements, must be written across several lines. This does not prevent you from executing them directly on the command line, however. For example, you can enter the *for* statement at the command prompt, enter the variable name and elements to use for the variable, and press Enter. You go into a command-line processor to execute the remaining statements in the loop, pressing Enter after each statement. The shell knows more code comes after you type the first line. It displays the `>` prompt, indicating it is ready for the control structure’s continuation. The shell reads further input lines until you type the word “done,” which marks the end of the *for* loop.

In the following lines, each of the elements tennis, swimming, movies, and travel are displayed one line at a time until the loop ends after displaying travel. (See Figure 6-4.)

```
$ for myhobbies in tennis swimming movies travel <Enter>
> do <Enter>
> echo $myhobbies <Enter>
> done <Enter>
```

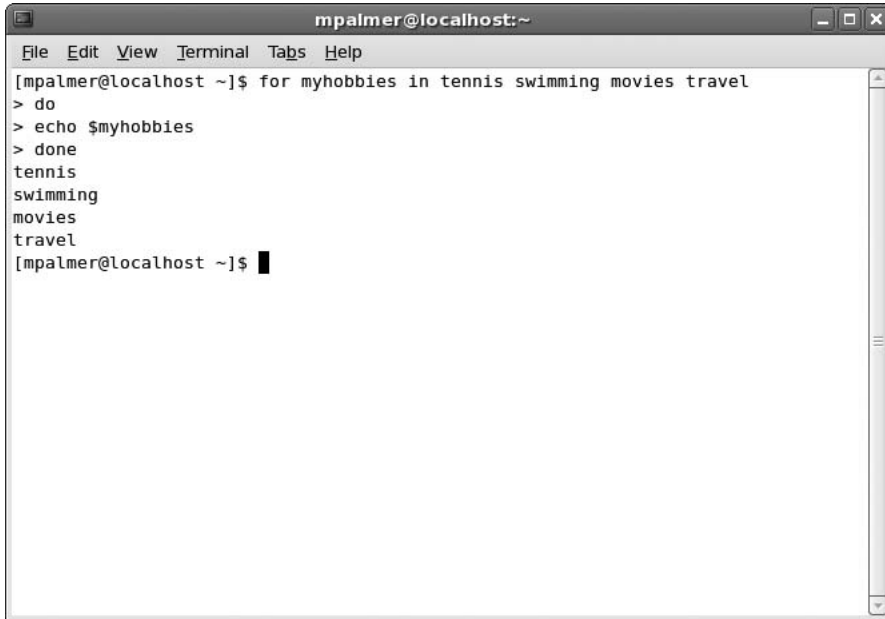
Hands-on Project 6-8 enables you to compare using the command line to using a shell script for executing a simple *for* loop.

Using Wildcard Characters in a Loop

The `[]` wildcard characters can be very useful in looping logic. For example, consider that you have four files that all start with the same four characters: `chap1`, `chap2`, `chap3`, and `chap4`. You can use the wildcard notation `chap[1234]` to output the contents of all four file names to the screen using the following statement:

```
for file in chap[1234]; do
    more $file
done
```

Notice that in the first line, two commands are combined by using the semicolon (`;`) character to run each on one line. Try Hands-on Project 6-9 to use brackets as wildcards.

A terminal window titled 'mpalmer@localhost:~' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows a shell script using a 'for' loop to iterate over the words 'tennis', 'swimming', 'movies', and 'travel'. Inside the loop, 'do' is followed by 'echo \$myhobbies' and 'done'. The output shows each hobby on a new line. The prompt returns to '[mpalmer@localhost ~]\$' after the loop completes.

```
mpalmer@localhost:~$ for myhobbies in tennis swimming movies travel
> do
> echo $myhobbies
> done
tennis
swimming
movies
travel
[mpalmer@localhost ~]$
```

Figure 6-4 Using the *for* loop from the command line

The While Loop

A second approach to looping logic is the *while* statement. The *while* statement continues to loop and execute commands or statements as long as a given condition or set of conditions is true. As soon as the condition or conditions are false, the loop is exited at the *done* statement. The following is an example of a simple shell script that uses a *while* statement:

```
echo -n "Try to guess my favorite color: "
read guess
while [ "$guess" != "red" ]; do
    echo "No, not that one. Try again. "; read guess
done
```

In this example, the first line asks the user to “Try to guess my favorite color:” and the user’s response is read into the variable *guess*.

The *while* loop tests an expression in a manner similar to the *if* statement. As long as the statement inside the brackets is true, the statements inside the *do* and *done* statements repeat. In this example, the expression inside the brackets is “*\$guess*” != “red”, which tests to see if the *guess* variable is not equal to the string, “red”. Note that != is the not-equal (inequality) operator. (Refer to Table 6-3.) The *while* statement tests the two operands on either side of the != operator and returns true if they are not equal. Otherwise, it returns false. In this example, the *echo* and *read* statements inside the loop repeat until the user enters red, which makes the expression “*\$guess*” != “red” false.

Hands-on Project 6-10 gives you the opportunity to program the sample code described here and then to program a more complex example such as might be used to input name and address information in a file.



Use looping logic in the form of *for* and *while* statements in applications in which code must be repeated a determined or undetermined number of times.

TIP

Case Logic

The **case logic** structure simplifies the selection of a match when you have a list of choices. It allows your script to perform one of many actions, depending on the value of a variable.

Consider a fund raiser in which contributors receive different premiums. If someone gives \$20 they get a free pen. If they give \$30 they get a T-shirt and if they give \$50 they get a free mug. People who give \$100 get a free CD. In case logic, you can take the four types of contributions and associate each one with a different action. For example, when the user types in 30 in answer to the question, “How much did you contribute?”, then the screen displays “Your free gift is a T-shirt.” If the user types in 100, then the screen displays, “Your free gift is a CD.” The advantage of case logic in this example is that it simplifies your work by using fewer lines—you use one *case* statement instead of several *if* statements, for example.

One common application of the case logic structure is in creating menu selections on a computer screen. A menu is a screen display that offers several choices. Consider, for example, a menu used in a human resources application in which there is a menu option to enter information for a new employee, another menu option to view employee name and address information, another menu option to view salary information, and so on. Each menu option branches to a different program. For example, if you select the option to enter information for a new employee, this starts the employee data entry program. Or, if you select to view employee salary information, a salary report program runs.

The following is a basic example of how the *case* statement works in case logic:

```
echo -n "Enter your favorite color: "; read color
case "$color" in
    "blue")  echo "As in My Blue Heaven.>";;
    "yellow") echo "As in the Yellow Sunset.>";;
    "red")   echo "As in Red Rover, Red Rover.>";;
    "orange") echo "As in Autumn has shades of Orange.>";;
    *)      echo "Sorry, I do not know that color.>";;
esac
```

In this sample script, the case structure examines the contents of the color variable, and searches for a match among the values listed. When a match is found, the statement that immediately follows the *case* value is executed. For example, if the color variable contains orange, the *echo* statement that appears after “orange”) is executed: “As in Autumn has shades

of Orange.” If the contents of the color variable do not match any of the values listed, the statement that appears after `*)` is executed: “Sorry, I do not know that color.”

Note the use of two semicolons (`;;`) that terminate the action(s) taken after the case structure matches what is being tested. Also notice that the case structure is terminated by the word “`esac`,” which is “case” spelled backward.

As you can see, case logic is designed to pick one course of action from a list of many, depending on the contents of a variable. This is why case logic is ideal for menus, in which the user chooses one of several values. Try Hands-on Project 6-11 to program using case logic.

USING SHELL SCRIPTING TO CREATE A MENU

When you create an application that consists of several shell scripts, it is often useful to create a menu with options that branch to specific scripts. You create menus in the Hands-on Projects section of this chapter. In preparation for creating a menu, you need to have one more command under your belt, the `tput` command. This is one of the less-publicized UNIX/Linux commands, but is important to know for developing an appealing and user-friendly menu presentation.

Syntax `tput` [-options] *arguments*

Dissection

- Can be used to initialize the terminal or terminal window display, position text, and position the cursor
 - Useful options include:
 - `bold='tput smso' offbold='tput rmso'` enables/disables boldfaced type
 - `clear` clears the screen
 - `cols` prints the number of columns
 - `cup` positions the cursor and text on the screen
-

The `tput` command enables you to initialize the terminal display or terminal window, to place text and prompts in desired locations, and to respond to what the user selects from the menu. Some examples of what you can do with the `tput` command are:

- `tput cup 0 0` moves the cursor to row 0, column 0, which is the upper-left corner of the screen.
- `tput clear` clears the screen.
- `tput cols` prints the number of columns for the current terminal display.

- `bold=`tput smso` offbold=`tput rmso`` sets boldfaced type by setting the bold shell variable for stand-out mode sequence and by setting the offbold shell variable to turn off stand-out mode sequence.

Hands-on Project 6-12 enables you to gain experience using the `tput` command. Hands-on Project 6-15 uses the `tput` command so you can begin building a menu to use with an actual application.

DEBUGGING A SHELL SCRIPT

As you have probably discovered by this point, sometimes a shell script does not execute because there is an error in one or more commands within the script. For example, perhaps you entered a colon instead of a semicolon or left out a bracket. Another common problem is leaving out a space or not putting a space in the correct location.

Now that you have some experience developing shell scripts and have possibly encountered some problems, you can appreciate why it is important to have a tool to help you troubleshoot errors. The `sh` command that is used to run a shell script includes several options for debugging.

Syntax `sh` [-options] [*shell script*]

Dissection

- In UNIX and Linux, it calls the command interpreter for shell scripts; and in Linux, it uses the Bash shell with the command interpreter
 - Useful options include:
 - n checks the syntax of a shell script, but does not execute command lines
 - v displays the lines of code while executing a shell script
 - x displays the command and arguments as a shell script is run
-

Two of the most commonly used `sh` options are `-v` and `-x`. The `-v` option displays the lines of code in the script as they are read by the interpreter. The `-x` option shows somewhat different information by displaying the command and accompanying arguments line by line as they are run.

By using the `sh` command with these options, you can view the shell script line by line as it is running and determine the location and nature of an error on a line when a script fails.

Try Hands-on Project 6-13 to compare `sh -v` and `sh -x` to debug a shell script.

Further, sometimes you want to test a script that updates a file, but you want to give the script a dry run without actually updating the file—particularly so that data in the file is not altered if the script fails at some point. Use the `-n` option for this purpose because it reads and

checks the syntax of commands in a script, but does not execute them. For example, if you are testing a script that is designed to add new information to a file, when you run it with the `sh -n` command, the script does not actually process the information or add it to the file. If a syntax error exists, you see an error message so you know that you must fix the script before using it on live data.

Now that you have an idea of how to create a menu script, it is helpful to learn some additional shell features and commands before creating your application. You first learn more about how to customize your personal environment and how to use the `trap` command to clean up unnecessary files in your environment.

CUSTOMIZING YOUR PERSONAL ENVIRONMENT

When your work requirements center on computer programming and shell scripting, consider customizing your environment by modifying the initial settings in the login scripts. A **login script** is a script that runs just after you log in to your account. For example, many programmers set up a personal bin directory in which they can store and test their new programs without interfering with ongoing operations. The traditional UNIX/Linux name for directories that hold executable files is `bin`.

A useful tool for customizing the command environment is the alias. An **alias** is a name that represents another command. You can use aliases to simplify and automate commands you use frequently. For example, the following command sets up an alias for the `rm` command.

```
alias rm="rm -i"
```

This command causes the `rm -i` command to execute any time the user enters the `rm` command. This is a commonly used alias because it ensures that users are always prompted before the `rm` command deletes a file. The following are two other common aliases that help safeguard files:

```
alias mv="mv -i"
alias cp="cp -i"
```

Syntax `alias` [-options] [*name* = "*command*"]

Dissection

- Creates an alternate name for a command
 - Useful options include:
 - p prints a list of all aliases
-

`alias` is a built-in Bash shell command. You can learn more about `alias` by entering `man bash` and find `alias` under the SHELL BUILTIN COMMANDS section. Hands-on Project 6-14 enables you to set aliases.

The **.bashrc** file that resides in your home directory as a hidden file (enter `ls -a` to view hidden files) can be used to establish customizations that take effect for each login session. The `.bashrc` script is executed each time you generate a shell, such as when you run a shell script. Any time a subshell is created, `.bashrc` is reexecuted. The following `.bashrc` file is commented to explain how you can make your own changes.

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc # if any global definitions are defined
                  # run them first
alias rm='rm -i'  # make sure user is prompted before
                  # removing files
alias mv='mv -i'  # make sure user is prompted before
                  # overlaying files
set -o ignoreeof  # Do not allow Ctrl-d to log out
set -o noclobber  # Force user to enter >| to write
                  # over existing files
PS1="\w \$"       # Set prompt to show working directory
```

In addition to knowing how to customize your work environment, you should also be familiar with the `trap` command to clean your storage of temporary files.

THE TRAP COMMAND

`trap` is a command in the Bash shell that is used to execute another command when a specific signal is received. For example, you might use `trap` to start a new program after it detects through an operating system signal that a different program has terminated. Another example is using `trap` to end a program when `trap` receives a specific signal, such as trapping when the user types Ctrl-c and gracefully ending the currently running program.

The `trap` command is useful when you want your shell program to automatically remove any temporary files that are created when the shell script runs. The `trap` command specifies that a command, listed as the argument to `trap`, is read and executed when the shell receives a specified system signal.

Syntax `trap` [*command*] [*signal number*]

Dissection

- When a signal is received from the operating system, the argument included with `trap` is executed.
- Common signals used with `trap` include:
 - 0 The completion of a shell script has occurred
 - 1 A hang up or logout signal has been issued
 - 2 An interrupt has been received, such as Ctrl+c

- 3 A quit signal has been issued
 - 4 An illegal instruction has been received
 - 9 A termination signal has been issued
 - 15 A program has been ended, such as through a *kill* command
 - 19 A process has been stopped
 - 20 A process has been suspended
- Useful options include:
 - l displays a listing of signal numbers and their associated signal designations

Here is an example of a use for the *trap* command:

```
trap "rm ~/tmp/* 2> /dev/null; exit" 0
```

This command has two arguments: a command to be executed and a signal number from the operating system. The command *rm ~/tmp/* 2> /dev/null; exit* deletes everything in the user's *tmp* directory, redirects the error output of the *rm* command to the null device (so it does not appear on the screen), and issues an *exit* command to terminate the shell. The signal specified is 0, which is the operating system signal generated when a shell script is exited. So, if this sample command is part of a script file, it causes the specified *rm* command to execute when signal 0 is sent by the operating system.

The programmer often sets up *~/tmp* (a subdirectory of the user's home directory) to store temporary files. When the script file exits, any files placed in *~/tmp* can be removed. This is called “good housekeeping” on the part of the programmer.

The *trap* command is another example of a built-in Bash shell command. You can learn more about *trap* by entering *man bash* and finding the *trap* command listed in the SHELL BUILTIN COMMANDS section.

PUTTING IT ALL TOGETHER IN AN APPLICATION

In this chapter, you learned all of the pieces necessary to create a multifunctional application. You learned how to:

- Assign and manage variables
- Use shell operators
- Employ shell logic structures
- Use additional wildcard characters
- Use *tput* for managing screen initialization and screen text placement
- Use the *trap* command to clean up temporary files used by an application

In Hands-on Projects 6-15 through 6-20, you use the skills and knowledge you have acquired in this and previous chapters to build a multipurpose application. The application

you build simulates one that an organization might use to track telephone numbers and other information about its employees. This application enables the user to input new telephone number and employee information, to print a list of telephone numbers, and to search for a specific telephone number. You build the application entirely from shell scripts. Also, to make this undertaking manageable (and be consistent with programming practices), you build the application through creating small pieces that you prototype, test, and later link together into one menu-based application.

CHAPTER SUMMARY

- ❑ A high-level language (such as C, C++, or COBOL) is a language that uses English-like expressions. A high-level language must be converted into a low-level (machine) language before the computer can execute it. Programmers use a compiler to convert the high-level language to machine language.
- ❑ An interpreter reads commands or a programming language and interprets each line into an action. A shell, such as the Bash shell, interprets UNIX/Linux shell scripts. Shell scripts do not need to be converted to machine language because the UNIX/Linux shell interprets the lines in shell scripts.
- ❑ UNIX/Linux shell scripts, created with the vi or Emacs editor, contain instructions that do not need to be written from scratch, but can be selectively chosen from the operating system's inventory of executable commands.
- ❑ Linux shells are derived from the UNIX Bourne, Korn, and C shells. The three typical Linux shells are Bash, csh/tcsh, and ksh/zsh; Bash is the most commonly used Linux shell.
- ❑ UNIX/Linux employ three types of variables: configuration, environment, and shell. Configuration variables contain setup information for the operating system. Environment variables hold information about your login session. Shell variables are created in a shell script or at the command line. The *export* command is used to make a shell variable an environment variable.
- ❑ The shell supports many operators, including ones that perform arithmetic operations.
- ❑ You can use wildcard characters in shell scripts, including the bracket ([]) characters. Brackets surround a set of values that can match an individual character in a name or string.
- ❑ The logic structures supported by the shell are sequential, decision, looping, and case.
- ❑ You can use the *tput* command to manage cursor placement.
- ❑ You can customize the .bashrc file that resides in your home directory to suit particular needs, such as setting alias and default shell conditions.
- ❑ You can create aliases and enter them into .bashrc to simplify commonly used commands, such as *ls -l* and *rm -i*.
- ❑ Use the *trap* command inside a script file to remove temporary files after the script file has been run (exited).

COMMAND SUMMARY: REVIEW OF CHAPTER 6 COMMANDS

Command	Purpose	Options Covered in This Chapter
alias	Establishes an alias	-p prints all aliases.
case. . .in. . .esac	Allows one action from a set of possible actions to be performed, depending on the value of a variable	
export	Makes a shell variable an environment variable	-n can be used to undo the export. -p lists the exported variables.
for: do. . .done	Causes a variable to take on each value in a set of values; an action is performed for each value	
if. . .then. . .else. . .fi	Causes one of two actions to be performed, depending on the condition	
let	Stores arithmetic values in a variable	
printenv	Prints a list of environment variables	
set	Displays currently set shell variables; when options are used, sets the shell environment	-a exports all shell variables after they are assigned. -n takes commands without executing them, so you can debug errors. -o sets a particular shell mode—when used with noclobber as the argument, it prevents files from being overwritten by use of the > operator. -u yields an error message when there is an attempt to use an undefined variable. -v displays command lines as they are executed.
sh	Calls the command interpreter for shell scripts	-n checks the syntax of a shell script, but does not execute command lines. -v displays the lines of code while executing a shell script. -x displays the command and arguments as a shell script is run.
tput cup	Moves the screen cursor to a specified row and column	
tput clear	Clears the screen	
tput cols	Prints the number of columns on the current terminal	
tput smso	Enables boldfaced output	
tput rmso	Disables boldfaced output	

Command	Purpose	Options Covered in This Chapter
trap	Executes a command when a specified signal is received from the operating system	-l displays a listing of signal numbers and their signal designations.
while: do . . done	Repeats an action while a condition exists	

KEY TERMS

.bashrc file — A file in your home directory that you can use to customize your work environment and specify what occurs each time you log in. Each time you start a shell, that shell executes the commands in .bashrc.

algorithm — A sequence of instructions, programming code, or commands that results in a program or that can be used as part of a program.

alias — A name that represents a command. Aliases are helpful in simplifying and automating frequently used commands.

arithmetic operator — A character that represents a mathematical activity. Arithmetic operators include + (addition), - (subtraction), * (multiplication), and / (division).

branch instruction — An instruction that tells a program to go to a different section of code.

case logic — One of the four basic shell logic structures employed in program development. Using case logic, a program can perform one of many actions, depending on the value of a variable and matching results to a test. It is often used when there is a list of several choices.

compiler — A program that reads the lines of code in a source file, converts them to machine-language instructions or calls the assembler to convert them into object code, and creates a machine-language file.

configuration variable — A variable that stores information about the operating system and does not change the value.

control structures — *See* logic structures.

debugging — The process of going through program code to locate errors and then fixing them.

decision logic — One of the four basic shell logic structures used in program development. In decision logic, commands execute only if a certain condition exists. The *if* statement is an example of a coded statement that sets the condition(s) for execution.

defining operator — Used to assign a value to a variable.

environment variable — A value in a storage area that is read by UNIX/Linux when you log in. Environment variables can be used to create and store default settings, such as the shell that you use or the command prompt format you prefer.

evaluating operator — Enables you to evaluate the contents of a variable, such as by displaying the contents.

glob — A character used to find or match file names; similar to a wildcard. Glob characters are part of glob patterns.

glob pattern — A combination of glob characters used to find or match multiple file names.

high-level language — A computer language that uses English-like expressions. COBOL, Visual Basic (VB), C, and C++ are high-level languages.

interpreter — A UNIX/Linux shell feature that reads statements in a program file, immediately translates them into executable instructions, and then runs the instructions. Unlike a compiler, an interpreter does not produce a binary (an executable file) because it translates the instructions and runs them in a single step.

logic structures — The techniques for structuring program code that affect the order in which the code is executed or how it is executed, such as looping back through the code from a particular point or jumping from one point in the code to another. Also called control structures or control logic.

login script — A script that runs just after you log in to your account.

looping logic — One of the four basic shell logic structures used in program development. In looping logic, a control structure (or loop) repeats until some specific condition exists or some action occurs.

nest — When creating program code, a practice of layering statements at two or more levels under an original statement structure.

operand — The variable name that appears to the left of an operator or the variable value that appears to the right of an operator. For example, in NAME=Becky, NAME is the variable name, = is the operator, and Becky is the variable value. Note that no spaces separate the operator and operands.

PATH variable — A path identifier that provides a list of directory locations where UNIX/Linux look for executable programs.

program development cycle — The process of developing a program, which includes (1) creating program specifications, (2) the design process, (3) writing code, (4) testing, (5) debugging, and (6) correcting errors.

prototype — A running model, which lets programmers review a program before committing to its design.

redirection operator — An operator or symbol that changes the input or output data stream from its default direction, such as using > to redirect output to a file instead of to the screen.

relational operator — Compares the relationship between two values or arguments, such as greater than (>), less than (<), equal to (=), and others.

sequential logic — One of four basic logic structures used for program development. In sequential logic, commands execute in the order they appear in the program, except when a branch instruction changes the flow of execution.

shell script operator — The symbols used with shell scripts that define and evaluate information, that perform arithmetic actions, and that perform redirection or piping operations.

shell variable — A variable you create at the command line or in a shell script. It is valuable for use in shell scripts for storing information temporarily.

source file — A file used for storing a program's high-level language statements (code) and created by an editor such as vi or Emacs. To execute, a source file must be converted to a low-level machine language file consisting of object code.

statement — A reference to a line of code that performs an action in a program.

string — A nonnumeric field of information treated simply as a group of characters. Numbers in a string are considered characters rather than digits.

symbolic name — A name used for a variable that consists of letters, numbers, or characters, that is used to reference the contents of a variable, and that often reflects the variable's purpose or contents.

syntax error — A grammatical mistake in a source file or script. Such mistakes prevent a compiler or interpreter from converting the file into an executable file or from running the commands in the file.

REVIEW QUESTIONS

1. Your organization routinely uses scripts, but as some employees have left, there are scripts that contain only command lines and no one is certain of their purpose. What steps can be taken to ensure a way for others to know the purpose of a script?
 - a. Create text documentation of scripts and use the *scriptdoc* command to organize and display the documentation.
 - b. Use the *whatis* command to create and save new documentation for scripts.
 - c. Require that script writers place comment lines inside the scripts using the *#* symbol to begin each comment line.
 - d. Require that scripts be named using the descriptive sentence naming function in UNIX/Linux.
2. Which of the following shells enables the use of scripts? (Choose all that apply.)
 - a. Bash
 - b. csh
 - c. sea
 - d. zsh
3. You frequently use the command *ls -a* and want to save time by just entering *l* to do the same thing. Which of the following commands enables you to set your system to view hidden files by only entering *l*?
 - a. *put l= ls -a*
 - b. *set l to ls -a*
 - c. *set "ls -a" to "l"*
 - d. *alias l="ls -a"*

4. You have written a script, but when you run it there is an error. Which of the following commands can you use to debug your script? (Choose all that apply.)
 - a. *debug -all*
 - b. *sh -v*
 - c. *./ -d*
 - d. *sh -x*
5. You have written a shell program that creates four temporary files. Which of the following commands can you use to remove these files when the script has completed its work?
 - a. *trap*
 - b. *grep*
 - c. *del*
 - d. *clear*
6. Which of the following commands works well for menus used in a script? (Choose all that apply.)
 - a. *do*
 - b. *case*
 - c. *choose*
 - d. *comm*
7. You are currently in the source directory, which is the new directory you have just created for storing and running your scripts. You want to make certain that the source directory is in your default path. Which of the following commands enables you to view the current default path settings?
 - a. *cat PATH*
 - b. *show path*
 - c. *sed PATH!*
 - d. *echo \$PATH*
8. You have created a script for use by your entire department in a commonly accessed directory. Only you are able to run the script, which works perfectly. Which of the following is likely to be the problem?
 - a. You did not link the script.
 - b. You did not give all users in your department execute permission for that script.
 - c. You did not designate to share ownership of the script.
 - d. There are two kinds of scripts, universal and private. You have created a private script and need to convert it to universal.

9. Your current working directory contains a series of files that start with the word “account” combined with a, b, c, d, and e, such as `accounta`, `accountb`, and so on. Which of the following commands enables you to view the contents of all of these files? (Choose all that apply.)
 - a. `ls account "a -e"`
 - b. `less account "a,e"`
 - c. `more account[a,b,c,d,e]`
 - d. `cat account{a to e}`
10. For which of the following logic structures used within a script is *fi* the final line for that logic structure? (Choose all that apply.)
 - a. *loop*
 - b. *case*
 - c. *for*
 - d. *if*
11. Which of the following are examples of arithmetic or relational operators? (Choose all that apply.)
 - a. `!`
 - b. `<`
 - c. `%`
 - d. `*`
12. You have created a series of scripts that use the same environment variables. However, when you run these scripts, some of them do not seem to recognize the environment variables you have set. What is the problem?
 - a. You need to use the *export* command so these variables have global use.
 - b. You are creating too many environment variables, because the maximum number is five.
 - c. You must use the *home* command to make these variables native to your home directory.
 - d. Only the system administrator can create environment variables and you should contact her to create the ones you need to use.

13. You have spent the last two hours creating a report in a file and afterwards you use *cat* to create a new file. Unfortunately the new file name you used was the same as the name you used for the report, and now your report is gone. What should you do next time to prevent this from happening?
 - a. Enter the *cat -s* command before you start.
 - b. Enter the command, *set -o noclobber* before you start.
 - c. Always use the *cat -m* command when you use *cat* to create a file, because this command checks to see if the file already exists.
 - d. After you created the report file you should have used the *chmod a-o* command to prevent the file from being deleted or overwritten.
14. You have remotely logged into a computer running UNIX or Linux, but you are not certain about which operating system you are using. However, when you display the contents of the _____ variable it shows which operating system you are using.
 - a. OP
 - b. OPTIND
 - c. OID
 - d. OSTYPE
15. What command can you use to view the environment and configuration variables already configured on your system?
 - a. *var*
 - b. *envvar*
 - c. *printenv*
 - d. *let -all*
16. Which of the following are valid expressions? (Choose all that apply.)
 - a. *let x=5*9*
 - b. *let x=y+10*
 - c. *let m=12/4*
 - d. *let r=128-80*
17. When you type *for wood maple spruce oak pine* at the command line and then press Enter, what should you type next at the *>* prompt?
 - a. *do*
 - b. *go*
 - c. *fi*
 - d. *term*

18. You want to store a long listing of your files in a variable called `myfiles`. Which of the following commands enables you to do this?
- `let ls -l=myfiles`
 - `echo ls -l > myfiles`
 - `myfiles=`ls -l``
 - `let ls -l > myfiles`
19. What error is in the following script code?
- ```
case "selection" in
" i ") ./listscript ;;
" ii ") ./numberscript ;;
" iii ") ./findscript ;;
esac
```
- All references to `;;` should be replaced with a back quote.
  - There should be a dollar sign in front of `selection`, as in `"$selection"`
  - There should be no double quote marks in the code.
  - The code must end with the statement, `"out"`.
20. You are working with a colleague on a script called `value` that updates several files. You want to test the script, but not update the files. Which of the following commands can you use?
- `test -nouupdate value`
  - `trap -u value`
  - `set -u value`
  - `sh -n value`
21. You only have to enter the name of a script to have it run, such as entering `myscript`. What setting enables you to do this?
- You have set the `SCRIPT` environment variable to 1 instead of the default 0.
  - Right after you logged in you entered `setup scripts`.
  - The first line in your scripts is always `run`, which enables scripts to be run in this way.
  - You have placed the directory from which you run the scripts in your `PATH` variable.
22. What would you expect to find in the `HOME` environment variable?
23. What is the difference between a compiler and an interpreter?
24. What command would you use to place the cursor in row 10 and column 15 on the screen or in a terminal window?
25. What is the purpose of a login script?

## HANDS-ON PROJECTS



### NOTE

Complete these projects using the command line, such as from a terminal window, and log in using your own account and home directory.



HANDS-ON  
PROJECTS

### Project 6-1

Before setting one or more environment variables, it is a good idea to view their current configurations. In this project, you use the `printenv` command to view a list of your environment variables.

#### To see a list of your environment variables:

1. Your list of environment variables might be longer than the default screen or terminal window size, so it can be helpful to pipe the output into the `more` command. Type **`printenv | more`** and press **Enter**.
2. Record some examples of environment variables. Press the **spacebar** to advance through the listing one screen at a time.
3. Type **`clear`** and press **Enter** to clear the screen.
4. Next, use the `printenv` command to view the contents of two variables: `SHELL` and `PATH`. Type **`printenv SHELL PATH`** and press **Enter**. (See Figure 6-5.)
5. Type **`clear`** and press **Enter** to clear the screen for the next project.



HANDS-ON  
PROJECTS

### Project 6-2

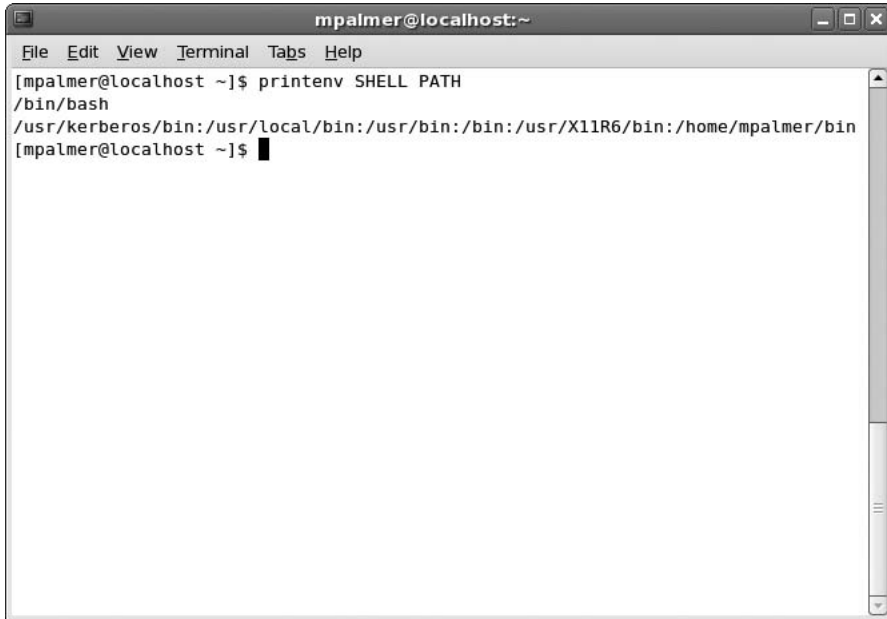
This project enables you to use the defining and evaluating operators to learn how they work. You begin by assigning a value to a variable and then view the contents of the variable you assigned. You then learn how to assign a variable that contains spaces, and you compare using single and double quotation marks to evaluate the contents of a variable. Finally, you use the back quote marks to execute a command and store the result in a variable.

#### To create a variable, and assign it a value:

1. Type **`DOG=Shepherd`** and press **Enter**.  
You've created the variable `DOG` and set its value to `Shepherd`.

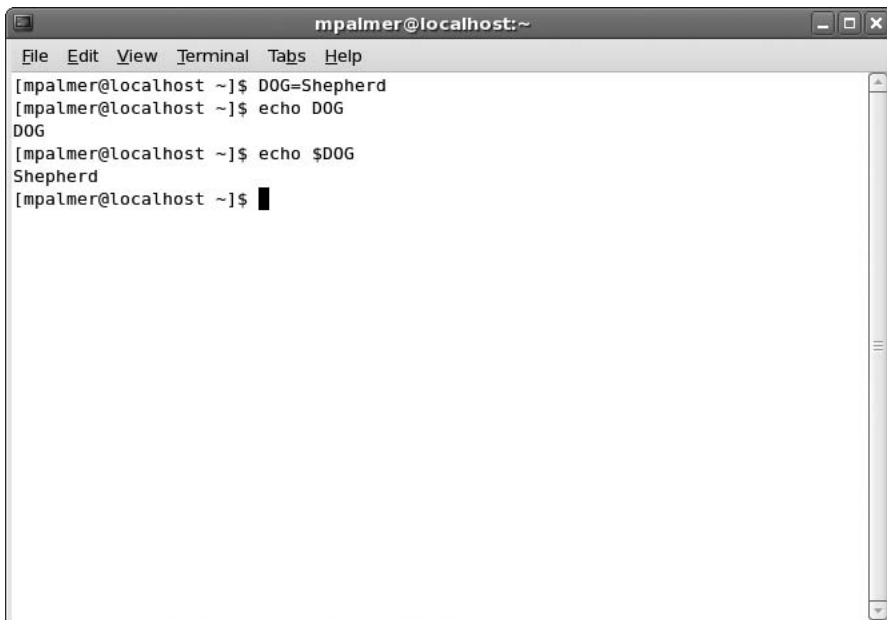
#### To see the contents of a variable:

1. Type **`echo DOG`** and press **Enter**.  
You see the word "DOG."
2. To see the contents of the `DOG` variable, you must precede the name of the variable with a `$` operator. Type **`echo $DOG`** and press **Enter**. You see the word "Shepherd." (See Figure 6-6.)



```
mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ printenv SHELL PATH
/bin/bash
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/mpalmer/bin
[mpalmer@localhost ~]$
```

Figure 6-5 Using *printenv* to view only two environment variables



```
mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ DOG=Shepherd
[mpalmer@localhost ~]$ echo DOG
DOG
[mpalmer@localhost ~]$ echo $DOG
Shepherd
[mpalmer@localhost ~]$
```

Figure 6-6 Viewing the contents of a variable

To use double quotation marks to set a variable to a string of characters containing spaces:

1. Type **MEMO="Meeting will be at noon today"** and press **Enter**.
2. Type **echo \$MEMO** and press **Enter**.

You see the contents of the MEMO variable: Meeting will be at noon today.

To demonstrate how double quotation marks do not suppress the viewing of a variable's contents, but single quotation marks do suppress the viewing:

1. Type **echo '\$HOME'** and press **Enter**.

You see \$HOME on the screen.

2. Type **echo "\$HOME"** and press **Enter**.

You see the path of your home directory on the screen.

To demonstrate the back quote operator for executing a command:

1. Type **TODAY=`date`** and press **Enter**. This command creates the variable TODAY, executes the *date* command, and stores the output of the *date* command in the variable TODAY. (No output appears on the screen.)
2. Type **echo \$TODAY** and press **Enter**. You see the output of the *date* command that was executed in Step 1.
3. Type **clear** and press **Enter** to clear the screen for the next project.



## Project 6-3

In this project, you employ the *let* command to practice using arithmetic operators to set the contents of a shell variable. First, you use an expression with constants (no variables), and then you use an expression containing a variable.

To practice using the arithmetic operators:

1. Type **let X=10+2\*7** and press **Enter**.
2. Type **echo \$X** and press **Enter**. You see 24 on the screen.
3. Type **let Y=X+2\*4** and press **Enter**.
4. Type **echo \$Y** and press **Enter**. You see 32 on the screen, as shown in Figure 6-7.
5. Type **clear** and press **Enter** to clear the screen for the next project.

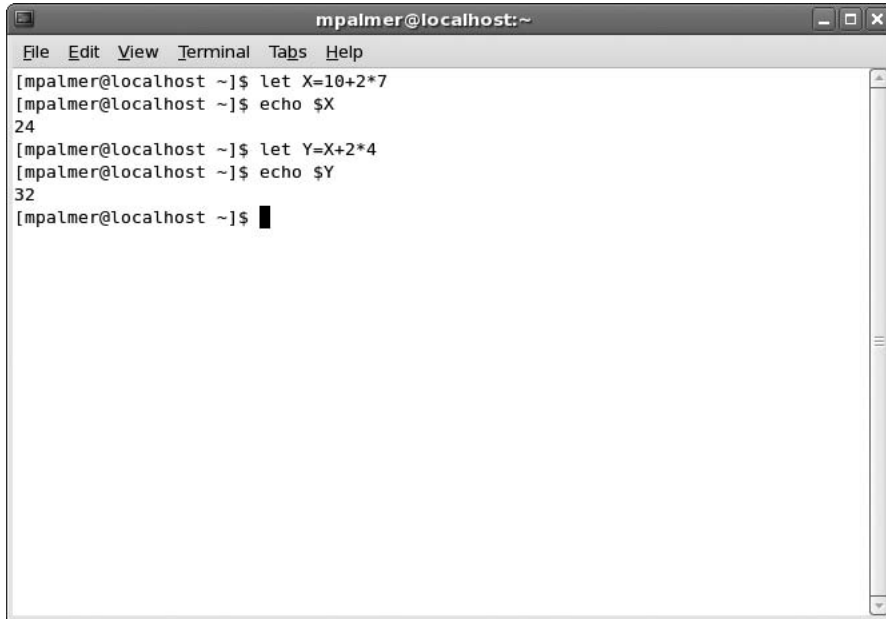


## Project 6-4

In this project, you export a shell variable to make it globally recognized.

To demonstrate the use of the *export* command:

1. Type **cat > testscript** and press **Enter**.



```

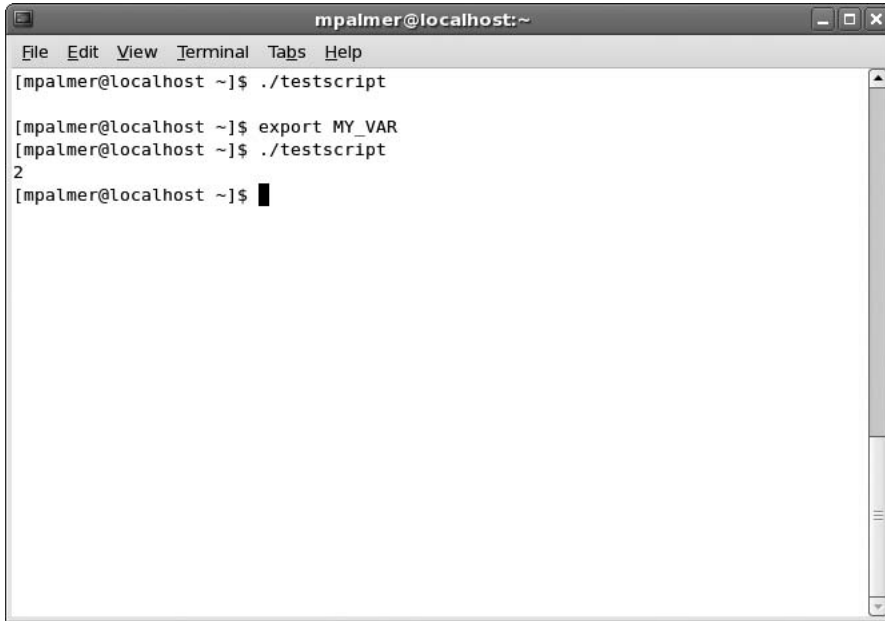
mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ let X=10+2*7
[mpalmer@localhost ~]$ echo $X
24
[mpalmer@localhost ~]$ let Y=X+2*4
[mpalmer@localhost ~]$ echo $Y
32
[mpalmer@localhost ~]$ █

```

**Figure 6-7** Assigning shell script variables using *let*

2. Type **echo \$MY\_VAR** and press **Enter**.
3. Type **Ctrl+d**. You have created a simple shell script named **testscript**. Its only function is to display the value of the **MY\_VAR** variable.
4. To make the script executable, type **chmod ugo+x testscript**, and press **Enter**.
5. Type **MY\_VAR=2**, and press **Enter**.
6. Type **echo \$MY\_VAR** and press **Enter** to confirm the preceding operation. You see 2 on the screen.
7. Next look at the list of environment variables. Type **printenv | more** and press **Enter**.  
Look carefully as you scroll through the output of the *printenv* command. You do not see the **MY\_VAR** variable.
8. Type **clear** and press **Enter** to clear the screen.
9. Execute the shell script by typing **./testscript** and pressing **Enter**. The script displays a blank line. This is because it does not have access to the shell variable **MY\_VAR**.
10. Make the variable available to the script by typing **export MY\_VAR** and pressing **Enter**.
11. Execute the script again by typing **./testscript** and pressing **Enter**. This time, the value 2 appears. (See Figure 6-8 on the next page.)

12. Now look at your list of environment variables by typing **printenv | more** and pressing **Enter**. Again, look carefully as you scroll through the list. This time, you see **MY\_VAR** listed.
13. Type **clear** and press **Enter** to clear the screen for the next project.



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$./testscript

[mpalmer@localhost ~]$ export MY_VAR
[mpalmer@localhost ~]$./testscript
2
[mpalmer@localhost ~]$

```

**Figure 6-8** Using the *export* command



## Project 6-5

In Hands-on Project 6-4, you had to use `./` before `testscript` because your current working directory is not in your `PATH` environment variable. In this project, you view the contents of the `PATH` variable. Next, you add the current working directory to the `PATH` variable and run `testscript` without using the `./` characters.

### To see the contents of the `PATH` variable:

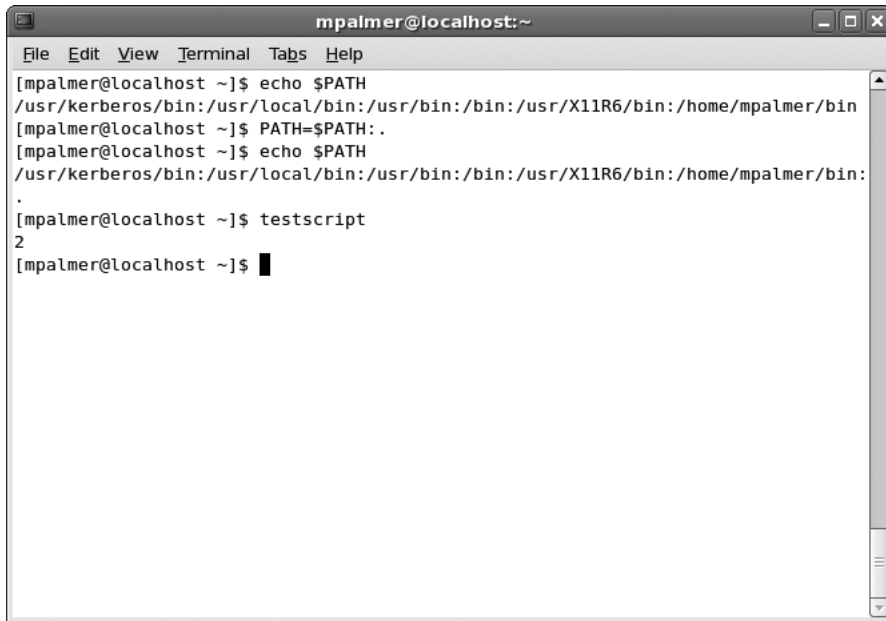
1. Type **echo \$PATH** and press **Enter**.

You see a list of directories. Notice that the path names are separated by colons (:).

### To add the current working directory to the `PATH` variable:

1. Type **PATH=\$PATH:.** and press **Enter**.
2. Type **echo \$PATH** and press **Enter**. The dot (.) is now appended to the list.

3. You can now run scripts in your current working directory without typing the `./` characters before their names. Test this by typing **testscript** and pressing **Enter**. You see testscript execute, as in Figure 6-9.



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/mpalmer/bin
[mpalmer@localhost ~]$ PATH=$PATH:.
[mpalmer@localhost ~]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/mpalmer/bin:
.
[mpalmer@localhost ~]$ testscript
2
[mpalmer@localhost ~]$

```

**Figure 6-9** Adding your current working directory to the PATH variable to run shell scripts



## Project 6-6

In this project, you gain further experience in writing a very simple shell script using sequential logic. In these steps, you create the shell script, seqtotal.

### To demonstrate sequential logic:

1. Type **vi seqtotal** and press **Enter**.
2. Type **i** to switch to vi's insert mode.
3. Type the following lines:
 

```

let a=1
let b=2
let c=3
let total=a+b+c
echo $total

```
4. Press **Esc** to switch to vi's command mode.
5. Type **:x** and press **Enter** to save the file and exit vi.



6. Next test the new shell script, `seqtotal`. (To save a few keystrokes, use the `sh` command instead of the `chmod` command.) Type **sh seqtotal** and press **Enter**.

You see the output of the script, which is 6.



## Project 6-7

This project provides your first introduction to using an *if* statement in a shell script and demonstrates decision logic. In the first set of steps, you create a script using a basic *if* statement. Then, in the second set of steps, you modify your script to include an *if* statement nested within an *if* statement.

6

**To demonstrate the *if* statement as well as to implement decision logic:**

1. Type **vi veg\_choice** and press **Enter**.
2. Type **i** to switch to vi's insert mode.
3. Type the following lines:

```
echo -n "What is your favorite vegetable? "
read veg_name
if ["$veg_name" = "broccoli"]
then
 echo "Broccoli is a healthy choice."
else
 echo "Don't forget to eat your broccoli also."
fi
```

4. Be certain your editing session looks like the one in Figure 6-10. Press **Esc** to switch to vi's command mode.
5. Type **:x** and press **Enter** to save the file and exit vi.
6. Make the script executable by typing **chmod ugo+x veg\_choice** and pressing **Enter**. Next, run the script by typing **./veg\_choice** and pressing **Enter** (if you are continuing the same terminal session from Hands-on Project 6-5, you don't need to type **./**, because the `PATH` environment variable still contains the current working directory.)
7. When asked to enter the name of your favorite vegetable, answer **broccoli**.
8. Run the script again and respond with **corn** or some other vegetable name.



**TIP**

Remember that you must use the `chmod` command first to make the script executable. Then, after the command prompt, type the path to the script plus the script's name to execute it. Another way to run the script is to use the `sh` command, as you did with the `seqtotal` script. Yet a third alternative after you use the `chmod` command is to add your current working directory to your path, as you learned in Hands-on Project 6-5. Then, you just enter the name of your script to run it.

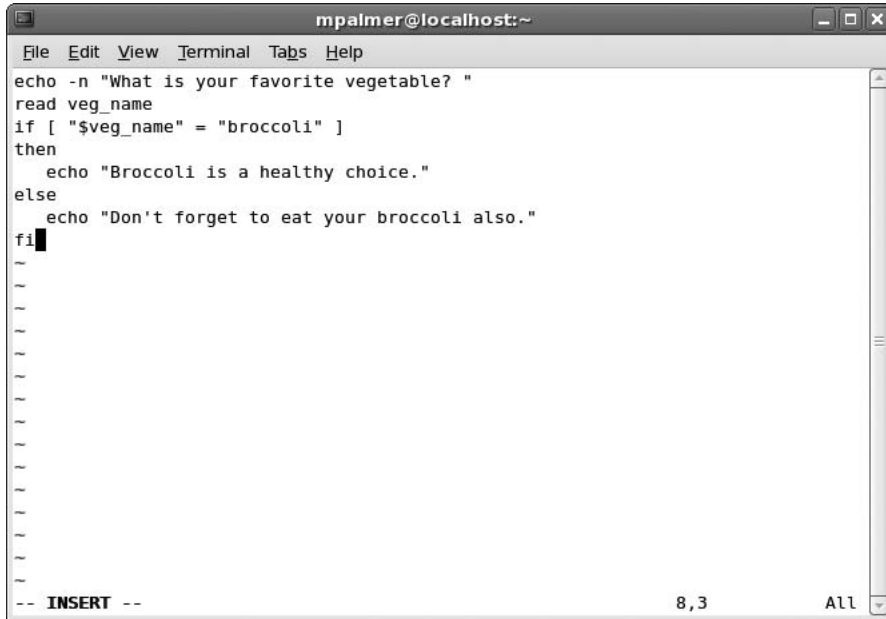


Figure 6-10 Creating the veg\_choice script

**To practice writing a nested *if* statement:**

1. Open the veg\_choice file in vi or Emacs.
2. Edit the file so it contains the following lines. (Code has been added to the *else* part of the original *if* statement. See the lines in bold.)

```
echo -n "What is your favorite vegetable? "
read veg_name
if ["$veg_name" = "broccoli"]
then
 echo "Broccoli is a healthy choice."
else
 if ["$veg_name" = "carrots"]
 then
 echo "Carrots are great for you."
 else
 echo "Don't forget to eat your broccoli also."
 fi
fi
```

3. Execute the script and respond with **carrots** when asked for your favorite vegetable. You should see the response "Carrots are great for you."
4. Type **clear** and press **Enter** to clear the screen for the next project.



## Project 6-8

In this project, you learn to use a *for* loop in a shell script and on the command line, both demonstrating how looping logic works.

### To demonstrate looping logic in a shell script:

1. Create the file `our_users` with `vi` or `Emacs`.
2. Type the following lines into the file:

```
for USERS in john ellen tom becky eli jill
do
 echo $USERS
done
```

3. Save the file and exit the editor.
4. Give the file execute permission, and run it. Your results should look similar to those shown in Figure 6-11.

A screenshot of a terminal window titled 'mpalmer@localhost:~'. The window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The terminal shows the following commands and output:

```
[mpalmer@localhost ~]$ vi our_users
[mpalmer@localhost ~]$ chmod a+x our_users
[mpalmer@localhost ~]$./our_users
john
ellen
tom
becky
eli
jill
[mpalmer@localhost ~]$
```

**Figure 6-11** Running the `our_users` script to execute a sample *for* loop

### To demonstrate entering the same *for* loop at the command line:

1. At the command line, enter `for USERS in john ellen tom becky eli jill` and press **Enter**.
2. At the `>` prompt, type `do` and press **Enter**.

3. Type **echo \$USERS** and press **Enter**.
4. Type **done** and press **Enter**. What do you see on the screen?
5. Type **clear** and press **Enter** to clear the screen for the next project.



## Project 6-9

In this project, you create a *for* loop and use the brackets wildcard format to loop through each element in a *for* statement, which consists of simulated book chapters. You first create the files chap1 through chap4. Next, you create a script that displays the contents of each file using the *more* command.

**To create the sample chapter file and use wildcards in a *for* loop:**

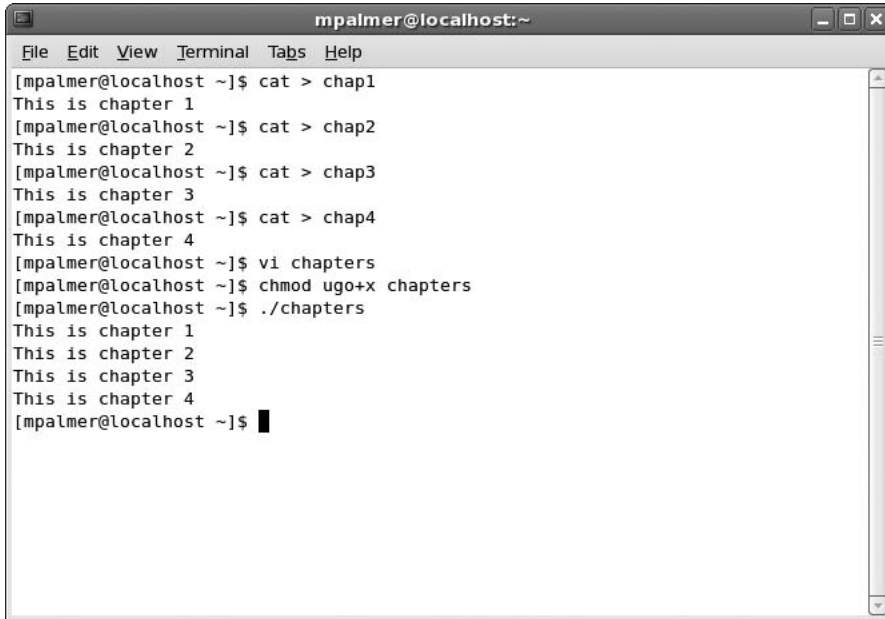
1. Type **cat > chap1** and press **Enter**.
2. Type **This is chapter 1** and press **Enter**.
3. Type **Ctrl+d**. The file chap1 is created.
4. Type **cat > chap2** and press **Enter**.
5. Type **This is chapter 2** and press **Enter**.
6. Type **Ctrl+d**. The file chap2 is created.
7. Type **cat > chap3** and press **Enter**.
8. Type **This is chapter 3** and press **Enter**.
9. Type **Ctrl+d**. The file chap3 is created.
10. Type **cat > chap4** and press **Enter**.
11. Type **This is chapter 4** and press **Enter**.
12. Type **Ctrl+d**. The file chap4 is created.
13. Use the vi or Emacs editor to create the shell script, chapters. The script should have these lines:

```
for file in chap[1234]; do
 more $file
done
```
14. Save the file and exit the editor.
15. Give the file execute permission, and test it. You see output similar to Figure 6-12.



## Project 6-10

The *while* statement is another example of looping logic in addition to the *for* statement. In this project, you first create a shell program that contains a basic *while* statement. Next, you create a shell program as might be used for an onscreen data input form to store name and address information in a flat data file.



```
mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ cat > chap1
This is chapter 1
[mpalmer@localhost ~]$ cat > chap2
This is chapter 2
[mpalmer@localhost ~]$ cat > chap3
This is chapter 3
[mpalmer@localhost ~]$ cat > chap4
This is chapter 4
[mpalmer@localhost ~]$ vi chapters
[mpalmer@localhost ~]$ chmod ugo+x chapters
[mpalmer@localhost ~]$./chapters
This is chapter 1
This is chapter 2
This is chapter 3
This is chapter 4
[mpalmer@localhost ~]$
```

Figure 6-12 Executing the chapters shell script

### To use a basic *while* statement in a shell script:

1. Use the vi or Emacs editor to create a shell script called colors.
2. Enter the following lines of code:

```
echo -n "Try to guess my favorite color: "
read guess
while ["$guess" != "red"]; do
 echo "No, not that one. Try again. "; read guess
done
```

3. Save the file and exit the editor.
4. Give the file execute permission, and test it. Type **clear** and press **Enter** to clear the screen.

Another example of the *while* statement is a data-entry form.

### To create a *while* loop that serves as a data-entry form:

1. Use vi or Emacs to create a script file, nameaddr.
2. Type these lines into the file:

```
looptest=y
while ["$looptest" = y]
do
 echo -n "Enter Name: "; read name
```

```

echo -n "Enter Street: "; read street
echo -n "Enter City: "; read city
echo -n "Enter State: "; read state
echo -n "Enter Zip Code: "; read zip
echo -n "Continue? (y)es or (n)o: "; read looptest
done

```

3. Save the file and exit the editor.
4. Give the file execute permission, and test it. As you test the script, enter several names and addresses. When you finish, answer **n** (for no) when the script asks you, "Continue? (y)es or (n)o." (See Figure 6-13.) Type **clear** and press **Enter** to clear the screen for the next project.



```

mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ vi nameaddr
[mpalmer@localhost ~]$ chmod a+x nameaddr
[mpalmer@localhost ~]$./nameaddr
Enter Name: Sara Lopez
Enter Street: 142 North Main Street
Enter City: Hanover
Enter State: NC
Enter Zip Code: 28766
Continue? (y)es or (n)o: y
Enter Name: Jim Mason
Enter Street: 722 Rutgers Lane
Enter City: Asheville
Enter State: NC
Enter Zip Code: 28801
Continue? (y)es or (n)o: n
[mpalmer@localhost ~]$

```

Figure 6-13 Using the nameaddr script



## Project 6-11

Case logic is often used when many choices are given through a program or when many responses can be made on the basis of one choice. In this project, you create a shell script that employs case logic to respond to your favorite color (many possible responses selected on the basis of one choice).

### To demonstrate case logic:

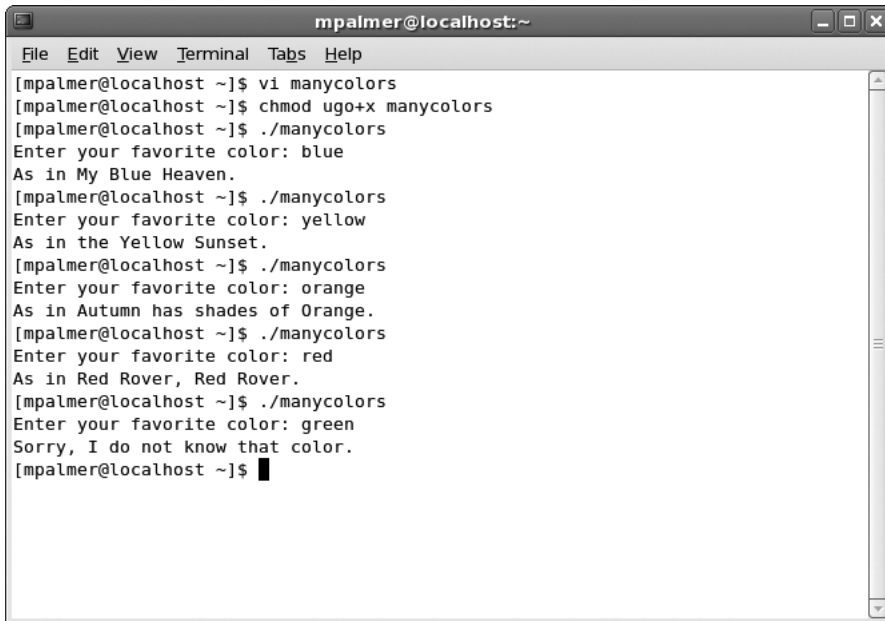
1. Use the vi or Emacs editor to create the manycolors shell script.

Type these lines into the file:

```
echo -n "Enter your favorite color: "; read color
case "$color" in
 "blue") echo "As in My Blue Heaven.>";;
 "yellow") echo "As in the Yellow Sunset.>";;
 "red") echo "As in Red Rover, Red Rover.>";;
 "orange") echo "As in Autumn has shades of Orange.>";;
 *) echo "Sorry, I do not know that color.>";;
esac
```

2. Save the file and exit the editor
3. Give the file execute permission, and test it. (See Figure 6-14.)

6



```
mpalmer@localhost:~
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ vi manycolors
[mpalmer@localhost ~]$ chmod ugo+x manycolors
[mpalmer@localhost ~]$./manycolors
Enter your favorite color: blue
As in My Blue Heaven.
[mpalmer@localhost ~]$./manycolors
Enter your favorite color: yellow
As in the Yellow Sunset.
[mpalmer@localhost ~]$./manycolors
Enter your favorite color: orange
As in Autumn has shades of Orange.
[mpalmer@localhost ~]$./manycolors
Enter your favorite color: red
As in Red Rover, Red Rover.
[mpalmer@localhost ~]$./manycolors
Enter your favorite color: green
Sorry, I do not know that color.
[mpalmer@localhost ~]$
```

Figure 6-14 Using the manycolors shell script



## Project 6-12

The *tput* command enables you to initialize the screen and position the cursor and text in an appealing way. This project introduces you to *tput*. First, you enter the command directly from the command line. Next, you create a sample shell script and menu to understand more about this command's capabilities.

### To use *tput* directly from the command line:

1. Type the following command sequence, and press **Enter**:

```
tput clear ; tput cup 10 15 ; echo "Hello" ; tput cup 20 0
```

In the results of this command sequence, the screen clears; the cursor is positioned at row 10, column 15, on the screen; the word “Hello” is printed; and the prompt’s position is row 20, column 0.

### To create a sample input menu in a shell script:

1. Use the vi or Emacs editor to create a screen-management script, `scrmanage`, containing the following lines:
 

```
tput cup $1 $2 # place cursor on row and col
tput clear # clear the screen
bold='tput smso' # set stand-out mode - bold
offbold='tput rmso' # reset screen - turn bold off
echo $bold # turn bold on
tput cup 10 20; echo "Type Last Name:" # bold caption
tput cup 12 20; echo "Type First Name:" # bold caption
echo $offbold # turn bold off
tput cup 10 41; read lastname # enter last name
tput cup 12 41; read firstname # enter first name
```
2. Save the file and exit the editor.
3. Give the file execute permission, and then test it. (See Figure 6-15.) Clear the screen for the next project.



The single back quotes around ``tput smso`` and ``tput rmso`` must be in the direction as shown or the bold/unbold command does not work. This single back quote mark is found in the upper-left corner of most keyboards, usually on the same key as the tilde (~).



## Project 6-13

In this project, you first compare the use of the `sh -v` and `sh -x` options in terms of the output to the screen. Next, you practice debugging a shell script using `sh -v`.

### To compare the results of the `sh -v` and `sh -x` options to debug a script:

1. Type **sh -v colors** (remember that `colors` is the script you created earlier in this chapter in which the favorite color is red), and press **Enter**.
2. Type **green** and press **Enter**.
3. Type **red** and press **Enter**. Notice that the command lines are printed.
4. Type **sh -x colors** and press **Enter**.
5. Type **green** and press **Enter**.
6. Type **red** and press **Enter**. Now, the command lines and arguments are displayed with a plus in front of them. Figure 6-16 illustrates the output of the `sh -v` and `sh -x` options.



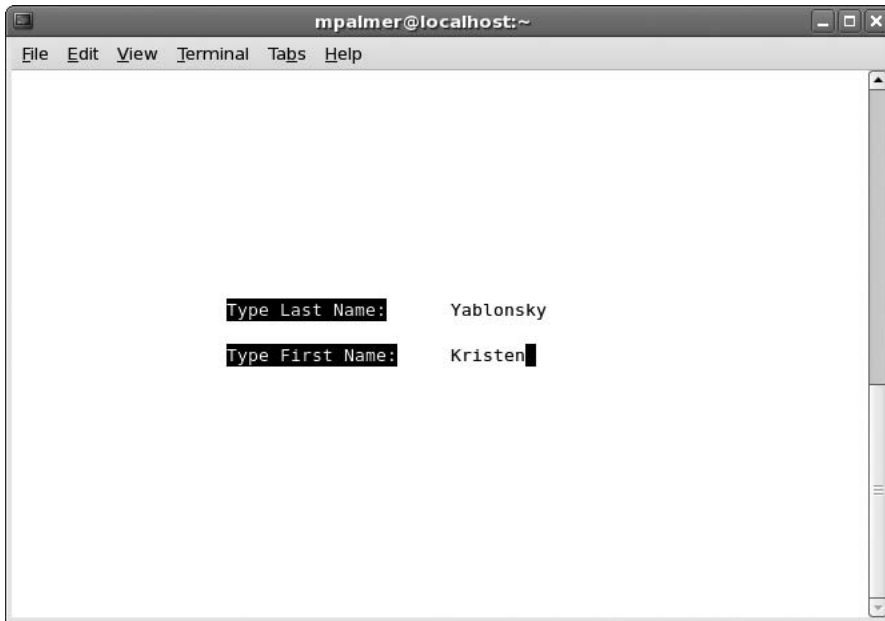


Figure 6-15 Using `tput` in a script to produce a simple menu

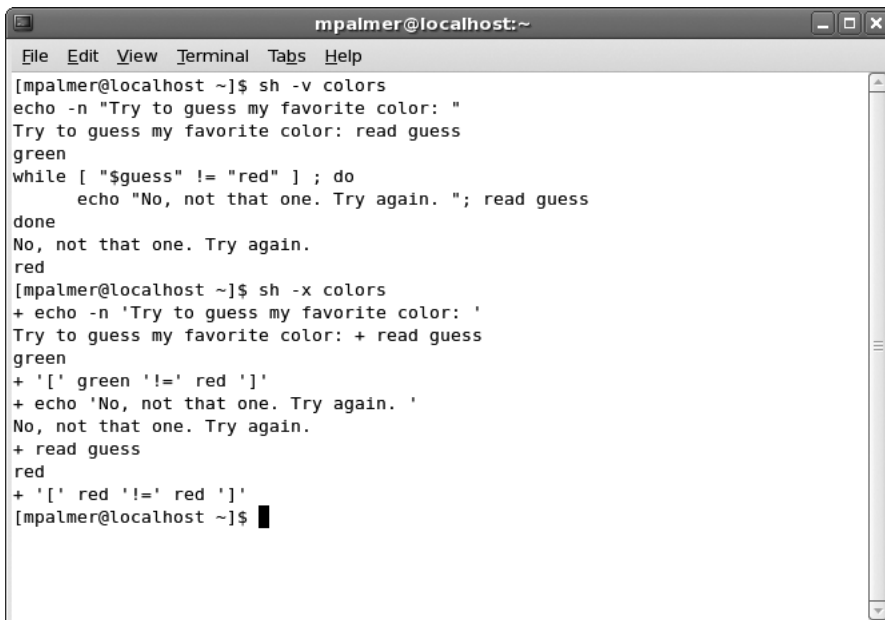


Figure 6-16 Comparing `sh -v` and `sh -x` for debugging

**To practice debugging a shell script:**

1. Use the vi or Emacs editor to open the colors script for editing.
2. Go to the third line and delete the closing (right) bracket (]) after “red” and then exit, saving your change.
3. Type **sh -v colors** and press **Enter**.
4. Type **green** and press **Enter**. In the final line of output, you’ll see a note that shows the closing bracket is missing on line 3 of the colors script:

```
colors: line 3: [: missing '']
```

5. Use vi or Emacs to open the colors script and put the missing closing bracket back in.
6. Delete the **echo** command (only the word echo and not the entire command line) on the fourth line of the colors script. Close the editor and save your work.
7. Type **sh -x colors** and press **Enter**.
8. Type **green** and press **Enter**. Notice in the message that a command is missing on line 4:

```
colors: line 4: No, not that one try again. : command
not found
```

9. Type **red** and press **Enter** to exit the script, or press **Ctrl+z** to exit.
10. Open the colors script using the vi or Emacs editor, retype the **echo** command on line 4, and close and save your work.



## Project 6-14

In this project, students learn how to create an alias.

**To create an alias:**

1. To create an alias called **ll** for the **ls** command, type **alias ll="ls -l"**, and press **Enter**. Now, when you use the new **ll** alias, the **ls -l** command executes automatically.
2. Test the alias by typing **ll** and pressing **Enter**. You see a long directory listing.



## Project 6-15

This project is the first in a series of projects to develop a sample application that tracks telephone number and other information for employees in an organization. In this project, you first ensure that you have a source subdirectory in which to store the source files you develop. Next, you create the initial menu that users see when they execute the application. Be certain you retain the phmenu script file that you create here so you can use it in later projects.

**To set up your source subdirectory:**

1. In Chapter 4, you created a source subdirectory under your home directory. With your home directory as your current working directory (type **cd** and press **Enter**), type **ls** and press **Enter** to ensure that your source subdirectory exists. If it does not exist, type **mkdir source**, and press **Enter**.
2. Type **cd source** and press **Enter** to make the source directory your current working directory. Be certain you are in your source directory for all of the projects that follow in this chapter. In this way, you ensure that your application files are in one place.

**To begin work on the menu for your application:**

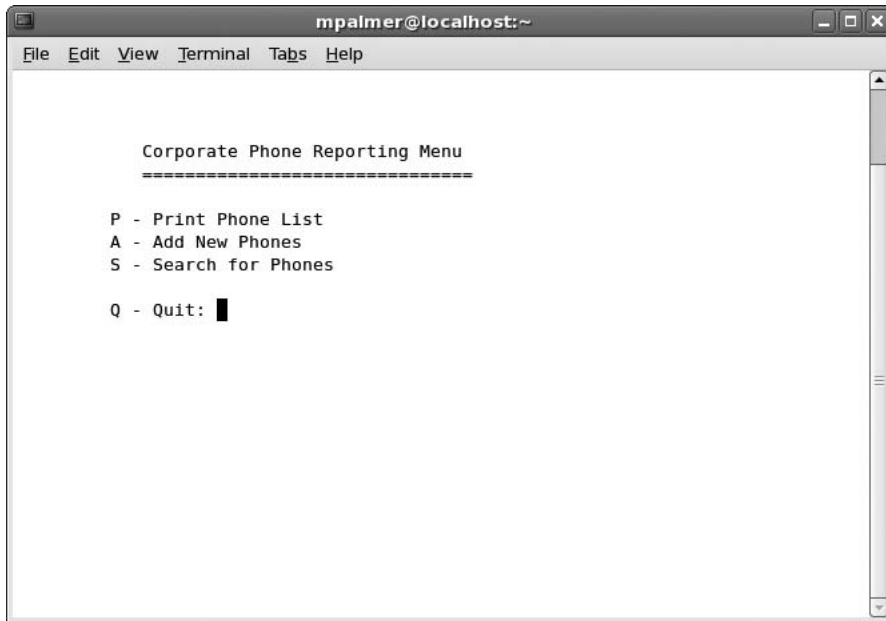
1. Use the vi or Emacs editor to enter the phmenu script shown next.

```
#=====
Script Name: phmenu
By: Your initials here
Date: Today's date
Purpose: A menu for the Corporate Phone List
Command Line: phmenu
#=====
loop=y
while ["$loop" = y]
do
 clear
 tput cup 3 12; echo "Corporate Phone Reporting Menu"
 tput cup 4 12; echo "===== "
 tput cup 6 9; echo "P - Print Phone List"
 tput cup 7 9; echo "A - Add New Phones"
 tput cup 8 9; echo "S - Search for Phones"
 tput cup 10 9; echo "Q - Quit: "
 tput cup 10 19;
 read choice || continue
done
```

2. Save the file and exit the editor.
3. Give the file execute permission by typing **chmod a+x phmenu** and pressing **Enter**. Next, test the script by typing **./phmenu** and pressing **Enter**. (See Figure 6-17.) (You have to press **Ctrl+c** to exit the script because the Quit option has not yet been programmed.)
4. Clear the screen for the next project.

**Project 6-16**

The data file that you use for your telephone number application is called `corp_phones`. In this project, you ensure that the `corp_phones` file is created and contains some preliminary



**Figure 6-17** Running the phmenu script

data that is useful for testing your application as you continue to develop it. You also practice extracting information from the file by using the *grep* command.

#### To create the corp\_phones file:

1. In Chapter 4, you created the corp\_phones, corp\_phones1, and corp\_phones2 files in your home directory and performed projects to manipulate and alter data in those files. If those files are still in your home directory, use the **rm** command to delete them. This ensures two things: that the corp\_phones file you create next has the proper information for using your new telephone number application, and that you know what is in the corp\_phones file for the projects you complete in this chapter. Toward that end, some of the data you now enter is slightly different than what you used in the files you created in Chapter 4.
2. If you are in your home directory, type **cd source**, and press **Enter**. Use the **pwd** command to ensure you are in the source directory.
3. Use the vi or Emacs editor to create the corp\_phones file.
4. Enter the following records in the file:

```

219-555-4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219-555-4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219-555-4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
219-555-4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219-555-4544:Polk:John:S:4520:Accountant:09-22-2001
219-555-4501:Robinson:Albert:J:4501:Secretary:08-12-1997

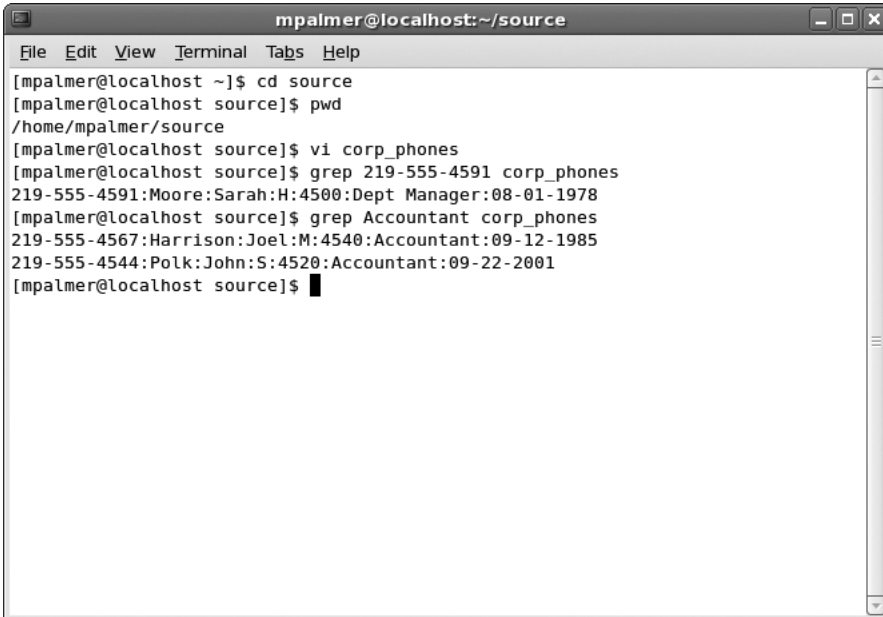
```

5. Save the file and exit the editor.
6. Type **grep 219-555-4591 corp\_phones** and press **Enter** to search for a specific telephone number.

The output should look like this:

```
219-555-4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
```

7. Type **grep Accountant corp\_phones** and press **Enter** to search the file for all accountants. (See Figure 6-18.)



```
mpalmer@localhost:~/source
File Edit View Terminal Tabs Help
[mpalmer@localhost ~]$ cd source
[mpalmer@localhost source]$ pwd
/home/mpalmer/source
[mpalmer@localhost source]$ vi corp_phones
[mpalmer@localhost source]$ grep 219-555-4591 corp_phones
219-555-4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
[mpalmer@localhost source]$ grep Accountant corp_phones
219-555-4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219-555-4544:Polk:John:S:4520:Accountant:09-22-2001
[mpalmer@localhost source]$
```

**Figure 6-18** Working with the corp\_phones data file



## Project 6-17

The phmenu menu script that you created and tested in Hands-on Project 6-15 still needs some work, because it doesn't contain a way to call applications after they are selected. In this project, you solve this problem by adding case logic to the menu.

### To make additions to the phone menu script:

1. If you have interrupted your development process and have just logged back in or switched to another directory, change to the source directory and use the *pwd* command to verify you are in that directory.
2. Use the vi or Emacs editor to open the phmenu file in your source directory. In the following lines, add the lines that are boldfaced to the script:

```
#=====
```

```

Script Name: phmenu
By: Your initials here
Date: November 2009
Purpose: A menu for the Corporate Phone List
Command Line: phmenu
#=====
phonefile=~/.source/corp_phones
loop=y
while ["$loop" = y]
do
 clear
 tput cup 3 12; echo "Corporate Phone Reporting Menu"
 tput cup 4 12; echo "=====
 tput cup 6 9; echo "P - Print Phone List"
 tput cup 7 9; echo "A - Add New Phones"
 tput cup 8 9; echo "S - Search for Phones"
 tput cup 10 9; echo "Q - Quit: "
 tput cup 10 19;
 read choice || continue
 case $choice in
 [Aa]) ./phoneadd ;;
 [Pp]) ./phlist1 ;;
 [Ss]) ./phonefind ;;
 [Qq]) exit ;;
 *) tput cup 14 4; echo "Invalid Code"; read choice ;;
 esac
done

```

3. Save the file and exit the editor.
4. Test the script. (Acceptable entries are A, a, P, p, S, s, V, v, Q, and q. Any other entries cause the message "Invalid Code" to appear.) Type **Ctrl+c** to exit when you are finished testing.



## Project 6-18

It can be useful to have a way to display unformatted file data, in such a display the records appear exactly as they are stored in the data file. This enables the application developer to ensure that records are accurate and that the application is working as expected. Also, some users might be interested in viewing the raw data in a file, so they can ensure it is accurately entered. In this project, you add an option in phmenu to use the *less* command to view the contents of the corp\_phones file.

### To use the *less* command to view unformatted records:

1. Be certain you are in the source directory. Next, open phmenu in the editor of your choice, and add the two boldfaced lines shown below:

```

#=====
Script Name: phmenu

```

```

By: Your initials here
Date: Today's date
Purpose: A menu for the Corporate Phone List
Command Line: phmenu
#=====
phonefile=~/.source/corp_phones
loop=y
while ["$loop" = y]
do
 clear
 tput cup 3 12; echo "Corporate Phone Reporting Menu"
 tput cup 4 12; echo "=====
 tput cup 6 9; echo "P - Print Phone List"
 tput cup 7 9; echo "A - Add New Phones"
 tput cup 8 9; echo "S - Search for Phones"
 tput cup 9 9; echo "V - View Phone List"
 tput cup 10 9; echo "Q - Quit: "
 tput cup 10 19;
 read choice || continue
 case $choice in
 [Aa]) ./phoneadd ;;
 [Pp]) ./phlist1 ;;
 [Ss]) ./phonefind ;;
 [Vv]) less $phonefile ;;
 [Qq]) exit ;;
 *) tput cup 14 4; echo "Invalid Code"; read choice ;;
 esac
done

```

2. Save the file and exit the editor.
3. Test the script.



## Project 6-19

In your menu design, when users enter P or p, the application should print a phone list. The *awk* program offers a good example of how the UNIX/Linux shell programmer can accelerate development because a single *awk* command can select fields from many records and display them in a specified format on the screen. In this project, you develop the *phlist1* script and use *awk* to display a phone list.

### To create the *phlist1* script:

1. From the source directory, use the editor of your choice to create the *phlist1* script as follows:

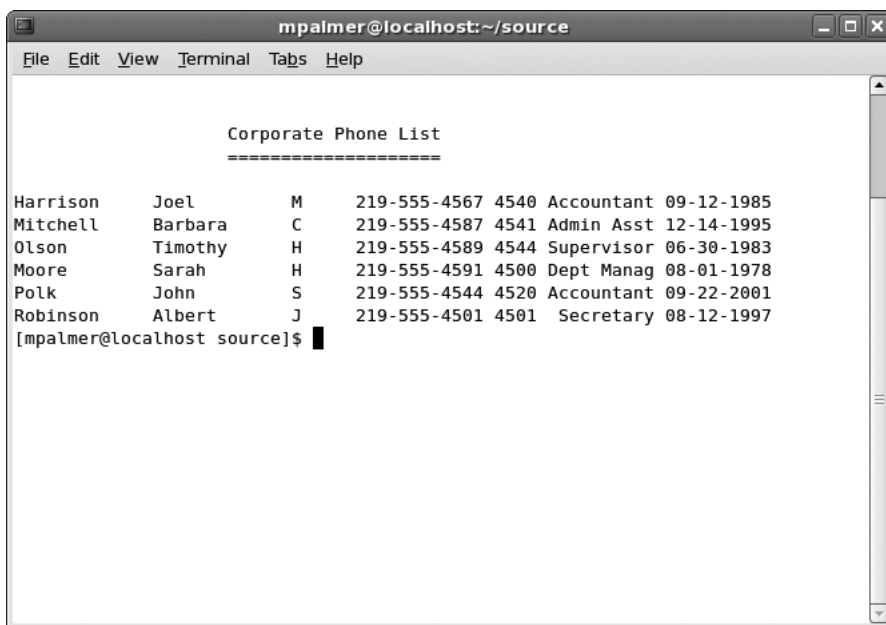
```

=====
Script Name: phlist1
By: Your initials here
Date: Today's date
Purpose: Use awk to format colon-separated fields

```

```
in a flat file and display to the screen
Command Line: phlist1
=====
clear
tput cup 2 20; echo "Corporate Phone List"
tput cup 3 20; echo "=====
tput cup 5 0;
awk -F: '{printf "%-12s %-12s %s\t%s %s %10.10s %s\n", $2, $3,
 $4, $1, $5, $6, $7}' corp_phones
```

2. Save the file and exit the editor.
3. Give the file execute permission by typing **chmod a+x phlist1** and pressing **Enter**. Run the script by typing **./phlist1** and pressing **Enter**. (See Figure 6-19.)



```
mpalmer@localhost:~/source
File Edit View Terminal Tabs Help

Corporate Phone List
=====
Harrison Joel M 219-555-4567 4540 Accountant 09-12-1985
Mitchell Barbara C 219-555-4587 4541 Admin Asst 12-14-1995
Olson Timothy H 219-555-4589 4544 Supervisor 06-30-1983
Moore Sarah H 219-555-4591 4500 Dept Manag 08-01-1978
Polk John S 219-555-4544 4520 Accountant 09-22-2001
Robinson Albert J 219-555-4501 4501 Secretary 08-12-1997
[mpalmer@localhost source]$
```

Figure 6-19 Testing the phlist1 script



## Project 6-20

Now, you need to develop a way to enter new telephone number information into the corp\_phones file so that users can eventually build a complete data file of all employees. Notice that the phmenu script offers the option to add a phone record. To accomplish this, you develop the phoneadd script.



**To create the phoneadd script for data entry:**

1. Ensure you are in the source directory and then use the vi or Emacs editor to create the script phoneadd. Enter the following code:

```
=====
Script Name: phoneadd
By: Your initials here
Date: Today's date
Purpose: A shell script that sets up a loop to add
new employees to the corp_phones file.
Command Line: phoneadd
#
=====
trap "rm ~/tmp/* 2> /dev/null; exit" 0 1 2 3
phonefile=~/source/corp_phones
looptest=y
while [$looptest = y]
do
 clear
 tput cup 1 4; echo "Corporate Phone List Additions"
 tput cup 2 4; echo "=====
 tput cup 4 4; echo "Phone Number: "
 tput cup 5 4; echo "Last Name : "
 tput cup 6 4; echo "First Name : "
 tput cup 7 4; echo "Middle Init : "
 tput cup 8 4; echo "Dept # : "
 tput cup 9 4; echo "Job Title : "
 tput cup 10 4; echo "Date Hired : "
 tput cup 12 4; echo "Add Another? (y)es or (q)uit: "
 tput cup 4 18; read phonenumber
 if ["$phonenumber" = "q"]
 then
 clear; exit
 fi
 tput cup 5 18; read lname
 tput cup 6 18; read fname
 tput cup 7 18; read midinit
 tput cup 8 18; read deptno
 tput cup 9 18; read jobtitle
 tput cup 10 18; read datehired
 # Check to see if last name is not a blank before you
 # write to disk
 if ["$lname" > " "]
 then
 echo "$phonenumber:$lname:$fname:$midinit:$deptno:$
jobtitle:$datehired" >> $phonefile
 fi
 tput cup 12 33; read looptest
 if ["$looptest" = "q"]
```

```

 then
 clear; exit
 fi
done

```

**NOTE**

Make sure you place the following statement on only one line:  
**echo "\$phonenumber:\$lname:\$fname:\$midinit:\$deptno:\$jobtitle:\$datehired"**  
**>> \$phonefile.** It is wrapped on two lines here because of space limitations on the page.

2. Save the file and exit the editor. Notice the section that checks to ensure the last name is not blank. (See the comment.) This is included so that incomplete data is not written to the file. In this code, the *if* statement checks to make certain that the *lname* variable is greater than an empty string (has contents). If *lname* does have a value (a string), the contents of all of the variables—*lname*, *fname*, *midinit*, *deptno*, *jobtitle*, *datehired*—are written to the *phonefile* variable (*corp\_phones*) using the *echo* command.
3. Give the file execute permission by typing **chmod a+x phoneadd** and pressing **Enter**.
4. Run the script by typing **./phoneadd** and pressing **Enter**.
5. Next, test the script by adding the following employees. (Press **Enter** after each separate field entry, for example, type 219-555-7175 and press **Enter** to go to the Last Name field.) See Figure 6-20.

```

219-555-7175 Mullins Allen L 7527 Sales Rep 02-19-2007
219-555-7176 Albertson Jeannette K 5547 DC Clerk 02-19-2007

```

6. Press **Ctrl+c** after you enter the data.

You've now created the foundation for the corporate employee telephone number application. However, you still need to address several deficiencies. For example, how can you return to a previous field as you enter the data? What happens when you enter the same employee twice? What happens if you assign a new employee a phone number that has already been assigned to someone else? In Chapter 7, you continue the development process and address these issues.



```
mpalmer@localhost:~/source
File Edit View Terminal Tabs Help

Corporate Phone List Additions
=====

Phone Number: 219-555-7176
Last Name : Albertson
First Name : Jeannette
Middle Init : K
Dept# : 5547
Job Title : DC Clerk
Date Hired : 02-19-2007

Add Another? (y)es or (q)uit:
```

Figure 6-20 Testing the phoneadd script

## DISCOVERY EXERCISES

1. Use two different commands to display the contents of the *HOME* variable.
2. Assign the variable *t* the value of 20. Next, assign the variable *s* the value of *t*+30. Finally, display the contents of *t* and *s* to verify you have correctly defined these variables.
3. Make the *s* variable you assigned in Exercise 2 an environment variable and use the command to verify it is recognized as an environment variable.
4. Switch to your source directory. Display the contents of the *PATH* variable. Next, use the command to add your current working directory to the *PATH* variable.
5. After completing Exercise 4, run the *phmenu* program in the easiest way.
6. Create a variable called *iam* and assign the results of the *whoami* command to it. Display the contents of the variable to verify your results.
7. Change back to your home directory, if you are not in it. Use the *set* command to set up your working environment to prevent you from overwriting a file.
8. Create an alias called *var* that displays your environment variables.
9. At the command line, use a *for* loop that uses the variable *sandwiches* and then displays a line at a time each of the following sandwiches: chicken, ham, hummus, tomato.

10. Create a script that uses case logic to have someone guess your favorite sandwich, such as tuna.
11. Display the contents of the `.bashrc` file. Next use the `vi` editor to edit that file and put in an alias so that every time you type `list`, you see a long file listing of a directory.
12. Use a command to simulate how you would troubleshoot a problem with the sandwich script you created in Exercise 10.
13. What is wrong with the following lines of code?

```
While ["$value" = "100" ; do
 Echo "That's a large number." Read value
fi
```

14. Use the `let` command to store the value 1024 in the variable `ram`. Display the contents of `ram`.
15. Temporarily change your home directory environment variable to `/home` and then use one command to go to your home directory. Change the home directory environment variable back to your regular home directory and switch to it.
16. Use the `tput` command to clear the screen and then to place the cursor in row 7, column 22.
17. Write a script that creates the following menu:

Soup Menu

=====

(t)omato

(b)ean

(s)quash

Select a soup . . . (q) to quit

18. List all of the signal numbers and designations for the `trap` command. What is the designation for signal 31?
19. Modify your script from Exercise 17 so that there is a beep or bell sound when the menu is ready to take the user's input. (Hint: review the *man* documentation for the `echo` command.)
20. Is there a command that you can use to prevent shell variables from being assigned new values? If so, what is it?