Shell Programming

This chapter concentrates on shell programming. It explains the capabilities of the shell as an interpretive high-level language. It describes shell programming constructs and particulars. It explores shell programming aspects such as variables and flow control commands. It shows the creation, debugging, and running of shell programs, and it introduces a few more shell commands.

In This Chapter

12.1 UNDERSTANDING UNIX SHELL PROGRAMMING LANGUAGE: AN INTRODUCTION

Command languages provide the means for writing programs using a sequence of commands, the same commands that you type at the prompt. Most of today's command languages provide more than just the execution of a list of commands. They have features that you find in traditional high-level languages, such as looping constructs and decision-making statements. This gives you a choice of programming in high-level languages or command languages. *Command languages* are interpreted languages, unlike *compiled languages* such as C and C++. Programs written in a command language are easier to debug and modify than programs written in compiled languages. But you pay a price for this convenience: These programs typically take much longer to execute than the compiled ones.

The UNIX shell has its own built-in programming language, and all shells (Bourne shell, Korn shell, C shell, etc.) provide you with this programming capability. The shell language is a command language with a lot of features common to many computer programming languages, including the structured language constructs: sequence, selection, and iteration. Using the shell programming language makes it easy to write, modify, and debug programs, and this language does not need compilation. You can execute a program as soon as you finish writing it.

The shell program files are called *shell procedures*, *shell scripts*, or simply *scripts*. A *shell script* is a file that contains a series of commands for the shell to execute. As you execute a shell script, each command in the script file is passed to the shell to execute, one at a time. When all the commands in the file are executed, or if an error occurs, the script ends.

You do not have to write shell programs. But as you use UNIX, you will find that sometimes you want UNIX to perform functions that it does not have a command for, or to perform several commands simultaneously. UNIX commands are numerous, difficult to remember, and involve a lot of typing. If you do not want to have to remember all that strange UNIX command syntax, or if you are not a good typist, then you may want to write script files and use them instead of complicated commands.

1. *The* **cat-n** *command is used to display most of the script (program) file examples on the screen. You can use vi to create these files. Also, you can use the* **cat** *command to create any one of these files quickly. Remember, the* **cat** *command is good for creating small files with few lines of code.*
2. *You can create these script files in the* `$HOME/Chapter12` *directory, under the appropriate subdirectories. For example, create programs in Section 12.2 in the* `$HOME/Chapter12/12.2` *directory or programs in Section 12.5 under the* `$HOME/Chapter12/12.5` *directory. Of course, you can create these files in any directory you wish and this is only a suggestion to make your directory structure and files organized according to the chapters and sections of your textbook.*

See page x for an explanation of the icons used to highlight information in this chapter.

12.1.1 Writing a Simple Script

You do not need to be a programmer to write simple script files. For example, suppose you want to know how many users are currently on the system. The command and output look like this:

```
who ¦ wc -l [Return]   . . . . . . Type the command.
6  . . . . . . . . . . . . . . . . UNIX will display the number.
```

The output of the **who** command is passed as input to the **wc** command, and the **-l** option counts the lines that indicate the number of current users on the system.

You can write a simple script file to do the same thing. Figure 12.1 shows a simple shell script called won (Who is ON) that uses the **who** and **wc** commands to report the number of people currently logged on the system. Shell scripts are stored in UNIX text files, so you use the vi editor (your favorite text editor) or the **cat** command to create them.

Figure 12.1
A Simple Shell Script

```
$ cat -n won
1 #
2 # won
3 # Display # of users currently logged in
4 #
5 who ¦ wc -l
$_
```

A # at the beginning of a line indicates that the line is a remark line and is for documentation. The shell ignores lines starting with #.

12.1.2 Executing a Script

There are two ways to execute a shell script: You can use the **sh** command, or you can make the shell script file an executable file.

Invoking Scripts

You can use the **sh** command to execute script files. Every time you type **sh** (or the name of any other shell such as **ksh** or **bash**) you invoke another copy (instance) of the shell. The shell script won is not an executable file, so you must invoke another shell to execute it. You specify the file name, and the new shell reads the script, executes the commands listed in it, and ends when all the commands have been executed (or when it reaches an error).

Figure 12.2 shows how the won file can be executed using this method. Typing **sh** (or **ksh**, **bash**, etc.) to invoke another shell each time you want to execute a script file is not very convenient. However, this method has its advantages, especially when you write complex script applications that need debugging and tracing tools. (These tools are discussed later in this chapter.) Under normal circumstances, the second method, making the script executable, is preferred. After all, the less typing, the better.

Figure 12.2
Using the **sh** Command to Run a Script

```
$ sh won
6
$_
```

As was mentioned before, you can use the name of your current shell or any other shell in your system instead of **sh**. For example, the following commands all invoke a shell script called won:

```
$ sh won [Return]  . . . . . Using sh.
$ ksh won [Return] . . . . . Using ksh.
$ bash won [Return]  . . . . . Using bash.
```

Making Files Executable: The chmod Command

The second method of executing a shell program is to make the script file an executable file. In this case, you do not need to invoke another shell; all you do is type the name of the script program, just as you do any other shell program (command). This is the preferred method.

To make a file executable, you must change the access permissions for that file. You use the **chmod** command to change the mode of a specified file. Table 12.1 shows the **chmod** options. Assuming that you have a file called `myfile`, the following example shows how you can change its access mode.

Table 12.1
chmod Command Options

Character	Who is Affected?
u	User/owner.
g	Group.
o	Others.
a	All; can be used instead of the combination of **ugo**.
Character	**Permission Category**
r	Read permission.
w	Write permission.
x	Execute permission.
-	No permission.
Operator	**Specific Action to be Taken on Permission Categories**
+	Grants permission.
-	Denies permission.
=	Sets all permissions for a specified user.

The following command makes `myfile` an executable file:

```
$ ls -l myfile [Return]  . . . . . Check myfile mode.
- rw- rw- r-- 1   david   student   64 Oct 18 15:45 myfile
$ chmod u+x myfile [Return]. . . Change myfile mode.
$ ls -l myfile [Return]  . . . . . Check myfile mode again.
- rwx rw-r-- 1    david   student   64 Oct 18 15:45 myfile
$_ . . . . . . . . . . . . . . . . . The shell prompt appears.
```

The **ls** command is used to verify the fact that its access mode is changed. The **u** indicates that the user has access to `myfile;` the **+x** indicates that the `myfile` access mode is to be changed to executable.

Assuming user, group, and others have read and write access on `myfile` file, the following command removes the write access for all other users:

```
$ chmod o-w myfile [Return]  . . . Change myfile mode.
$_ . . . . . . . . . . . . . . . . . The prompt reappears.
```

You and members of your group still have read and write access privileges, but others can only read from your file. (You can use the **ls** command to verify the changes.) The letter **o** indicates others, and **-w** indicates that the others do not have write access to `myfile`.

The following command changes the access privileges for all users (owner, group, and others) to read and write:

```
$ chmod a=rw myfile [Return] . . . Change myfile mode.
$_ . . . . . . . . . . . . . . . . . Return to prompt.
```

Now anyone can read or write to `myfile`. (Again you can use the **ls** command to verify the changes.) The letter **a** indicates all users, and **rw** indicates read and write access.

The following commands remove all access privileges for the group and other users:

```
$ ls -l myfile [Return]  . . . . . Display myfile mode.
- rwx rw- r-- 1   david   student   64 Oct 18 15:45 myfile
$ chmod go= myfile [Return]  . . . Change myfile mode. Notice that
                                   there is at least one space between
                                   "=" and "myfile."
$ ls -l myfile [Return]  . . . . . Check the mode changes.
- rwx------1      david   student   64 Oct 18 15:45 myfile
$_ . . . . . . . . . . . . . . . . . Return to the prompt.
```

The owner (in this case you) is the only one who has access to `myfile`. The **ls** command verifies the changes. The **go** command indicates group and others, so **go=** indicates removal of all access privileges to `myfile`.

Returning to your `won` shell script, let's make it an executable file:

```
$ ls -l won [Return] . . . . . Check mode set for won.
-rw- rw- r-- 1    david   student   64 Oct 18 15:45 won
$ chmod u+x won [Return] . . . . . Change mode.
$ ls -l won [Return] . . . . . Verify the change.
-rwx rw- r-- 1    david   student   64 Oct 18 16:45 won
$_ . . . . . . . . . . . . . . . . . Return to the prompt.
```

Now won, your shell script file, is an executable file, and you do not need to invoke another shell program to execute it. You execute won like any other command (executable file) by simply typing the filename and pressing **[Return]**.

```
$ won [Return]  . . . . . .  Execute won.
6  . . . . . .  The output shows there are 6 users.
$_ . . . . . .  The shell prompt appears.
```

12.2 WRITING MORE SHELL SCRIPTS

Shell programming is relatively simple, and it is a powerful tool at your disposal. You can place any command or sequence of commands in a file, make the file executable, and then execute its contents simply by typing its name at the **$** prompt.

Table 12.2 lists the shell built-in commands that are covered in this chapter and shows their availability under different shells.

Table 12.2
The Shell Built-in Commands

Command	Built into		
	Bourne Shell	Korn Shell	Bourne Again Shell
exit	sh	ksh	bash
for	sh	ksh	bash
if	sh	ksh	bash
let		ksh	bash
read	sh	ksh	bash
test	sh	ksh	bash
until	sh	ksh	bash
while	sh	ksh	bash

1. *Most UNIX systems include more than one shell. The commands and script files (programs) in this book should work with the* sh, ksh, *and* bash *shells. However, there are many variations with subtle differences in the installation. Use the* **man** *command to see how a particular command is used in your version of the shell.*
2. *You can change your current shell by typing* **sh**, **ksh**, *or* **bash** *to invoke any of the three shells, the Bourne shell, Korn shell, or Bourne Again shell. This does not change your login shell. Typing* **exit** *terminates the current shell and you are back to your login shell.*
3. *The standard variable* SHELL *is set to your login shell. You can find out what is your login shell by typing the following command line:*

```
$ echo $SHELL [Return]  . . . . . .  Display content of the SHELL
                                      variable.
```

Let's modify the won file and add a few more commands to it. Figure 12.3 shows won2, the modified version of the won script file.

Figure 12.3
A Shell Script Sample won2 Program

```
$ cat -n won2
   1 #
   2 # won2
   3 # won version 2
   4 # Displays the current date and time, # of users currently
   5 # logged in, and your current working directory
   6 #
   7 date              # displays current date
   8 who ¦ wc -l       # displays # of users logged in
   9 pwd               # diplays current working directory
$_
```

Assuming the won access mode is changed and it is an executable file, Figure 12.4 shows the output of the second version of the won program.

Figure 12.4
The Output of the won2 Program

```
$ won2
Wed Nov 30 14:00:52 EDT 2005
    14
/usr/students/david
$_
```

The output of the second version of the won program is cryptic and certainly can be improved. Let's use a few **echo** commands here and there to make the output more meaningful. Figure 12.5 shows the won script, third version, called won3.

Figure 12.5
Another Version of the won3 Program

```
$ cat -n won3
   1 #
   2 # won3
   3 # won version 3 - The user-friendly version
   4 # Displays the current date and time, # of users currently
   5 # logged in, and your current working directory
   6 #
   7 echo                  # skip a line
   8 echo "Date and Time:\c"
   9 date                  # displays current date
  10 echo "Number of users on the system:\c"
  11 who ¦ wc -l           # displays # of users logged in
  12 echo "Your current directory:\c"
  13 pwd                   # displays your current directory
  14 echo                  # skip a line
$_
```

The **echo** command with no argument is one way to output a blank line. The **echo** command ends its output with a new line.

Figure 12.6 shows the output of the won program, third version (won3). It is more informative and looks better.

Figure 12.6
The Output of the won3 Shell Program

```
$ won3
Date and Time:Wed Nov 30 15:00:52 EDT 2005
Number of users logged in:14
Your current directory:/usr/students/david
$_
```

12.2.1 Using Special Characters

As was discussed in Chapter 9, the **echo** command recognizes special characters, called *escape characters*. They all start with the backslash (\), and you can use them as part of the **echo** command argument string. These characters give you more control over the format of your output. For example, the following command produces four new lines:

```
$ echo "\n\n\n" [Return]  . . . . .  Produce four blank lines.
```

Three blank lines are produced by the three **\n** codes, and one blank line is produced by the **echo** command default next line (return key) at the end of the output string.

Table 12.3 summarizes these escape characters. The following command sequence shows examples of using the escape characters.

Table 12.3
The **echo** Command Special Characters

Escape Character	Meaning
\b	A backspace.
\c	Inhibits the default next line (**[Return]** key) at the end of the output string.
\n	A carriage return and a form feed next line (**[Return]** key).
\r	A carriage return without a line feed.
\t	A tab character.
\0*n*	A zero followed by 1-, 2-, or 3-digit octal number representing the ASCII code of a character.

If these commands do not work, make sure you are using the **-e** *option to make the* **echo** *command to recognize escape characters. For example, the previous command is typed as*

```
$ echo -e "\n\n\n"[Return]  . . . .  Using the -e option.
```

This note is applicable to the rest of the programs that use the **echo** *command with escape characters such as* **\c**, **\b**, *and so on.*

Use the *n* escape character:

```
$ echo "\nHello\n" [Return]. . . .  Use \n escape character.
      . . . . . . . . . . . . . .  A blank line is produced by the
                                   first \n.
hello. . . . . . . . . . . . . . .  The word hello.
      . . . . . . . . . . . . . .  A blank line is produced by the
                                   second \n.
      . . . . . . . . . . . . . .  A blank line is produced  by the
                                   echo command.
$_ . . . . . . . . . . . . . . . .  The prompt reappears.
```

Beeping Sound **[Ctrl-G]** produces a beeping sound on most terminals, and the ASCII code for it is the octal number 7. You can use the **\0n** format to sound a beep on your terminal.

Use the **\0***n* escape code to produce a beeping sound:

```
$ echo "\07\07WARNING" [Return]  . Use \07 for the bell sound.
WARNING
$_ . . . . . . . . . . . . . . . .  The prompt reappears.
```

You hear the terminal bell sound twice (beep beep), and the **WARNING** *message is displayed.*

Clearing Screen When you write scripts, especially interactive ones, you usually need to clear the screen before doing anything else. One way to clear the screen is to use the **\0***n* escape character format with the **echo** command to send the clear screen code to the terminal. The code for clearing the screen is terminal dependent; you may need to ask your system administrator or look it up in the terminal user/technical manual.

Assuming **[Ctrl-z]** clears your terminal screen (octal number 32), use the **echo** command to clear the screen.

```
$ echo "\032" [Return]  . . . . . .  Clear the screen.
$_ . . . . . . . . . . . . . . . . .  Return to the prompt.
```

1. *The screen is cleared, and the prompt appears at the top left corner of the screen.*
2. *If this command doesn't clear your terminal screen, it means* **[Ctrl-z]** *is not the clear screen code for your terminal. For most shells you can simply type* **clear** *and press the* **[Return]** *key to clear the screen.*

12.2.2 Logging Off in Style

Normally, you press **[Ctrl-d]** or use the **exit** command to log off and end your session. Suppose you want to change that: You want to type **bye** to log off.

Write a script file called `bye` that contains one line of code, the command **exit**:

```
$ cat bye [Return]  . . . . . .  Show contents of bye.
exit . . . . . . . . . . . . . .  One line of code.
$_ . . . . . . . . . . . . . . .  Prompt.
```

Change bye to an executable file and run it to log out:

```
$ chmod u+x bye [Return] . . . . . Change the file mode.
$ bye [Return] . . . . . . . . . . Log out.
$_ . . . . . . . . . . . . . . . . You are still in UNIX.
```

Why didn't **exit** work? When you give a command to the shell, it creates a child process to execute the command (as described in Chapter 9). Your login shell is the parent process, and your script file bye is the child process. The child process (bye) gets the control and executes the **exit** command that consequently terminates the child process (bye). When the child is dead, control goes back to the parent (your login shell). Therefore, the **$** prompt is displayed.

To make the bye script work, you must prevent the shell from creating a child process (by using the **dot** command as described in the following section), so that the shell executes your program in its own environment. Then the **exit** command terminates the current shell (your login shell), and you are logged out.

12.2.3 Executing Commands: The dot Command

The **dot** command (**.**) is a built-in shell command that lets you execute a program in the current shell and prevents the shell from creating the child process. This is useful if you want to test your script files, such as the .profile startup file. The .profile file in your home directory contains any command that you want to execute whenever you log in. You do not need to log out and log in to activate your .profile file. Using the dot command, you can execute .profile, and the commands in it are applied to the current shell, your login shell.

Returning to your bye script file, let's execute it again, this time using the **dot** command:

```
$ . bye [Return] . . . . . Use the dot command.
UNIX System V Release 4.0
login:_ . . . . . Receive login prompt.
```

As with other commands in UNIX, there is a space between the dot command and its argument (in this case, your bye program).

Let's try another version of the bye script that does not require the **dot** command so you can simply type **bye** and press **[Return]** to log off.

In Chapter 9, you learned about the **kill** command. The following command placed in the bye script file means to kill all processes, including the login shell.

```
$ cat bye [Return] . . . . . bye new version.
kill -9 0 . . . . . Place command to kill all processes.
$_ . . . . . Receive prompt.
```

Only one problem remains. You want to be able to use the bye script regardless of your current directory. If bye is in your HOME directory, you have to either change to your HOME directory every time you want to execute bye or type the full pathname to the bye file.

Path Modification You can modify the *PATH* shell variable and add your HOME directory to it. Thus your HOME directory is also searched to find the commands.

Change the *PATH* variable to add your HOME directory.

```
$ echo $PATH [Return] . . . . . Check the PATH assignment.
PATH=:/bin:/usr/bin
$ PATH=$PATH:$HOME [Return]. . . Add your HOME directory.
$ echo $PATH [Return] . . . . . Check again.
PATH=:/bin:/usr/bin:/usr/students/david
$_ . . . . . Receive the prompt.
```

Now everything is ready; type **bye** and press **[Return]** at the **$** prompt, and you are logged off.

To make your changes permanent, place the new PATH *in your* `.profile` *file. Then, any time you log in, the* PATH *variable is set to your desired path.*

Command Substitution You can place the output of a command in the argument string. The shell executes the command surrounded by a pair of grave accent marks (`) and then substitutes the output of the command in the string passed to the **echo**. For example, if you type this:

```
$ echo Your current directory: 'pwd' [Return]
```

the output looks like this:

```
Your current directory: /usr/students/david
```

In this case, **pwd** *is executed, and its output, your current directory name, is placed in a string that is passed to the* **echo** *command.*

12.2.4 Reading Inputs: The read Command

One way to assign values to variables is to use the assignment operator, the equal sign (=). You can also store values in variables by reading strings from the standard input device.

You use the **read** command to read the user input and save it in a user-defined variable. This is one of the most common uses of the user-defined variables, especially in interactive programs in which you prompt the user for information and then read the user response. When **read** is executed, the shell waits until the user enters a line of text; then it stores the entered text in one or more variables. The variables are listed after the **read** on the command line, and the end of user input is signaled when the user presses **[Return]**.

Figure 12.7 is a sample script named `kb_read` that shows how the **read** command works.

The **read** command is usually used in combination with **echo**. You use the **echo** command to prompt the user for what you want to be entered, and the **read** command waits for the user to enter something.

Invoke the `kb_read` shell script:

```
$ kb_read [Return]  . . . . . Run it.

Enter your name:. . . . . User is prompted.
david [Return]. . . . . Enter a name.
Your name is david  . . . . . Name is echoed back.

$_ . . . . . Receive prompt.
```

Figure 12.7
A Shell Script Sample: kb_read

```
$ cat -n kb_read
    1 #
    2 # kb_read
    3 # A sample program to show the read command
    4 #
    5 echo                          # skip a line
    6 echo "Enter your name:\c"     # prompt the user
    7 read name                     # read from keyboard and save it in name
    8 echo "Your name is $name"     # echo back the inputted data
    9 echo                          # skip a line
$_
```

1. *Your input string is stored in the variable name and then displayed.*
2. *It is a good idea to put the variables in quotation marks because you cannot anticipate the user input, and you do not want the shell to interpret special characters such as* **[*]**, **[?]**, *and so on.*

The **read** command reads one line from the input device. The first input word is stored in the first variable, the second word in the second variable, and so on. If your input string contains more words than the number of variables, the leftover words are stored in the last variable.

The characters assigned to the IFS *shell variable (see Chapter 9) determine the words' delimiters, in most cases the space character.*

Figure 12.8 shows a simple script, `read_test`, that reads the user response and displays it back on the screen. In this example, your input string consists of five words. The **read** command has three arguments (variables *Word1*, *Word2*, and *Rest*) to store your entire input. The first two words go to the first two variables, and the rest of the input string is stored in the third and last variable, *Rest*. If the **read** command in this example had only one parameter, say the variable *Word1*, the entire input string would have been stored in *Word1*.

Figure 12.8
A Simple Script, `read_test`, to Test the **read** Command

```
$ cat -n read_test
  1 #
  2 # read_test
  3 A sample program to test the read command
  4 #
  5 echo                                # skip a line
  6 echo "Type in a long sentence:\c" # prompt user
  7 read Word1 Word2 Rest               # read user input
  8 echo "$Word1 \n $Word2 \n $Rest"   # display the contents of the variables
  9 echo "End of my act! :-)"          # signal the end of the program
  10 echo                               # skip a line
$_
```

Let's run the `read_test` script:

```
$ read_test [Return] . . . . . Assume read_test is an
                               executable file.
Type in a long sentence: . . . Prompt is displayed.
Let's test the read command. [Return] . . . . . Show your input.
Let's . . . . . Content of $Word1.
test . . . . . Content of $Word2.
the read command . . . . . Content of $Rest.
End of my act! :-)

$_ . . . . . Receive prompt.
```

12.3 EXPLORING THE SHELL PROGRAMMING BASICS

Now that you know you can place a sequence of commands in a file and create a simple script file, let's explore the shell script as a full-powered programming language that you can use to write applications.

Like any complete programming language, the *shell* provides you with commands and constructs that help you write well-structured, readable, and maintainable programs. This section explains the syntax of these commands and constructs.

12.3.1 Comments

Documentation is important when writing computer programs, and writing a shell script program is no exception. Documentation is essential to explain the purpose and logic of the program and commands used in the program that are not obvious. Documentation is for you and anybody else who reads your program. If you look at a program a few weeks after you have written it, you will be surprised at how much of your code you do not remember.

The shell recognizes **#** *as the comment symbol; therefore, characters after the* **#** *are ignored.*

The following command sequences show examples of a comment line:

```
#  . . . . . This is a comment.
# program version 3  . . . . . This is also a comment line.
date   # show the current date. . . . . This is a comment that begins in
                                         the middle of the line.
```

12.3.2 Variables

Like other programming languages, the UNIX shell lets you create variables to store values in them. To store a value in a variable, you simply write the name of the variable, followed by the equal sign and the value you want to store in the variable, like this:

```
variable=value
```

Spaces are not permitted on either side of the equal sign.

The UNIX shell does not support data types (integer, character, floating point, etc.) as some programming languages do. It interprets any value assigned to a variable as a string of characters. For example, **count=1** means store character *1* in the variable called *count.*

The variable names follow the same syntax and rules that apply to file names as described in Chapter 5. Briefly, to refresh your memory, names must begin with a letter or an underscore character (_); you can use letters and numbers for the rest of the name.

The shell is an interpretive language, and the commands or variables you place in your script file can be directly typed at the **$** prompt. Of course, this method is a one-time process; if you want to repeat the sequence of commands, you have to type them again.

The following examples are valid variable assignments, and they remain in effect until you change them or log off:

```
$ count=1 [Return] . . . . . Assign character 1 to count.
$ header="The Main Menu" [Return] . . Store the string in header.
$ BUG=insect [Return] . . . . . Make another variable
                                assignment.
```

If your value string contains embedded white spaces, then you must place it between quotation marks.

Variables in your shell script stay in the memory until the shell script is finished/terminated, or you can erase the variables by using the **unset** command. To do this, you type **unset**, specify the name of the variable you want to erase, and press **[Return]**. For example, the following command erases the variable called *XYZ*:

```
$ unset XYZ [Return]
```

Displaying Variables

You know from Chapter 9 that the **echo** command can be used to display the contents of variables. The format is as follows:

```
echo $variable
```

Let's show the contents of the three variables assigned in the previous examples:

```
$ echo $count $header $BUG [Return]   . Show the values stored in the
                                        variables.
1 The Main Menu insect
$_  . . . . . .The shell prompt appears.
```

Command Substitution

You can store the output of a command in a variable. You enclose a command in a pair of grave accent marks (`), and the shell executes the command and replaces the command with its output, like this:

```
$ DATE=`date` [Return] . . . . . Save the output of the date
                                 command in the variable DATE.
$ echo $DATE [Return] . . . . . Check what is stored in DATE.
Wed Nov 29 14:30:52 EDT 2001
$_. . . . . The shell prompt appears.
```

The output of the **date** *command is stored in the* DATE *variable, and the* **echo** *command is used to display* DATE.

12.3.3 The Command Line Parameters

The shell scripts can read up to 10 command line parameters (also called *arguments*) from the command line into the special variables (also called *positional variables*, or *parameters*). *Command line arguments* are the items you type after the command, usually separated by spaces. These arguments are passed to the program and change the behavior of the program or make it act in a specific order. The *special variables* are numbered in sequence from 0 to 9 (no number 10) and are named *$0*, *$1*, *$2*, and so on. Table 12.4 shows the list of the special (positional) variables.

Table 12.4
The Shell Positional Variables

Variable	Meaning
$0	Contains the name of the script, as typed on the command line.
$1, $2,... $9	Contains the first through ninth command line parameters, respectively.
$#	Contains the number of command line parameters.
$@	Contains all command line parameters: "$1 $2 . . . $9".
$?	Contains the exit status of the last command.
$*	Contains all command line parameters: "$1 $2 . . . $9".
$$	Contains the PID (process ID) number of the executing process.

Using Special Shell Variables

Let's look at examples to understand how these special shell variables are used and arranged. Suppose you have a shell script named BOX, whose mode you have changed to an executable file, using the **chmod** command. Figure 12.9 shows your BOX script file.

1. The special variable *$0* always holds the name of the script typed on the command line.
2. Special variables *$1*, *$2*, *$3*, *$4*, *$5*, *$6*, *$7*, *$8*, and *$9* hold arguments 1 through 9, respectively. Command line arguments after the ninth parameter are ignored.
3. The special variable *$** holds all of the command line arguments passed to the program in a single string. It can store more than nine parameters.
4. The special variable *$@* holds all of the command line arguments, just like the $*. However, it stores them with quotation marks around each command line argument.
5. The special variable *$#* holds the count of the command line arguments that are typed on the command line.
6. The special variable *$?* holds the exit status received from the **exit** command in your script file. If you have not coded an **exit**, then it holds the status of the last non-background command executed in your script file.
7. The special variable *$$* holds the process ID of the current process.

Figure 12.9
A Shell Script File Named BOX

```
$ cat -n BOX
   1 #
   2 # BOX
   3 # A sample program to show the shell variables
   4 #
   5 echo                    # skip a line
   6 echo "The following is output of the $0 script: "
   7 echo "Total number of command line arguments: $#"
   8 echo "The first parameter is: $1 "
   9 echo "The second parameter is: $2"
  10 echo "This is the list of all is parameters: $* "
  11 echo                    # skip a line
$_
```

The following command sequences show different invocations of the BOX program (with or without the command line arguments), and the output of each run is investigated.

Invoke BOX with no command line arguments; just type the name of the file:

```
$ BOX [Return]  . . . . . .
```
No command line arguments are given.

```
The following is output of the BOX script:
Total number of command line arguments: 0
The first parameter is:
The second parameter is:
This is the list of all parameters:

$_   . . . . . .
```
Receive prompt.

The variable *$0* holds the name of the script that is invoked. In this case, BOX is stored in *$0*. There are no command line arguments, so the *$#* holds *$0*, and no value is stored in variables *$1*, *$2*, and *$**. The **echo** command shows the prompt strings.

Invoke BOX with two command line arguments.

```
$ BOX IS EMPTY [Return]. . .
```
Command line has two arguments.

```
The following is output of the BOX script:
Total number of command line arguments: 2
The first parameter is: IS
The second parameter is: EMPTY
This is the list of all parameters: IS EMPTY

$_ . . . . . .
```
Receive prompt.

The script name is stored in *$0*; in this case, BOX is stored in *$0*. The first argument is stored in special variable *$1*, the second argument in *$2*, and so on. In this case, **IS** is the first command line argument (stored in *$1*), and **EMPTY** is the second one (stored in *$2*). The *$** holds the list of all the command line arguments. In this case, it holds **IS EMPTY**.

If a script with more than nine command line arguments is invoked, any argument after the ninth one is ignored. However, you can capture all of them when you use the special variable *$**.

Assigning Values

Another way to assign values to the positional variables is to use the **set** command. What you type as the **set** command arguments are assigned to the positional variables.

Examine the following examples:

```
$ set One Two Three [Return]  . .Assign three arguments.
$ echo $1 $2 $3 [Return] . . . .Display the values assigned to $1–$3.
One  Two  Three
$ set 'date' [Return]  . . . . .This is another example.
$ echo $1 $2 $3 [Return] . . . .Display the positional variables.
Wed Nov 29
$_ . . . . . .Back to the prompt.
```

The **date** command is executed, and its output becomes the argument to the **set** command. If the output of the **date** command is

```
Wed Nov 30  14:00:52 EDT 2005
```

then the **set** command has six arguments (the space character is the delimiter), and the *$1* variable holds *Wed* (the first argument), *$2* holds *Nov* (the second argument), and so on.

Scenario Now let's write a script file and make use of these positional variables. Suppose you want to write a script file that stores the specified file in a directory called `keep` in your `HOME` directory, and then invoke the vi editor to edit the specified file. The commands to do this job are

```
$ cp xyz $HOME/keep [Return]  . .  Copy the specified file xyz to keep.
$ vi xyz [Return] . . . . . .  Invoke the vi editor.
```

Figure 12.10 shows a script file called `svi` that also does the job. Commands in the `svi` file are the same commands you type at the prompt. The positional variables are used to make `svi` a versatile program. It saves and edits any specified file passed to it on the command line argument.

If `svi` works, the `xyz` file is saved in the `keep` directory, the vi editor is invoked, and you are able to edit the `xyz` file. You can use the **ls** command to check if a copy of the `xyz` file is saved in `keep`.

Figure 12.10
The `svi` Script File

```
$ cat -n svi
   1 #
   2 # svi
   3 # save and invoke vi
   4 # A sample program to show the usage of the shell variables
   5 #
   6 DIR=$HOME/keep       # assign keep pathname to DIR
   7 cp $1 $DIR           # copy specified file to keep
   8 vi $1                # invoke vi with the specified filename
   9 exit 0               # end of the program, exit
$_
```

Scenario Suppose the `svi` mode is changed to make it an executable file, and you have a file called `xyz` in your current directory that you want to edit.

Let's run `svi` and look at the results:

```
$ svi xyz [Return]  . . . . . .  Run svi with a specified file
                                 name.
hello. . . . . .  You are in the vi editor, and the
                  contents of xyz are displayed.
~ . . . . . .  Show the rest of the vi screen.
"xyz" 1 Line, 5 characters . . . .  The vi status line.
$ . . . . . .  The prompt is displayed after
               you exit vi.
```

What if you do not have the `xyz` file in your current directory? How does the `svi` program work if you don't specify a file name? The `svi` program will work, but it may not be exactly the way you would expect. In the first case, the **cp** command fails to find the `xyz` file in your current directory, and the vi editor is invoked with the `xyz` as a new file. In the second case, the **cp** command fails again, and the vi editor is invoked with no file name. In both cases, you do not see the **cp** command's error messages, as vi is invoked before you have a chance to see them.

The `svi` program needs fixing. It must be able to recognize errors, show the appropriate error messages, and not invoke **cp** or vi if the specified file is not in your directory or the filename is not specified on the command line.

Before modifying the `svi` code to handle these problems, you need to know some other commands and constructs of the shell language.

Terminating Programs: The exit Command

The **exit** command is a built-in shell command that you can use to immediately terminate execution of your shell program. The format of this command is as follows:

```
exit n
```

where *n* is the exit status, which is also called the return code (RC). If no exit value is provided, the exit value of the last command executed by the shell is used. To be consistent with the other UNIX programs (commands) that usually return a status code upon completion, you can program your script to return an exit status to the parent process. As you know, typing **exit** at the **$** prompt terminates your login shell and consequently logs you off.

12.3.4 Conditions and Tests

You can specify that certain commands be executed depending on the results of the other commands' executions. You often need this kind of control when writing shell scripts.

Every command executed in the UNIX system returns a number that you can test to control the flow of your program. A command returns either a 0 (zero), which means success, or some other number, which indicates failure. These true and false conditions are used in the shell programming constructs and are tested to determine the flow of the program. Let's look at these constructs.

The if-then construct

The **if** statement provides a mechanism to check whether the condition of something is true or false. Depending on the results of the **if** test, you may change the sequence of command executions in your program. The following shows the **if** statement (construct) syntax:

```
if [ condition ]
then
  commands
  ...
  last-command
fi
```

1. *The* **if** *statement ends with the reserved word* **fi** *(***if** *typed backward).*
2. *The indentation is not necessary, but it certainly makes the code more readable.*
3. *If the condition is true, then all the commands between the* **then** and **fi**, *called the body of the* **if**, *are executed. If the condition is false, the body of the* **if** *is skipped, and the line after* **fi** *is executed.*

The square brackets around the conditions are necessary and must be surrounded by white spaces, either spaces or tabs.

Modify the `svi` script file to include an **if** statement to match Figure 12.11 (`svi2`). The **if** statement is putting some control over the `svi2` output.

Figure 12.11

Another Version: The `svi2` Script File

```
$ cat -n svi2
   1 # svi2
   2 # save and invoke vi
   3 # A sample program to show the usage of the shell variables
   4 # Version 2: adding the if-then statement
   5 #
   6 DIR=$HOME/keep          # assign keep pathname to DIR
   7 if [ $# = 1 ]           # check for number of command line arguments
   8 then
   9   cp $1 $HOME/keep      # copy specified file to keep
  10 fi                      # end of the if statement
  11 vi $1                   # invoke vi with the specified file name
  12 exit 0                  # end of the program, exit
$_
```

If the count of the command line arguments (**$#**) is one (a filename is specified), then the condition is true and the body of the **if** (the **cp** command) is executed, followed by the vi invocation with the specified file name, exactly as the last `svi` program run.

If the count of the command line arguments (**$#**) is zero (no filename is specified on the command line), then the condition is false, the body of the **if** (the **cp** command) is skipped, and only the vi editor is invoked with no file name.

You can run the `svi2` program by typing the following command line:

```
$ svi2 [Return] . . . . . .  Run svi without an argument.
$ svi2 myfirst [Return] . . .  Run svi2 with a file name.
```

The if-then-else Construct

By adding the **else** clause to the **if** construct, you can execute certain commands when the test of the condition returns a false status. The syntax of this more complex **if** construct is as follows:

```
if [ condition ]
then
  true-commands
  ...
  last-true-command
else
  false-commands
  ...
  last-false-command
fi
```

1. *If the condition is true, then all the commands between the* **then** *and* **else**, *the body of the* **if**, *are executed.*
2. *If the condition is false, the body of the* **if** *is skipped and all the commands between* **else** *and* **fi**, *the body of the* **else**, *are executed.*

Modify the `svi2` script file to include the **if** and **else** statements. Figure 12.12 shows this modification, called `svi3`.

Figure 12.12
Another Version: The **svi3** Script File

```
$ cat -n svi3
   1 #
   2 # svi3
   3 # save and invoke vi
   4 # A sample program to show the usage of the shell variables
   5 # Version 3: adding the if-then-else statement
   6 #
   7 # DIR=$HOME/keep     # assign keep pathname to DIR
   8 if  [ $# = 1 ]       # check for the number of command line arguments
   9 then
  10   cp $1 $DIR         # copy specified file to keep
  11   vi $1              # invoke vi with the specified file name
  12 else
  13   echo "You must specify a file name. Try again."  #display error message
  14 fi                   # end of the if statement
  15 exit 0               #end of the program, exit
$_
```

If the count of the command line arguments **($#)** is one (a filename is specified), then the condition is true and the body of the **if** (the **cp** and **vi** commands) is executed. Next, the body of the **else** is skipped, and the **exit** command is executed, exactly as in the previous `svi2` run.

If the count of the command line arguments **($#)** is not one, then the condition is false, the body of the **if** (the **cp** and **vi** commands) is skipped, and the body of the **else** (the **echo** command) is executed. The **echo** command shows the error message, and then the **exit** command is executed.

Let's run this new version of the program `svi3`.

```
$ svi3 [Return] . . . . . .  No command line argument.
You must specify a filename. Try again.
$_ . . . . . .  Receive prompt.
```

1. *You must specify a filename on the command line; otherwise, you get the error message and the vi editor is not invoked.*
2. *If you do not specify a filename, then the number of arguments (the value of the positional variable* **$#***) is zero. Therefore, the* **if** *condition fails and the body of the* **if** *is skipped; the body of the* **else,** *the* **echo** *command, is executed.*

In this version, the `svi3` script does not check for the file's existence. If the specified filename is not in the specified directory, the copy command is going to complain, but the vi editor is invoked with the specified filename as a name for a new file.

The if-then-elif Construct

When you have nested sets of **if** and **else** constructs in a script file, you can use the **elif** (short for **else if**) statement. The **elif** combines the **else** and **if** statements. The complete syntax is as follows:

```
if [condition_1]
then
     commands_1
elif [condition_2]
then
     commands_2
elif [condition_3]
then
     commands_3
...
...
else
     commands_n
fi
```

The script file in Figure 12.13, called `greetings1`, outputs greetings according to the time of the day. It shows the **Good Morning** message before noon, **Good Afternoon** if the time is between 12:00 and 18:00, and so on. The **if-then-elif** construct is used to determine the morning, afternoon, and evening hours.

1. *The* **set** *command is used with* **date** *as its argument to set the positional variables.*
2. *The* **hour:minute:second** *is the fourth field in the date and time string (the output of the* **date** *command), and it is assigned to the positional variable* $4.

Figure 12.13
A Sample Script Using the **if-then-elif** Construct: `greetings1`

```
$ cat -n greetings1
   1 #
   2 # greetings
   3 # greetings program version 1
   4 # A sample program using the if-then-elif construct
   5 # This program displays greetings according to the time of the day
   6 #
   7 echo
   8 set `date`         # set the positional variables to the date string
   9 hour=$4            # store the part of the date string that shows the hour
  10 if [ "$hour" -lt 12 ]              # check for the morning hours
  11 then
  12    echo "GOOD MORNING"
  13 elif [ "$hour" -lt 18 ]            # check for the afternoon hours
  14 then
  15    echo "GOOD AFTERNOON"
  16 else                               # it must be evening
  17    echo "GOOD EVENING"
  18 fi                                 # end of the if statement
  19 echo
  20 exit                               # end of the program, exit
$_
```

3. *For this version of the* `greetings1` *script to work, you must have the standard shell (*sh*). If your login shell is* ksh *or* bash, *type* **sh** *and* **[Return]** *to invoke a copy of the* sh *shell and then execute the* `greetings1` *file. You can type* **exit** *to return to your login shell.*

To run this program, you type the following command line. Notice that it is assumed that you have already made `greetings1` an executable file.

```
$ greetings1 [Return] . . . . . .
```
Run `greetings1`.

You can write the `greetings` program in many ways depending on the shell commands you know and want to use. For example, instead of using the **set** command and the positional variables, you could use the **date** command capabilities and obtain only the hour field from its output string.

Figure 12.14 shows another version of the `greetings` program, which we will call `greetings2`.

The line **hour=`date +%H`** requires more explanation. The **%H** limits the output of the **date** command to just the part of the date string that shows the hour.

Use the **date** command capabilities:

```
$ date [Return] . . . . . .
```
Display date and time.
```
Wed Nov 30  14:00:52 EDT 2005
$ date +%H [Return] . . . . . .
```
Show only the hour of the day.
```
14
$_ . . . . . .
```
The prompt reappears.

Figure 12.14
A Sample Script Using the **if-then-elif** Construct: `greetings2`

```
$ cat -n greetings2
  1 #
  2 # greetings2
  3 # greetings program version 2
  4 # A sample program using the if-then-elif construct
  5 # This program displays greetings according to the time of the day
  6 # Version2: using the date command format control option
  7 #
  8 echo                    # skip a line
  9 hour=`date +%H`         # store the part of the date string that shows
                              the hour
 10 if [ "$hour" -le 18 ]   # check for the morning hours
 11 then
 12   echo "GOOD MORNING"
 13 elif [ "$hour" -le 18 ] # check for the afternoon hours
 14 then
 15   echo "GOOD AFTERNOON"
 16 else                    # it must be evening
 17   echo "GOOD EVENING"
 18 fi
 19 echo                    # skip a line
 20 exit 0                  # end of the program, exit
$_
```

The **date** command has numerous field descriptors that let you limit or format its output. You type **+** at the beginning of the argument followed by field descriptors such as **%H**, **%M**, and so on. You can use the **man** command to obtain a full list of the field descriptors. Let's look at more examples.

Use the **date** command field descriptors:

```
$ date '+DATE: %m-%d-%y' [Return] . . . . . .
```
Show only the date separated by hyphens.

```
DATE: 05-10-99
$ date '+TIME: %H:%M:%S' [Return] . . . . . .
```
Show only the time separated by colons.

```
TIME: 16:10:52
```

1. *The argument must start with a plus sign. It indicates that the output format is under your control. Each field descriptor is preceded by a percent sign (%).*
2. *If the argument contains white-space characters, then it must be put inside quotation marks.*
3. *You can add the* `greetings2` *program to your* `.profile` *file, so that every time you log in, the appropriate greeting message is displayed.*

True or False: The test Command

The **test** command is a built-in shell command that evaluates the expression given to it as an argument and returns true if the expression is true (if it returns 0) or, if otherwise, false (if it returns nonzero). The expression could be simple, such as testing two

numbers for equality, or complex, like testing several commands that are related with logical operators. The **test** command is particularly helpful in writing script files. In fact, the brackets around the condition in the **if** statement are a special form of the **test** command. Figure 12.15 shows a sample script file using the **test** command to test the **if** condition.

Figure 12.15
A Shell Script File Named `check_test`

```
$ cat -n check_test
   1 #
   2 # check_test
   3 # A sample program using the test command
   4 #
   5 echo                                    # skip a line
   6 echo "Are you OK?"
   7 echo "Input Y for yes and N for no:\c" # prompt user
   8 read answer                            # store user response in answer
   9 if test "$answer" = Y                  # test if user entered Y
  10 then
  11   echo "Glad to hear that!"            # display the message
  12 else                                   # user entered N
  13  echo "Go home!"                       # display the message
  14 fi
  15 echo                                   # skip a line
  16 exit 0                                 # end of the program, exit
$_
```

1. *The* `check_test` *script prompts the user to enter* **(Y)***es or* **(N)***o, and then reads the user answer.*
2. *If the user response is the letter* **Y**, *the* **test** *command returns zero, and the* **if** *condition is true, then the body of the* **if** *is executed. The message* **Glad to hear that!** *is displayed, and the body of the* **else** *is skipped.*
3. *Any input from the user other than a single letter* **Y** *makes the* **if** *condition fail; therefore, the body of the* **if** *is skipped and the* **else** *is executed. The message* **Go home!** *is displayed.*

Invoking test with Brackets The shell offers you another way to invoke the **test** command. You can use square brackets ([and]) instead of the word **test**. So the **if** statement can be written as follows:

```
    if test "$variable" = value
or
    if [ "$variable" = value ]
```

12.3.5 Testing Different Categories

Using the **test** command, you can test different categories of things including the following: numeric values, string values, and files. Each of these categories is explained in the following sections.

Numeric Values

You can use the **test** command to test (compare) two integer numbers algebraically. You also can combine expressions comparing numbers with logical operators. The format is as follows:

```
test expression_1 logical operator expression_2
```

The logical operators are as follows:

- Logical *and* operator (**-a**): The **test** command returns 0 (condition code true) if both expressions are true.
- Logical *or* operator (**-o**): The **test** command returns 0 (condition code true) if one or both of the expressions are true.
- Logical *not* operator (**!**): The **test** command returns 0 (condition code true) if the expression is false.

The operators available for comparing variables that hold numeric values are summarized in Table 12.5.

Table 12.5
The **test** Command Numeric Test Operators

Operator	Example	Meaning
-eq	***number1* -eq*number2***	Is *number1* equal to *number2*?
-ne	***number1* -ne*number2***	Is *number1* not equal to *number2*?
-gt	***number1* -gt*number2***	Is *number1* greater than *number2*?
-ge	***number1* -ge*number2***	Is *number1* greater than or equal to *number2*?
-lt	***number1* -lt*number2***	Is *number1* less than *number2*?
-le	***number1* -le*number2***	Is *number1* less than or equal to *number2*?

Scenario Suppose you want to write a script that accepts three numbers as input (command line arguments) and displays the largest of the three. Using the **cat** command, Figure 12.16 shows one way of writing this program, which is named `largest`.

1. *When you run the* `largest` *program, it waits for you to enter three numbers. Then your input is passed through the* **if-then-elif** *construct to find the largest number.*
2. *If neither of the first two numbers you enter is the largest, then the first* **if** *statement fails; next, the* **elif** *also fails, and your program gets to the* **else** *statement. There is no need for more checking. If the largest number is neither the first nor the second number, then it must be the third one.*

Try a sample run of the `largest` program:

```
$ chmod +x largest [Return]  . . .  Change it into an executable file.
$ largest [Return]    . . . . . .  Run the program.
Enter three numbers and I will show you the largest of them>>_
100 10 400 [Return] . . . . . .  Enter three numbers.
The largest number is: 400
Done! :-)
$_ . . . . . .  The prompt reappears.
```

Figure 12.16
A Shell Program File Named `largest`

```
$ cat -n largest
   1 #
   2 # largest
   3 # A sample program using the test command
   4 # This program accepts three numbers and shows the largest of them
   5 #
   6 echo                       # skip a line
   7 echo "Enter three numbers and I will show you the largest of them>> \c"
   8 read num1 num2 num3
   9 if test "$num1" -gt "$num2" -a "$num1" -gt "$num3"
  10 then
  11    echo "The largest number is: $num1"
  12 elif test "$num2" -gt "$num1" -a "$num2" -gt "$num3"
  13 then
  14    echo "The largest number is:$num2"
  15 else
  16    echo "The largest number is:$num3"
  17 fi
  18 echo "Done! :-)"          # end of the program message
  19 echo                      # skip a line
  20 exit 0                    # end of the program, exit
$_
```

Think about how you can improve the `largest` program. For example, could you use fewer lines of code? (e.g., Do you have to repeat the **echo** command in the body of the **if**, **elif**, and **else**?) Could you do error checking? (What if you enter two numbers?)

String Values

You can also compare (test) strings with the **test** command. The **test** command provides a different set of operators for the string comparison. These operators are summarized in Table 12.6. The following examples show the use of the **test** command with string arguments.

Table 12.6
The **test** Command String Test Operators

Operator	Example	Meaning
=	***string1* = *string2***	Does *string1* match *string2*?
!=	***string1* != *string2***	Does *string1* not match *string2*?
-n	**-n *string***	Does *string* contain characters (nonzero length)?
-z	**-z *string***	Is *string* an empty string (zero length)?

First experiment with the null variable:

```
$ STRING5 [Return]. . . . . .  Declare a null variable.
$ test -z $STRING [Return] . . . . .  Test for zero length.
test: Argument expected . . . . . .  Error message.
$_ . . . . . .  The prompt reappears.
```

1. *The shell substitutes the* null *string for* $STRING*, thus the error message.*
2. *Enclosing a string variable in quotation marks ensures a proper test, even if the variable contains spaces or tabs.*

Now try the variable *$STRING* in quotation marks:

```
$ test -z "$STRING" [Return] . . . .  Test for zero length.
```

This time the **test** *command returns zero (true), which means the variable* $STRING *contains a zero length string.*

In another example, the commands are typed directly at the $ prompt. If the command is not complete, the shell shows the secondary prompt (>) and waits for you to complete the command:

```
$ DATE1= `date` [Return] . . . . . .  Initialize DATE1.
$ DATE2= `date` [Return] . . . . . .  Initialize DATE2.
$ if test "$DATE1" = "$DATE2" [Return] . . . . . .  Test for equality.
> then [Return] . . . . . .  The shell shows the second
                             prompt sign since the if
                             command is not
                             completed yet.
> echo "STOP! The computer clock is dead!" [Return]
> else [Return] . . . . . .  Begin the else body.
> echo "Everything is fine." [Return]
> fi [Return] . . . . . .  End of if; as soon as you press
                           [Return], the program is
                           executed.
Everything is fine . . . . . .  Get the output of the
                                program.
$_ . . . . . .  The prompt reappears.
```

*The result of this test (***if** *condition) is false. Therefore, the body of the* **else** *is executed, and the message is displayed. Can the values stored in* DATE1 *and* DATE2 *ever be equal?*

Files

You can use the **test** command to test file characteristics, such as file size, file type, and file permissions. More than 10 file attributes can be checked; a few of them are introduced here. Table 12.7 summarizes the file test operators. The following example shows the use of the **test** command in connection with files. The commands are typed directly at the **$** prompt.

Table 12.7
The **test** Command File Test Operators

Operator	Example	Meaning
-r	**-r** ***filename***	Does *filename* exist, and is it readable?
-w	**-w** ***filename***	Does *filename* exist, and is it writable?
-s	**-s** ***filename***	Does *filename* exist, and does it have a nonzero length?
-f	**-f** ***filename***	Does *filename* exist, but is it not a directory file?
-d	**-d** ***filename***	Does *filename* exist, and is it a directory file?

Suppose you have a file called `myfile`, with only read access to it. Let's type the following commands and see the output:

```
$ FILE=myfile [Return] . . . . . Initialize FILE variable.
$ If test -r "$FILE" [Return] . . . . . Check to see if myfile is readable.
> then [Return] . . . . . Secondary prompt.
> echo "READABLE" [Return] . . . . . Show message.
> elif test -w "$FILE" [Return] . . . . . Check to see if myfile is writable.
> then [Return]
> echo "WRITABLE" [Return] . . . . . Show message.
> else [Return]
> echo "Read and Write Access Denied" [Return]
> fi [Return] . . . . . End of the if construct.
READABLE
$_ . . . . . Return back to the primary prompt.
```

1. *As soon as you type* **fi**, *the end of the* **if** *construct, the shell executes the commands, produces the output, and shows the* **$** *prompt.*
2. *The first* **if** *tests the read access to* `myfile`. *Since you have read access, the* **test** *command returns 0 (true), and the* **if** *condition is true. So only the body of the first* **if** *is executed, and the message* **READABLE** *is displayed.*
3. *The message* **WRITABLE** *is echoed if you have only write access to* `myfile`.
4. *The message* **Read and Write Access Denied** *is displayed if you do not have write or read access to* `myfile` *and the* **if** *condition and* **elif** *condition fail.*

Figure 12.17 shows yet another version of the `svi` script file. In this version (`svi4`), the existence of the specified file is checked, and only if the file exists will the **cp** and **vi** commands be executed.

Run this new version of the `svi` program `svi4`:

```
$ svi4 xyz [Return] . . . . . Run it, specifying a nonexistent file xyz.
File not found. Try again.
$ svi4 [Return] . . . . . Run it again; no file is specified.
You must specify a filename. Try again.
$_ . . . . . Return to the prompt.
```

Figure 12.17
Another Version of the `svi` Script File: `svi4`

```
$ cat -n svi4
   1 #
   2 # svi4
   3 # save and invoke vi
   4 # A sample program to use the test command and file test operators
   5 # Version 4: This version checks for the existence of the file
   6 #
   7 DIR=$HOME/keep    # assign keep pathname to DIR
   8 if test $# = 1    # check for number of command line argument
   9 then
  10  if test -f $1    # check for the existence of the file
  11  then
  12    cp $1 $DIR     # copy specified file to keep
  13    vi $1          # invoke vi with the specified filename
  14  else
  15    echo "File not found. Try again"
  16  fi
  17 else
  18   echo " You must specify a filename. Try again."
  19 fi                # end of the if statement
  20 exit 0           # end of the program, exit
$_
```

1. *To check the* **if** *conditions, the word* test *is used instead of brackets.*
2. *This program uses nested* **if** *constructs (*`if` *inside another* `if`*).*
3. *If the specified file is not found, then it shows the* **File not found. Try again.** *message.*
4. *If the specified file is found, then it copies the file to the* `keep` *directory and invokes the vi editor.*
5. *If no filename is specified on the command line, the first* **if** *condition fails, and it shows the* **You must specify a filename. Try again.** *message.*

12.3.6 Parameter Substitution

The shell provides a parameter substitution facility that lets you test the value of a parameter and change its value according to a specified option. This is useful in shell programming when you want to check if a variable is set to something. For example, when you issue a **read** command in a script file, you want to make sure the user has entered something before taking any action.

The format consists of a dollar sign (**$**), a set of braces ({ and }), a variable, a colon (**:**), an option character, and a word, as follows:

```
${parameter:option character word}
```

An option character determines what you intend to do with the *word*. The four option characters are specified by the **-**, **+**, **?**, and **=** signs. These four options perform differently, depending on whether the variable is an empty variable.

A variable is an *empty variable* (a *null variable*) if its value is an empty string. For example, all of the following variables are set to a null value and are empty variables.

```
EMPTY=  . . . . . . . . . . . . This is an empty variable called EMPTY.
EMPTY=""  . . . . . . . . . . . This is an empty variable called EMPTY.
EMPTY=''  . . . . . . . . . . . This is an empty variable called EMPTY.
```

To create an empty variable, put no blank characters between the quotation marks.

Table 12.8 summarizes the shell variable substitution (evaluation) options. A brief explanation and example of each type follow.

Table 12.8
The Shell Variable Evaluation Options

Variable Option	Meaning
$variable	The value stored in *variable*.
${variable}	The value stored in *variable*.
${variable: -string}	The value of *variable* if it is set and not empty; otherwise, the value of *string*.
${variable: +string}	The value of *string* if *variable* is set and not empty; otherwise, nothing.
${variable: =string}	The value of *variable* if it is set and not empty; otherwise, *variable* is set to the value of *string*.
${variable: ?string}	The value of *variable* if it is set and not empty; otherwise, print the value of *string* and exit.

${parameter} Placing the variable (parameter) inside the braces prevents any conflict caused by the character that follows the variable name. The following example clarifies this matter.

Suppose you want to change the name of a file called `memo`, specified in the variable called *FILE,* to `memoX`.

```
$ echo $FILE [Return] . . . . . . . Check what is stored in FILE.
memo
$ mv $FILE $FILEX [Return] . . . . Change memo to memoX.
Usage: mv
$_ . . . . . . . . . . . . . . . . . . Return to prompt.
```

This command did not work because the shell considers *$FILEX* the name of the variable, which does not exist.

```
$ mv $FILE ${FILE}X [Return] . . . Change memo to memoX.
$_ . . . . . . . . . . . . . . . . . Job done; the prompt is back.
```

This command worked because the shell considers *$FILE* the variable name and substitutes its value, in this case `memo`.

${parameter:-*word*} The **-** (hyphen) option means if the listed variable (parameter) is set and not empty (nonnull), use its value. Otherwise, if the variable is empty (null) or unset, substitute its value with *word*. For example:

```
$ FILE= [Return] . . . . . . . . . . . . . . . . . . . . . . . This is an empty variable.
$ echo ${FILE:-/usr/david/xfile} [Return] . . . . . . . . . . Display FILE value.
/usr/david/xfile
$ echo"$FILE" [Return] . . . . . . . . . . . . . . . . . . . . Check the FILE variable.
   . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . It remains empty.
$_ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Return to prompt.
```

1. *The shell evaluates the variable* FILE; *it is empty, so the* **-** *option results in substitution of the* `/usr/david/xfile` *string, which is passed to the* **echo** *command to display it.*
2. *The* FILE *variable remains an empty variable.*

${parameter:+*word*} The **+** option is the opposite of the **-** option. It means that if the listed variable (parameter) is set and is not empty (nonnull), then substitute its value with *word*. Otherwise, the variable's value remains the same. For example:

```
$ HELP="wanted" [Return] . . . . . . . . Set the HELP variable
$ echo ${HELP:+"Help is on the way"} [Return]
Help is on the way
$ echo $HELP [Return] . . . . . . . . . . Check the HELP variable.
wanted . . . . . . . . . . . . . . . . . . It remains the same.
$_ . . . . . . . . . . . . . . . . . . . . Return to prompt.
```

1. *The shell evaluates the* HELP *variable; it is set to* wanted. *So the* **+** *option results in the substitution of the* **Help is on the way** *string, which is passed to the* **echo** *command to display it.*
2. *The value stored in the* HELP *variable remains the same.*

${parameter:=*word*} The **=** option means if the listed variable (parameter) is not set or it is empty (null), then substitute its value with *word*. Otherwise, if the variable is not empty, then its value remains the same. For example:

```
$ MESG= [Return]. . . . . . . . . . . . . Empty variable.
$ echo ${MESG:="Hello There!"} [Return]   Display MESG value.
Hello There!
$ echo $MESG [Return] . . . . . . . . . . Check the MESG variable.
Hello There! . . . . . . . . . . . . . . . It is set to the specified
                                           string.
$_ . . . . . . . . . . . . . . . . . . . . Return to prompt.
```

The *word* can be a string with embedded white spaces. It works as long as you place it in quotation marks.

1. *The shell evaluates the variable* MESG. *It is empty. So the* **=** *option results in setting* MESG *to the string* **Hello There!**. *The substitution is done, and the* **echo** *command shows the value stored in* MESG.
2. *The value of the* MESG *variable is changed and it is not an empty variable any more.*

${parameter:?*word*} The **?** option means if the listed variable (parameter) is set and is not empty, then substitute its value. Otherwise, if the variable is empty, print the string

word and exit the current script. If *word* is omitted, then show the preset message **parameter null or not set**. For example:

```
$ MESG=[Return] . . . . . . . . . . . . . . . . . . . . . MESG is an empty variable.
$ echo ${MESG:? "ERROR!"} [Return]  . . . . . . . . . . Perform the substitution according
                                                        to the option.
ERROR!
$_ . . . . . . . . . . . . . . . . . . . . . . . . . . . Return to prompt.
```

The shell evaluates the MESG *variable; it is an empty variable. So the* **?** *option results in the substitution of the ERROR string, which is passed to the* **echo** *command to display it.*

You can use this option to show an error message and terminate your shell script if the user just presses **[Return]** in response to a **read** command. Figure 12.18 shows an example script file called `name`.

Figure 12.18

A Shell Script File Called `name`

```
$ cat -n name
   1 # name
   2 # Read a name from keyboard
   3 # A sample program to use the shell parameter substitutions
   4 #
   5 echo                                # skip a line
   6 echo "Enter your name: \c"          # prompts the user
   7 read name
   8 echo ${name:?"You must enter your name"}
   9 echo "Thank you. That's all."
  10 echo                                # skip a line
  11 exit 0                              # end of the program, exit
$_
```

Run the name script:

```
$ name [Return] . . . . . . . . . . Run the name script file.

Enter your name:_ . . . . . . . . . User prompt.
[Return] . . . . . . . . . . . . . Let's say you just press [Return].
You must enter your name . . . . . This is feedback to user.

$_ . . . . . . . . . . . . . . . . The script is terminated, and the
                                   prompt is back.
```

1. *If you press* **[Return]** *in response to the* **read** *command, name remains a null variable. The* **?** *option tests the variable; it is a null variable. The specified message is displayed, and the shell program is terminated.*
2. *However, if you enter your name or another word, then the script continues, and the message* **Thank you. That's all.** *is displayed.*

Let's run `name` again:

```
$ name [Return] . . . . . . . . . . Run name.

Enter your name: _  . . . . . . . . User prompt.
Emma [Return] . . . . . . . . . . . Enter your name.
Thank you. That's all.

$_ . . . . . . . . . . . . . . . . . Prompt is back.
```

12.4 ARITHMETIC OPERATIONS

The shell does not include a simple, built-in operator for arithmetic operations. For example, let's try to add one to the value of a shell variable. The result is not what you might expect.

```
$ x=10 [Return] . . . . . Initialize x to 10.
$ echo $x [Return] . . . . . Display the x value.
10
$ x=$x + 1 [Return]. . . . . Adds one to the x value.
$ echo $x [Return] . . . . . Display the x value.
10 + 1
$_ . . . . . Return to prompt.
```

The shell did not add the number 1 to the value of x. *It concatenated the string* +1 *to the string value of* x.

12.4.1 Arithmetic Operations: The expr Command

You can use the **expr** command to evaluate expressions. This command provides arithmetic operations capability and evaluates either numeric or nonnumeric character strings. The **expr** command takes the arguments as expressions, evaluates them, and displays the results on the standard output device.

Arithmetic Operators

The following command sequences introduce the arithmetic operator and demonstrate the use of the **expr** command to evaluate constants.

The following example demonstrates adding and subtracting constants:

```
$ expr 1 + 2 [Return]  . . . The + sign is the addition operator.
3
$ expr 15 - 6 [Return] . . . The - sign is the subtraction operator.
9
$ expr 6 - 15 [Return]
-9
$ expr 2.4 - 1[Return] . . . This is a bad argument (2.4); integers only.
expr: nonnumeric argument . UNIX error message.
$ expr 1+1 [Return]  . . . . This is a bad argument (1+1).
1+1 . . . . . It displays string 1+1.
$_
```

Spaces between the elements of an expression are necessary. The correct command is typed as `expr 1 + 1`. Notice the spaces around the + sign.

This example shows dividing, multiplying, and calculating a division remainder:

```
$ expr 10 / 2 [Return]  . . . . . The / sign is the division operator.
5
```

```
$ expr 10 \* 2 [Return] . . . . . .  The * is the multiplication
                                      operator.
20
$ expr 10 \% 3 [Return] . . . . . .  The % is the remainder
                                      operator.
1
$_
```

Because the * (multiplication) and % (remainder) characters have special meaning to the shell, they must be preceded by a backslash [\] for the shell to ignore their special meanings.

The following example demonstrates adding an integer to a shell variable:

```
$ x=10 [Return] . . . . . .  Initializes x to 10.
$ x=`expr $x + 1` [Return] .  Adds 1 to x.
$ echo $x [Return] . . . . .  Displays x.
11
$_ . . . . . .  Prompt.
```

The grave accent marks [` and `] surrounding the command are necessary and cause the output of the command (expr) to be substituted.

Relational Operators

The **expr** command provides relational operators that work on both numeric and nonnumeric arguments. If both arguments are numeric, the comparison is numeric. If one or both arguments are nonnumeric, the comparison is nonnumeric and uses the ASCII values. The relational operators are

```
=  . . . . . .  equal to
!= . . . . . .  not equal to
<  . . . . . .  less than
<= . . . . . .  less than or equal to
>  . . . . . .  greater than
>= . . . . . .  greater than or equal to
```

1. *The* **expr** *command displays 1 when the comparison is true.*
2. *The* **expr** *command displays 0 when the comparison is false.*

The following command sequences show the use of the **expr** command with the relational operators:

```
$ expr Gabe = Gabe [Return] . . .  Alphabetic comparison.
1 . . . . . .  Display 1, indicating true.
$ expr Gabe = Daniel [Return] . . .  Alphabetic comparison.
0 . . . . . .  Display 0, indicating false.
$ expr 10 \< 20 [Return] . . . . . .  Numeric comparison.
1 . . . . . .  Display 1, indicating true.
$ expr 10 \> 20 [Return] . . . . . .  Numeric comparison.
0 . . . . . .  Display 0, indicating false.
$_ . . . . . .  Return to prompt.
```

Again, because the > (greater than) and < (less than) characters have special meaning to the shell, they must be preceded by a backslash [\] for the shell to ignore their special meanings.

```
$ expr 6 \> A [Return] . . . . . Mixed comparison (numeric and
                                 alphabetic).
0  . . . . . . . . . . . . . . . Display 0, indicating false.
$_ . . . . . . . . . . . . . . . Return to prompt.
```

The **expr** *command treated 6 as an alphabetic argument, and compared the ASCII value of 6 (54) with the ASCII value of letter A (65).*

12.4.2 Arithmetic Operations: The let Command

You can use the **let** command for dealing with integer arithmetic. The **let** command is a simpler alternative to the **expr** command and includes all the basic arithmetic operations, including addition, subtraction, multiplication, and division. For example:

```
$ x=100 [Return]. . . . . . . . . Initialize x to 100.
$ let x=x+1 [Return]. . . . . . . Add 1 to x.
$ echo $x [Return]. . . . . . . . Display x value.
101
$ let y=x*2 [Return]. . . . . . . Use let command for multiplication.
$ echo $y [Return]  . . . . . . . Display y value.
202
$_
```

1. *The* **let** *command automatically uses the values of the variables. In this example, the value of the variable* x *is obtained by typing* **x** *instead of* **$x**.
2. *The* **let** *command interprets the* ***** *and* **%** *signs as the multiplication and remainder operators respectively, and there is no need to use the* ***** *or* **\%** *to prevent their special meanings.*

12.5 THE LOOP CONSTRUCTS

You use loop constructs in programs when you want to repeat a set of statements or commands. The loop constructs save a lot of time for you, the programmer. Can you imagine writing 100 lines of code to display a simple message 100 times? The shell provides you with three looping constructs: the **for** loop, **while** loop, and **until** loop. These loops enable you to execute commands repeatedly for a certain number of times or until certain conditions are met.

As was mentioned before, the **cat -n** *command is used throughout this chapter to display the source code for the script files. You can create these files in the* `Chapter12/12.5` *directory using the vi editor. Creating these files in a specific directory will keep your files organized and is highly recommended.*

12.5.1 The For Loop: The for-in-done Construct

The **for** loop is used to execute a set of commands a specified number of times. Its basic format is as follows:

```
for variable
in list-of-values
do
   commands
   ...
   last-command
done
```

The shell scans the *list-of-values*, stores the first *word* (value) in the *loop variable*, and executes the commands between **do** and **done** (what is called the *body* of the loop). Next, the second *word* is assigned to the loop variable, and again the body of the loop is executed. The commands in the body of the loop are executed for as many values as the *list-of-values* contains.

Figure 12.19 shows the source code for the `for_in_done` script. This program shows the usage of the *for-in-done* loop.

Figure 12.19
The `for_in_done` Script File

```
$ cat -n for_in_done
   1 #
   2 # for_in_done
   3 # A sample program to show the for-in-done construct
   4 #
   5 echo                                # skip a line
   6 for count in 1 2 3                 # start of the loop
   7 do
   8   echo "In the loop for $count times"
   9 done                                # end of the for loop
  10 echo                                # skip a line
  11 exit 0                             # end of the program
$_
```

The following command sequences show how the **for** loop works. This is a command line version of the `for_in_done` program. Notice that you are typing the program on the command line and the shell prompts you with > until the loop command is completed when you type in done. Refer to Figure 12.19 for the source code of this program.

```
$ for count in 1 2 3 [Return]  . .  Set the loop header.
> do [Return] . . . . . . . . . . .  Program waits for you to
                                     complete the command.
> echo "In the loop for $count times" [Return]
  . . . . . . . . . . . . . . . . . .  Display message.
> done [Return] . . . . . . . . . .  End of the for loop.
In the loop for 1 times
```

```
In the loop for 2 times
In the loop for 3 times
$_  . . . . . . . . . . . . . . . . . . . . Return to prompt.
```

1. *In this example,* count *is the loop variable, the* list-of-values *consists of numbers (1, 2, and 3), and the body of the loop is a single* **echo** *command.*
2. *Because three values are listed in the* list-of-values, *the body of the loop is executed a total of three times.*
3. *The values in the* list-of-values *are assigned one by one to the* count *variable. Each time through the loop, a new value is assigned until the* list-of-values *is exhausted.*

Scenario Suppose you want to save the name of the files you print along with the time you print them in a file. You write a script file called `slp` (super line printer) that prints your files and saves the information about them in a file called `pfile`. Figure 12.20 shows one way to write an `slp` script.

Figure 12.20

The `slp` Script File

```
$ cat -n slp
   1 #
   2 # slp
   3 # super line printer
   4 # A sample program to show the for_in_done construct
   5 #
   6 echo                                          # skip a line
   7 echo "Enter the name of the file(s)>>\c"      # prompt the user
   8 read filename                                 # read input
   9 for FILE in $filename                         # start of the for loop
  10 do
  11  echo "\nFile name:$FILE\n Printed:`date`" >> pfile # save in pfile
  12  lp $FILE                                    # print the file
  13 done                                         # end of the for loop
  14 echo "\n\07Job done"                         # inform user
  15 echo                                         # skip a line
  16 exit 0                                       # end of the program, exit
$_
```

1. *You can enter more than one filename. Each filename from the list of files in variable* filename *is assigned to the loop variable* `FILE`. *The* **echo** *command takes care of saving the information in* `pfile`, *and the* **lp** *command prints it.*
2. *The* **echo** *command creates* `pfile` *the first time you use this program. On consequent runs, it appends (the* `>>` *redirection operator) the filenames of the printed files.*

Run the `slp` program:

```
$ slp [Return]   . . . . . . . . . . . . Run it.
Enter the name of the files(s)>> . . . Waiting for input.
```

```
myfile [Return] . . . . . . . . . . . . . . You enter myfile.
lp: request id is lp 1-9223 (1 file) . . The lp message.
Job done . . . . . . . . . . . . . . . . . Beep and final message.
$_ . . . . . . . . . . . . . . . . . . . . Prompt.
$ cat pfile [Return] . . . . . . . . . . . Check pfile.
File name: myfile
Printed: Mon Dec 5   12:01:35 EST 2005
$_ . . . . . . . . . . . . . . . . . . . . The shell prompt appears.
```

12.5.2 The While Loop: The while-do-done Construct

The second type of looping construct explained here is the **while** loop. Unlike the **for** loop, whose number of iterations depends on the number of values in the *list-of-values*, the **while** loop continues as long as the loop condition is true. The format is as follows:

```
while [ condition ]
do
  commands
  ...
  last command
done
```

The commands in the body of the loop (between **do** and **done**) are repeatedly executed as long as the loop condition is true (nonzero). The loop condition must eventually return to false (zero); otherwise, your loop is an infinite loop and goes on forever. That is one of those times that you must know the **kill** key on your system so you can terminate the process.

Figure 12.21 shows the source code for the `carryon` script. This program shows the usage of the **while-do-done** loop.

Figure 12.21
The `carryon` Script

```
$ cat -n carryon
   1 #
   2 # carryon
   3 # A sample program to show the while-do-done construct
   4 #
   5 echo                              # skip a line
   6 carryon=Y                         # initialize the carryon variable
   7 while [ $carryon = Y ]            # start the while loop
   8 do
   9  echo "I do the job as long as you type Y:_\b\c"
  10  read carryon                     # read from keyboard
  11 done                             # end of the while loop
  12 echo "Job Done!"                  # end of the program message
  13 echo                             # skip a line
  14 exit 0                           # end of the program, exit
$_
```

The following command sequence shows how the **while** loop works. This is a command line version of the `carryon` program. Notice that you are typing the program on the command line and the shell prompts you with > until the loop command is completed when you type **done**. Refer to Figure 12.21 for the source code of this program.

```
$ carryon=Y [Return] . . . . . Initial the carryon variable.

$ while [ $carryon = Y ] [Return] . . . . . Set up the while loop.
> do [Return]
> echo "I do the job as long as you type Y:_\b\C" [Return]
> read carryon [Return]
> done [Return] . . . . . Read input.
I do the job as long as you type Y:_ . . . You are waiting for input.
Y [Return] . . . . . Press [Y] and [Return].
I do the job as long as you type Y:_ . . . You are waiting for input again.
N [Return] . . . . . Press [N] and [Return].
$_ . . . . . The shell prompt appears.
```

In this example, as long as you press **Y**, *the loop condition is true, and the body of the loop, the* **echo** *and* **read** *commands, is repeated. You stop this program by pressing any character but* **Y**.

Figure 12.22 shows the source code for the script file called `counter1`. The `counter1` program is another example of the **while** loop construct that displays the numbers from 0 to 9. The **expr** command is used to calculate the next number to be displayed. The **cat** command is used to display the source code.

Figure 12.22

The `counter1` Script File

```
$ cat -n counter1
   1 #
   2 # counter1
   3 # Counter: Counts from 1-9 using while loop
   4 #
   5 echo                              # skip a line
   6 count=1                           # initialize the count variable
   7 while [ $count -lt 10 ]           # start the while loop
   8 do
   9  echo $count                      # display count value
  10  count=`expr $count + 1`          # increment count
  11 done                             # end of the while loop
  12 echo "End! :-)"                   # end of the program message
  13 echo                             # skip a line
  14 exit 0                           # end of the program, exit
$_
```

Notice that the command **expr $count + 1** *is enclosed with a pair of grave accent marks (` and `). When you run the program, the output of this command is stored in the* count *variable.*

Running the `counter1` program.

```
$ counter1 [Return] . . . . Execute the counter1 program.

1
2
3
4
5
6
7
8
9
End! :-)

$_ . . . . . . . . . . . . . . . The shell prompt appears.
```

Figure 12.23 shows a new version of this script, `counter2`, using the **let** instead of the **expr** command. The **cat** command is used to show the source code.

Figure 12.23

The New Version of the `counter2`: Script File

```
$ cat -n counter2
   1 #
   2 # counter2
   3 # Counter: Counts from 1-9 using while loop and the let command
   4 #
   5 echo                              # skip a line
   6 count=1                           # initialize the count variable
   7 while [ $count -lt 10 ]           # start the while loop
   8 do
   9  echo $count                      # display count value
  10  let count=count+1                # increment count
  11 done                             # end of the while loop
  12 echo "End! :-)"                  # end of the program message
  13 echo                             # skip a line
  14 exit 0                           # end of the program, exit
$
```

The following versions of the `counter` *program work only with the* ksh *shell.*

The let Command Abbreviated

You can abbreviate the **let** command to double parentheses, **(( ))**. Figure 12.24 shows yet another version of the program we have been working with, now `counter3`. In this version the **let** command abbreviation is used instead of **expr**.

Figure 12.24
Another Version of the counter3: Script File

```
$ cat -n counter3
   1 #
   2 # counter3
   3 # Counter: Counts from 1-9 using while loop and the let command
   4 #          abbreviation (())
   5 #
   6 echo                              # skip a line
   7 count=1                           # initialize the count variable
   8 while (( $count < 10 ))           # start of the while loop
   9 do
  10  echo $count                      # display count value
  11  (( count=count+1 ))              # increment count
  12 done                             # end of the while loop
  13 echo "Done! :-)"                 # end of the program message
  14 echo                             # skip a line
  15 exit 0                           # end of the program, exit
$_
```

12.5.3 The Until Loop: The until-do-done Construct

The third type of the looping construct explained here is the **until** loop. The **until** loop is similar to the **while** loop, except that it continues executing the body of the loop as long as the condition of the loop is false (zero). The **until** loop is useful in writing scripts whose execution depends on other events occurring. The format is as follows:

```
until [ condition ]
do
  commands
  ...
  last-command
done
```

The body of the until loop might never get executed if the loop condition is true (nonzero) the first time it is executed.

Scenario You want to check whether a specified user is on the system or, if not, to be informed as soon as the user logs in. Figure 12.25 shows one way to write the script file uon (User ON).

In the uon script, the **until** loops stops as soon as the loop condition is true. If **grep** (Chapter 9) does not find the specified user ID passed to it by the **who** command in the list of the users, then the loop condition remains false (nonzero) and the **until** loop continues executing the body of the loop, the **sleep** command. As soon as **grep** finds the specified user ID in the users list, the loop condition becomes true (zero) and the loop stops. Next, the command after the loop is executed, which informs you, with two beeps, that the specified user is logged in.

1. *The output of the* **grep** *command is redirected to the null device (never-never land). That means you neither want to see the output nor save it.*

Figure 12.25
The uon Script File

```
$ cat -n uon
   1 #
   2 # uon
   3 # Let me know if xyz is on the system
   4 #
   5 echo                                # skip a line
   6 until who | grep "$1" > /dev/null# redirect the output of grep
   7 do
   8   sleep 30                          # wait half a minute
   9 done                                # end of the until loop
  10 echo "\07\07$1 is on the system" # inform user
  11 echo                                # skip a line
  10 exit 0                             # end of the program, exit
$
```

2. *The* **sleep** *command stops your program for half a minute. The net effect is that the* **grep** *command checks the list of users every half minute.*

There is only one problem left. If you run this program in the foreground and the specified user is not logged in, then you are at the mercy of the user to log in, and you cannot use your terminal while this program is running. A better idea is to run uon in the background.

Run uon in the background:

```
$ uon emma &[Return]  . . . . . .  Run it in the background.
                                   Check to see if emma is logged in.
4483 . . . . . . . . . . . . . . . .  Process ID.
$_ . . . . . . . . . . . . . . . . .  The prompt is back.
You can do other work. You will be informed when emma logs in.
emma is on the system . . . . . .  Beep beep. You are informed.
You can go on with whatever you were doing.
```

12.6 DEBUGGING SHELL PROGRAMS

It is easy to make mistakes when you write long and complex script files. Because you do not compile script files, you do not have the luxury of compiler error checking. Therefore, you have to run the program and try to decipher the error messages displayed on the screen. But do not despair!

12.6.1 The Shell Command

You can use the **sh**, **ksh**, or **bash** command with one of its options to make the debugging of your script files easier. For example, the **-x** option causes the shell to echo each command it executes. This trace of your script execution can help you to find the whereabouts of the bugs in your program.

Shell Options

Table 12.9 summarizes the shell command options, and the following examples show how to use them. In the following examples, **sh** is used, but you can use one of the other shells, **ksh** or **bash**.

Table 12.9
The **sh** Command Options

Option	Operation
-n	Reads commands but does not execute them.
-v	Shows the shell input lines as they are read.
-x	Shows the commands and their arguments as they are executed.

If you want to debug the BOX script file, you type the following:

```
$ sh -x BOX [Return]
```

You also can place the **sh** options as commands in your script file. Use the **set** command and type the following line at the beginning of your file, or place it wherever you want the debugging to start:

```
set -x
```

-x Option The **-x** option shows the commands in your script file as they look after the parameter and command substitutions have taken place. Using the **sh** command with the **-x** option, let's execute the BOX script file in different ways and explore the possibilities.

Run BOX with the **-x** option, but do not specify command line arguments:

```
$ sh -x BOX[Return] . . . . . Use the -x option.
+ echo The following is the output of the BOX script.
The following is the output of the BOX script.
+ echo Total number of command line arguments: 0
Total number of command line arguments: 0
+ echo The first parameter is:
The first parameter is:
+ echo The second parameter is:
The second parameter is:
+ echo This is the list of all parameters:
This is the list of all parameters:
$_ . . . . . Return to prompt.
```

1. *All lines beginning with plus signs (+) are commands that are executed by the shell, and the lines below them show the output of the commands.*
2. *The* **echo** *commands are displayed after the variable substitutions are done.*

Run BOX using **sh** with the -**x** option, and specify some command line parameters:

```
$ sh -x BOX of candy [Return]  . . . Use the -x option.
+ echo The following is the output of the BOX script.
```

```
The following is the output of the BOX script.
+ echo Total number of command line arguments: 2
Total number of command line arguments: 2
+ echo The first parameter is: of
The first parameter is: of
echo The second parameter is: candy
The second parameter is: candy
+ echo This is the list of all parameters: of candy
This is the list of all parameters: of candy
$_ . . . . . . . . . . . . . . . . . . Return to prompt.
```

-v Option The **-v** option is similar to the **-x** option. However, it displays the commands before the substitution of the variables and commands is done.

Run the BOX program with the **-v** option, and specify some command line arguments.

```
$ sh -v BOX is full of gold nuggets [Return] . . . Use the -v option.
echo "The following is the output of the $0 script."
The following is the output of the BOX script.
echo "The total number of command line arguments: $#"
Total number of command line arguments: 5
echo "The first parameter is: $1"
The first parameter is: is
echo "The second parameter is: $2"
The second parameter is: full
echo "This is the list of all parameters: $*"
This is the list of all parameters: is full of gold nuggets
$_ . . . . . . . . . . . . . . . . . . . . . . . . . . Return to prompt.
```

1. *One line shows the command that is executed by the shell, and the line below it shows the output of the command.*
2. *The* **echo** *commands are displayed before the variable substitutions are done. (This is different from the* **-x** *option, which displays the commands after the substitutions are done.)*

You can use the **-x** and **-v** options together on a command line. Using both options enables you to look at the commands in your file before and after execution, plus the output they produce. The following command line shows that the **sh** is invoked with both options:

```
$ sh -xv BOX [Return]
```

-n Option The **-n** option is used to detect syntax errors in your script file. Use this option when you want to make sure that you do not have syntax errors in a program before you run it.

For example, suppose you have a script file called `check_syntax`. Figure 12.26 shows the source code for this program, which has an intentional syntax error.

Run the `check_syntax` program using the -n option, and observe its output:

```
$ sh -n check_syntax [Return]  . . Run it.
check_syntax: syntax error at line 10 `else' unexpected
$_ . . . . . . . . . . . . . . . . . . Return to prompt.
```

Figure 12.26
Source Code for the `check_syntax` Program

```
$ cat -n check_syntax
   1 #
   2 # check_syntax
   3 # A sample program to show the output of the shell -n option
   4 #
   5 echo                    # skip a line
   6 echo "$0: checking the program syntax"
   7 if [ $# -gt 0 ]         # start of the if
   8 # then                  # this line is intentionally commented out
   9     echo "Number of the command line arguments: $#"
  10  else
  11     echo "No command line arguments"
  12 fi                      # end of the if
  13 echo "Bye!"             # end of the program message
  14 exit 0                  # end of the program, exit
$
```

1. *Using the* **-n** *option, none of the commands in your program are executed. This option only locates and recognizes the syntax errors.*
2. *If you use the* **-x** *or* **-v** *option, the program is executed until it reaches the part with the wrong syntax. Then, the error message is displayed and the program is terminated.*

What is wrong at line 10? You have to add the word *then* after the **if** statement to correct the **if-then-else** construct in your program. In our program example, the word *then* is commented out and it is sufficient to remove the # sign before the word *then.*

What is the output of `check_syntax` after you have fixed the syntax error? The **$0** contains the name of the program, in this case `check_syntax`. The **$#** contains the number of the command line arguments. If there are command line arguments, the body of the **if** is executed; otherwise, the body of the **else** is executed.

Run `check_syntax` again:

```
$ check_syntax one two three [Return] . . .   Use three command
                                               line arguments.
check_syntax: checking the program syntax
Number of the command line arguments: 3
$ . . . . . . . . . . . . . . . . . . . . . .  Return to prompt.
$ check_syntax [Return] . . . . . . . . . . .  No command
                                               line arguments
check_syntax: checking the program syntax
No command line arguments
$_ . . . . . . . . . . . . . . . . . . . . . .  Return to prompt.
```

The **sh** debugging options are mostly useful when you write long and complex shell programs. They will come in handy when you explore the script files presented in Chapter 13.

COMMAND SUMMARY

The following commands and options have been discussed in this chapter.

chmod

This command changes the access permission of a specified file according to the option letters indicating different categories of users. The user categories are **u** (for user/owner), **g** (for group), **o** (for others), and **a** (for all). The access categories are **r** (for read), **w** (for write), and **x** (for executable).

. (dot)

This command lets you run a process in the current shell environment and does not allow the shell to create a child process to run the command.

exit

This command terminates the current shell program whenever it is executed. It can also return a status code (RC) to indicate the success or failure of a program. It also terminates your login shell and logs you off if it is typed at the **$** prompt.

expr

This command is a built-in operator for arithmetic operations. It provides arithmetic and relational operators.

let

This command provides arithmetic operations.

read

This command reads input from the input device and stores the input string in one or more variables specified as the command arguments.

sh, ksh, or bash

This command invokes a new copy of the shell. You can run your script files using this command. Only three of the numerous options are mentioned.

Option	Operation
-n	Reads commands but does not execute them.
-v	Prints the input to the shell as the shell reads it.
-x	Prints command lines and their arguments as they are executed. This option is used mostly for debugging.

test
This command tests the condition of an expression given to it as an argument, and returns true or false depending on the status of the expression. It gives you the capability of testing different types of expressions.

REVIEW EXERCISES

1. How do you execute a shell script file?
2. What is the command to make a file an executable file?
3. When do you use the **.** (dot) command?
4. What is the command to read input from the keyboard?
5. Explain the command line parameters.
6. What are the shell positional variables?
7. How are the positional variables related to the command line parameters?
8. How do you debug your shell script?
9. Name the command that terminates a shell script.
10. What are the constructs in the shell language?
11. What is the use of the loop construct?
12. What is the difference between the **while** and **until** loops?
13. What is the command that performs arithmetic operations?
14. What is the command to add two integer numbers?
15. What is the command to multiply the values of two variables?
16. What is the shell built-in programming language?
17. What is the command to change the file permissions?
18. What is the command to change permission for a file called `xyz` to provide access to all users (user, group, and others)?
19. When do you use **sh** (**ksh**, **bash**) to execute a command?
20. When do we use the **chmod** command?
21. What does the command **sh -n** provide?
22. What does the command **sh -v** provide?
23. What does the command **sh -x** provide?
24. What is the command to read from the keyboard?
25. What is the **test** command?
26. What is the **let** command?
27. How do you write a comment line in a script (program)?
28. How do you compare two strings?
29. How do you compare two numeric values?
30. What is the command to show current date in mm/dd/yy format?

Terminal Session

In this terminal session, you are going to write a few script files and improve on the script file examples presented in this chapter.

1. Create a shell script named `LL` that lists your directory in a long format.
   a. Execute `LL` using the **sh** command.
   b. Change `LL` to an executable file.
   c. Execute `LL` again.
2. Create a script file that performs the following:
   a. Clears the screen.
   b. Skips two lines.
   c. Shows the current date and time.
   d. Shows the number of the users on the system.
   e. Beeps a few times and shows the message **Now at your service**
3. Modify the `largest` script file from this chapter to recognize the number of inputs and display appropriate messages.
4. Write a script file similar to `largest` that calculates the smallest of three integer numbers that are read from the keyboard. Make it able to recognize some input errors.
5. Create a script file for each of the script examples in this chapter that were typed at the **$** prompt. Make the appropriate modifications if necessary and run them. Use **sh** with the **-x** and other options to debug, and investigate the way shell scripts are executed.
6. Write a script file that sums the numbers passed to it as arguments on the command line and displays the results. Use the **for** loop construct in your program. For example, if you name this program `SUM`, and you type

   ```
   $ SUM 10 20 30 [Return]
   the program displays the following:
   10 + 20 + 30 = 60
   ```

7. Rewrite the `SUM` program, this time using the **while** loop.
8. Rewrite the `SUM` program, this time using the **until** loop.
9. Modify the script file `largest` from this chapter to accept numbers from the command line. For example, you type *LARGEST 1 2 3* and the program displays:

   ```
   The largest numbers is: 3
   ```

10. Modify the script file `greetings1` from this chapter and use the **cut** command to calculate the hour of the day.
11. Create a script file called `file_checker` that reads the filename entered and outputs the file properties such as: exists, readable and so on.
12. Write a script file named `d` that shows the current date.
13. Write a script file named `t` that shows the current time.
14. Write a script file called `s` that displays the name of your current shell.
15. Write a script file that checks if you have a `.profile` file in your `HOME` directory or not and displays an appropriate message.