The Complete Reference

Linux

Shell Programming

A shell program combines Linux commands in such a way as to perform a specific task. The Linux shell provides you with many programming tools with which to create shell programs. You can define variables and assign values to them. You can also define variables in a script file and have a user interactively enter values for them when the script is executed. There are loop and conditional control structures that repeat Linux commands or make decisions on which commands you want to execute. You can also construct expressions that perform arithmetic or comparison operations. All these programming tools operate like those found in other programming languages.

You can combine shell variables, control structures, expressions, and Linux commands to form a shell program. Usually, the instructions making up a shell program are entered into a script file that can then be executed. You can create this script file using any standard editor. To run the shell program, you then execute its script file. You can even distribute your program among several script files, one of which will contain instructions to execute others. You can think of variables, expressions, and control structures as tools you use to bring together several Linux commands into one operation. In this sense, a shell program is a new, complex Linux command that you have created. The BASH, TCSH, and Z shells that are supported on Linux distributions each have their own shell programming language with slightly different syntax. This chapter discusses BASH shell programming. Table 1 lists several commonly used BASH shell commands discussed throughout this chapter and previously in Chapter 11.

BASH Shell Commands	Effect
break	Exits from **for**, **while**, or **until** loop
continue	Skips remaining commands in loop and continues with next iteration
echo	Displays values **-n** eliminates output of new line
eval	Executes the command line
exec	Executes command in place of current process; does not generate a new subshell, but uses the current one
exit	Exits from the current shell
export ***var***	Generates a copy of **var** variable for each new subshell (call-by-value)
history	Lists recent history events

Table 1. *BASH Shell Commands and Arguments*

BASH Shell Commands	Effect
`let` "*expression*"	Evaluates an arithmetic, relational, or assignment expression using operators listed in Table 41-3. The expression must be quoted
`read`	Reads a line from the standard input
`return`	Exits from a function
`set`	Assigns new values for these arguments (when used with command line arguments); lists all defined variables (when used alone)
`shift`	Moves each command line argument to the left so that the number used to reference it is one less than before; argument $3 would then be referenced by $2, and so on; $1 is lost

`test` *value option value*
`[` *value option value* `]`
	Compares two arguments; used as the Linux command tested in control structures
	`test 2 -eq $count`
	`[ 2 -eq $count ]`

`unset`	Undefines a variable
Command Line Arguments	
`$0`	Name of Linux command
`$n`	The *n*th command line argument beginning from 1, $1–$n; you can use `set` to change them
`$*`	All the command line arguments beginning from 1; you can use `set` to change them
`$@`	The command line arguments individually quoted
`$#`	The count of the command line arguments
Process Variables	
`$$`	The PID number, process ID, of the current process
`$!`	The PID number of the most recent background job
`$?`	The exit status of the last Linux command executed

Table 1. *BASH Shell Commands and Arguments* (continued)

Shell Scripts: Commands and Comments

A shell script is a text file that contains Linux commands, which you enter using any standard editor. You can then execute the commands in the file by using the filename as an argument to any **sh** or dot command (**.**). They read the commands in shell scripts and execute them. You can also make the script file itself executable and use its name directly on the command line as you would use the name of any command. To better identify your shell scripts, you can add the **.sh** extension to them, as in **hello.sh**. However, this is not necessary.

You make a script file executable by setting its execute permission using the **chmod** command. The executable permission for the **chmod** command can be set using either symbolic or absolute references. The symbolic reference **u+x** sets the execute permission of a file. The command **chmod u+x hello** will set the execute permission of the **hello** script file. You can now use the script filename **hello** as if it were a Linux command. You only need to set the executable permission once. Once set, it remains set until you explicitly change it. The contents of the **hello** script are shown here.

hello

```
echo “Hello, how are you”
```

The user then sets the execute permission and runs the script using just the script name, in this case, hello.

```
$ chmod u+x hello
$ hello
Hello, how are you
$
```

An absolute reference will set read and write permissions at the same time that it sets the execute permission. See Chapter 12 for a more detailed explanation of absolute and symbolic permission references. In brief, a 700 will set execute as well as read and write permissions for the user; 500 will set only execute and read permissions; 300 only execute and write permissions; and 400 only execute permission. Users most often set 700 or 500. In the next example, the user sets the execute permission using an absolute reference:

```
$ chmod 750 hello
$ hello
Hello, how are you
$
```

It is often helpful to include in a script file short explanations describing what the file's task is as well as describing the purpose of certain commands and variables. You can enter such explanations using comments. A *comment* is any line or part of a line preceded by a sharp (or hash) sign, #, with the exception of the first line. The end of the comment is the next newline character, the end of the line. Any characters entered on a line after a sharp sign will be ignored by the shell. The first line is reserved for identification of the shell, as noted in the following discussion. In the next example, a comment describing the name and function of the script is placed at the head of the file.

hello

```
# The hello script says hello
echo “Hello, how are you”
```

You may want to be able to execute a script that is written for one of the Linux shells while you are working in another. Suppose you are currently in the TCSH shell and want to execute a script you wrote in the BASH shell that contains BASH shell commands. First you would have to change to the BASH shell with the **`sh`** command, execute the script, and then change back to the TCSH shell. You can, however, automate this process by placing, as the first characters in your script, `#!`, followed by the pathname for the shell program on your system.

Your shell always examines the first character of a script to find out what type of script it is—a BASH, PDKSH, or TCSH shell script. If the first character is a space, the script is assumed to be either a BASH or PDKSH shell script. If there is a # alone, the script is a TCSH shell script. If, however, the # is followed by a `!` character, then your shell reads the pathname of a shell program that follows. A `#!` should always be followed by the pathname of a shell program identifying the type of shell the script works in. If you are currently in a different shell, that shell will read the pathname of the shell program, change to that shell, and execute your script. If you are in a different shell, the space or # alone is not enough to identify a BASH or TCSH shell script. Such identification works only in their own shells. To identify a script from a different shell, you need to include the `#!` characters followed by a pathname.

For example, if you put **`#!/bin/sh`** at the beginning of the first line of the **hello** script, you could execute it directly from the TCSH shell. The script will first change to the BASH shell, execute its commands, and then return to the TCSH shell (or whatever type of shell it was executed from). In the next example, the **hello** script includes the **`#!/bin/sh`** command.

hello

```
#!/bin/sh
# The hello script says hello
echo “Hello, how are you”
```

The user then executes the script while in the TCSH shell.

```
> hello
Hello, how are you
```

Variables and Scripts

In the shell, you can create shell programs using variables and scripts. Within a shell program, you can define variables and assign values to them. Variables are used extensively in script input and output operations. The **read** command allows the user to interactively enter a value for a variable. Often **read** is combined with a prompt notifying the user when to enter a response. Another form of script input, called the Here document, allows you to use lines in a script as input to a command (discussed later). This overcomes the need to always read input from an outside source such as a file.

Definition and Evaluation of Variables: =, $, set, unset

A variable is defined in a shell when you first use the variable's name. A variable name may be any set of alphabetic characters, including the underscore. The name may also include a number, but the number cannot be the first character in the name. A name may not have any other type of character, such as an exclamation point, ampersand, or even a space. Such symbols are reserved by a shell for its own use. A name may not include more than one word, because a shell uses spaces to parse commands, delimiting command names and arguments.

You assign a value to a variable with the assignment operator. You type the variable name, the assignment operator, **=**, and then the value assigned. Note that you cannot place any spaces around the assignment operator. Any set of characters can be assigned to a variable. In the next example, the **greeting** variable is assigned the string "Hello":

```
$ greeting="Hello"
```

Once you have assigned a value to a variable, you can then use that variable to reference the value. Often, you use the values of variables as arguments for a command. You can reference the value of a variable using the variable name preceded by the **$** operator. The dollar sign is a special operator that uses a variable name to reference a variable's value, in effect evaluating the variable. Evaluation retrieves a variable's value—a set of characters. This set of characters then replaces the variable name on the command line. Thus, wherever a **$** is placed before the variable name, the variable name is replaced with the value of the variable.

In the next example, the shell variable **greeting** is evaluated and its contents, "Hello", are then used as the argument for an **echo** command. The **echo** command simply echoes or prints a set of characters to the screen.

```
$ echo $greeting
Hello
```

Tip *You can obtain a list of all the defined variables with the* **set** *command. If you decide that you do not want a certain variable, you can remove it with the* **unset** *command.*

Variable Values: Strings

The values that you assign to variables may consist of any set of characters. These characters may be a character string that you explicitly type in or the result obtained from executing a Linux command. In most cases, you will need to quote your values using either single quotes, double quotes, backslashes, or back quotes. Single quotes, double quotes, and backslashes allow you to quote strings in different ways. Back quotes have the special function of executing a Linux command and using the results as arguments on the command line.

Although variable values can be made up of any characters, problems occur when you want to include characters that are also used by the shell as operators. Your shell has certain special characters that it uses in evaluating the command line. If you want to use any of these characters as part of the value of a variable, you must first quote them. Quoting a special character on a command line makes it just another character.

- A space is used to parse arguments on the command line.
- The asterisk, question mark, and brackets are special characters used to generate lists of filenames.
- The period represents the current directory.
- The dollar sign is used to evaluate variables, and the greater-than and less-than characters are redirection operators.
- The ampersand is used to execute background commands, and the vertical bar pipes execute output. Double and single quotes allow you to quote several special characters at a time. Any special characters within double or single quotes are quoted. A backslash quotes a single character—the one that it precedes. If you want to assign more than one word to a variable, you need to quote the spaces separating the words. You can do so by enclosing the words within double quotes. You can think of this as creating a character string to be assigned to the variable. Of course, any other special characters enclosed within the double quotes will also be quoted.

The following examples show three ways of quoting strings. In the first example, the double quotes enclose words separated by spaces. Because the spaces are enclosed within double quotes, they are treated as characters—not as delimiters used to parse command line arguments. In the second example, single quotes also enclose a period, treating it as just a character. In the third example, an asterisk is also enclosed within the double quotes. The asterisk is considered just another character in the string and is not evaluated.

```
$ notice="The meeting will be tomorrow"
$ echo $notice
The meeting will be tomorrow
```

```
$ message='The project is on time.'
$ echo $message
The project is on time.

$ notice="You can get a list of files with ls *.c"
$ echo $notice
You can get a list of files with ls *.c
```

Double quotes, however, do not quote the dollar sign—the operator that evaluates variables. A **$** next to a variable name enclosed within double quotes will still be evaluated, replacing the variable name with its value. The value of the variable will then become part of the string, not the variable name. There may be times when you want a variable within quotes to be evaluated. In the next example, the double quotes are used so that the winner's name will be included in the notice:

```
$ winner=dylan
$ notice="The person who won is $winner"

$ echo $notice
The person who won is dylan
```

You can always quote any special character, including the **$** operator, by preceding it with a backslash. The backslash is useful when you want to evaluate variables within a string and also include **$** characters. In the next example, the backslash is placed before the dollar sign in order to treat it as a dollar sign character, **\$**. At the same time, the variable **$winner** is evaluated, since double quotes do not themselves quote the **$** operator.

```
$ winner=dylan
$ result="$winner won \$100.00""
$ echo $result
dylan won $100.00
```

Values from Linux Commands: Back Quotes

Though you can create variable values by typing in characters or character strings, you can also obtain values from other Linux commands. However, to assign the result of a Linux command to a variable, you first need to execute the command. If you place a Linux command within back quotes on the command line, that command is executed first and its result becomes an argument on the command line. In the case of assignments, the result of a command can be assigned to a variable by placing the command within back quotes to execute it first.

Tip *Think of back quotes as a kind of expression that contains both a command to be executed and its result, which is then assigned to the variable. The characters making up the command itself are not assigned.*

In the next example, the command **`ls *.c`** is executed and its result is then assigned to the variable **`listc`**. The command **`ls *.c`** generates a list of all files with a **.c** extension, and this list of files will then be assigned to the **`listc`** variable.

```
$ listc=`ls *.c`
$ echo $listc
main.c prog.c lib.c
```

Script Input and Output: echo, read, and <<

Within a script, you can use the **`echo`** command to output data and the **`read`** command to read input into variables. Also within a script, the **`echo`** command will send data to the standard output. The data is in the form of a string of characters. As you have seen, the **`echo`** command can output variable values as well as string constants. The **`read`** command reads in a value for a variable. It is used to allow a user to interactively input a value for a variable. The **`read`** command literally reads the next line in the standard input. Everything in the standard input up to the newline character is read in and assigned to a variable. In shell programs, you can combine the **`echo`** command with the **`read`** command to prompt the user to enter a value and then read that value into a variable. In the **greetvar** script in the next example, the user is prompted to enter a value for the greeting variable. The **`read`** command then reads the value the user typed and assigns it to the **`greeting`** variable.

greetvar

```
echo Please enter a greeting:
read greeting
echo “The greeting you entered was $greeting”
```

The **greetvar** script is then run, as shown here:

```
$ greetvar
Please enter a greeting:
hi
The greeting you entered was hi
$
```

If the value of a variable is a special character and the variable's value is referenced with a **$**, then the special character will be evaluated by the shell. However, placing the evaluated variable within quotes prevents any evaluation of special characters such as **$**.

In the **greetvar** script, **`$greeting`** was placed within a quoted string, preventing evaluation of any special characters. If **`$greeting`** is not quoted, then any special characters it contains will be evaluated.

The Here operation is a redirection operation, redirecting data within a shell script into a command. It is called Here because the redirected data is here in the shell script, not somewhere else in another file. The Here operation is represented by two less-than signs, **<<**. The **<<** operator can be thought of as a kind of redirection operator, redirecting lines in a shell script as input to a command. The **<<** operator is placed after the command to which input is being redirected. Lines following the **<<** operator are then taken as input to the command. The end of the input can be specified by an end-of-file character, CTRL-D. Instead of using an end-of- file character, you can specify your own delimiter. A word following the **<<** operator on the same line is taken to be the ending delimiter for the input lines. The delimiter can be any set of symbols. All lines up to the delimiter are read as input to the command.

In the next example, a message is sent to the user mark. The input for the message is obtained from a Here operation. The delimiter for the Here operation is the word **`myend`**.

mailmark
```
mail mark << myend
Did you remember the meeting
 robert
myend
```

Script Command Line Arguments

Like Linux commands, a shell script can take arguments. When you invoke a script, you can enter arguments on the command line after the script name. These arguments can then be referenced within the script using the **`$`** operator and the number of its position on the command line. Arguments on the command line are sequentially numbered from 1. The first argument is referenced with **`$1`**, the second argument with **`$2`**, and so on. The argument **`$0`** will contain the name of the shell script, the first word on the command line.

Note *These argument references can be thought of as referencing read-only variables. For those familiar with programming terminology, you can think of words on the command line as arguments that are passed into argument variables, **`$1`** through **`$9`**.*

The argument variables are read-only variables. You cannot assign values to them. Once given the initial values, they cannot be altered. In this sense, argument variables function more as constants—constants determined by the command line arguments. Each word on the command line is parsed into an argument unless it is quoted. If you enter more than one argument, you can reference them with each corresponding argument number. In the next example, four arguments are entered on the command line.

greetargs

```
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "The third argument is: $3"
echo "The fourth argument is: $4"
```

Here is a run of the greetargs script:

```
$ greetargs Hello Hi Salutations "How are you"
The first argument is: Hello
The second argument is: Hi
The third argument is: Salutations
The fourth argument is: How are you
$
```

A set of special arguments allows you to reference different aspects of command line arguments, such as the number of arguments or all the arguments together: **$***, **$@**, **$#**. The **$#** argument contains the number of arguments entered on the command line. This is useful when you need to specify a fixed number of arguments for a script. The argument **$*** references all the arguments in the command line. A command line may have more than nine arguments. The **$@** also references all the arguments on the command line, but allows you to separately quote each one. The difference between **$*** and **$@** is not clear until you use them to reference arguments using the **for-in** control structure. For this reason, they are discussed only briefly here and more extensively in the section on control structures later in the chapter.

Export Variables and Script Shells

When you execute a script file, you initiate a new process that has its own shell. Within this shell you can define variables, execute Linux commands, and even execute other scripts. If you execute another script from within the script currently running, the current script suspends execution, and control is transferred to the other script. All the commands in this other script are first executed before returning to continue with the suspended script. The process of executing one script from another operates much like a function or procedure call in programming languages. You can think of a script calling another script. The calling script waits until the called script finishes execution before continuing with its next command.

Any variable definitions that you place in a script will be defined within the script's shell and only known within that script's shell. Variable definitions are local to their own shells. In a sense, the variable is hidden within its shell. Suppose, however, you want to be able to define a variable within a script and use it in any scripts it may call. You cannot do this directly, but you can export a variable definition from one shell to another using the **export** command. When the **export** command is applied to a variable, it will instruct the system to define a copy of that variable for each new subshell generated. Each new subshell will have its own copy of the exported variable. In the next example, the **myname** variable is defined and exported:

```
$ myname="Charles"
$ export myname
```

Note *It is a mistake to think of exported variables as global variables. A shell can never reference a variable outside of itself. Instead, a copy of the variable with its value is generated for the new shell. An exported variable operates to some extent like a scoped global parameter. It is copied to any shell derived from its own shell. Any shell script called directly or indirectly after the exported variable's shell will be given a copy of the exported variable with the initial value.*

In the BASH shell, an environment variable can be thought of as a regular variable with added capabilities. To make an environment variable, you apply the **export** command to a variable you have already defined. The **export** command instructs the system to define a copy of that variable for each new subshell generated. Each new subshell will have its own copy of the environment variable. This process is called *exporting variables.*

In the next example, the variable **myfile** is defined in the **dispfile** script. It is then turned into an environment variable using the **export** command. The **myfile** variable will consequently be exported to any subshells, such as that generated when **printfile** is executed.

dispfile

```
myfile="List"
export myfile
echo "Displaying $myfile"
pr -t -n $myfile
printfile
```

printfile

```
echo "Printing $myfile"
lp $myfile &
```

A run of the dispfile script follows:

```
$ dispfile
Displaying List
1 screen
2 modem
3 paper
Printing List
$
```

Arithmetic Shell Operations: let

The **let** command is the BASH shell command for performing operations on arithmetic values. With **let**, you can compare two values or perform arithmetic operations such as addition or multiplication on them. Such operations are used often in shell programs to manage control structures or perform necessary calculations. The **let** command can be indicated either with the keyword **let** or with a set of double parentheses. The syntax consists of the keyword **let** followed by two numeric values separated by an arithmetic or relational operator, as shown here:

```
$ let value1 operator value2
```

You can use as your operator any of those listed in Table 2. The **let** command automatically assumes that operators are arithmetic or relational. You do not have to quote shell-like operators. The **let** command also automatically evaluates any variables and converts their values to arithmetic values. This means that you can write your arithmetic operations as simple arithmetic expressions. In the next example, the **let** command multiplies the values 2 and 7. The result is output to the standard output and displayed.

```
$ let 2*7
14
```

Arithmetic Operators	Function
`*`	Multiplication
`/`	Division
`+`	Addition
`-`	Subtraction
`%`	Modulo—results in the remainder of a division
Relational Operators	
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to

Table 2. *BASH Shell Operators*

Arithmetic Operators	Function
Relational Operators (continued)	
`<=`	Less than or equal to
`=`	Equal in **`expr`**
`==`	Equal in **`let`**
`!=`	Not equal
`&`	Logical AND
`|`	Logical OR
`!`	Logical NOT

Table 2. *BASH Shell Operators* (continued)

If you want to have spaces between operands in the arithmetic expression, you must quote the expression. The **`let`** command expects one string.

```
$ let “2 * 7"
```

You can also include assignment operations in your **`let`** expression. In the next example, the result of the multiplication is assigned to **`res`**:

```
$ let “res = 2 * 7"
$ echo $res
14
$
```

You can also use any of the relational operators to perform comparisons between numeric values, such as checking to see whether one value is less than another. Relational operations are often used to manage control structures such as loops and conditions. In the next example, **`helloprg`** displays the word “hello” three times. It makes use of a **`let`** less-than-or-equal operation to manage the loop, **`let ”again <= 3 ”`**, and to increment the again variable, **`let ”again = again + 1"`**. Notice that when **`again`** is incremented, it does not need to be evaluated. No preceding **`$`** is needed. The **`let`** command will automatically evaluate variables used in expressions.

helloprg
```
again=1
while let “again <= 3"
do
```

```
echo $again Hello
let “again = again + 1"
done
```

Here is a run of the **helloprg** script.

```
$ helloprg
1 Hello
2 Hello
3 Hello
```

Control Structures

You can control the execution of Linux commands in a shell program with control structures. Control structures allow you to repeat commands and to select certain commands over others. A control structure consists of two major components: a test and commands. If the test is successful, the commands are executed. In this way, you can use control structures to make decisions as to whether commands should be executed.

There are two different kinds of control structures: *loops* and *conditions.* A loop repeats commands, whereas a condition executes a command when certain conditions are met. The BASH shell has three loop control structures: **`while`**, **`for`**, and **`for-in`**. There are two condition structures: **`if`** and **`case`**.

The **`while`** and **`if`** control structures are more for general purposes, such as performing iterations and making decisions using a variety of different tests. The **`case`** and **`for`** control structures are more specialized. The **`case`** structure is a restricted form of the **`if`** condition and is often used to implement menus. The **`for`** structure is a limited type of loop. It runs through a list of values, assigning a new value to a variable with each iteration.

The **`if`** and **`while`** control structures have as their test the execution of a Linux command. All Linux commands return an exit status after they have finished executing. If a command is successful, its exit status will be 0. If the command fails for any reason, its exit status will be a positive value referencing the type of failure that occurred. The **`if`** and **`while`** control structures check to see if the exit status of a Linux command is 0 or some other value. In the case of the **`if`** and **`while`** structures, if the exit status is a zero value, the command was successful and the structure continues.

The test Command

Often you may need to perform a test that compares two values. Yet the test used in control structures is a Linux command, not a relational expression. There is, however, a Linux command called **`test`** that can perform such a comparison of values. The **`test`** command will compare two values and return as its exit status a 0 if the comparison is successful.

With the **test** command, you can compare integers, compare strings, and even perform logical operations. The command consists of the keyword **test** followed by the values being compared, separated by an option that specifies what kind of comparison is taking place. The option can be thought of as the operator, but is written, like other options, with a minus sign and letter codes. For example, **-eq** is the option that represents the equality comparison. However, there are two string operations that actually use an operator instead of an option. When you compare two strings for equality you use the equal sign, **=**. For inequality, you use **!=**. Table 3 lists all the options and operators used by **test**. The syntax for the **test** command is shown here:

```
test value -option value
test string = string
```

Integer Comparisons	**Function**
-gt	Greater than
-lt	Less than
-ge	Greater than or equal to
-le	Less than or equal to
-eq	Equal
-ne	Not equal
String Comparisons	
-z	Tests for empty string
-n	Tests for string value
=	Equal strings
!=	Not-equal strings
str	Tests to see if string is not a null string
Logical Operations	
-a	Logical AND
-o	Logical OR
!	Logical NOT

Table 3. *BASH Shell Test Operators*

Integer Comparisons	**Function**
File Tests	
-f	File exists and is a regular file
-s	File is not empty
-r	File is readable
-w	File can be written to, modified
-x	File is executable
-d	Filename is a directory name
-h	Filename is a symbolic link
-c	Filename references a character device
-b	Filename references a block file

Table 3. *BASH Shell Test Operators* (continued)

In the next example, the user compares two integer values to see if they are equal. In this case, you need to use the equality option, **-eq**. The exit status of the **test** command is examined to find out the result of the test operation. The shell special variable **$?** holds the exit status of the most recently executed Linux command.

```
$ num=5
$ test $num -eq 10
$ echo $?
1
```

Instead of using the keyword **test** for the **test** command, you can use enclosing brackets. The command **test $greeting = ”hi”** can be written as

```
$ [ $greeting = “hi” ]
```

Similarly, the **test** command **test $num -eq 10** can be written as

```
$ [ $num -eq 10 ]
```

The brackets themselves must be surrounded by white space: a space, TAB, or ENTER. Without the spaces, it would be invalid.

The **`test`** command is used extensively as the Linux command in the test component of control structures. Be sure to keep in mind the different options used for strings and integers. Do not confuse string comparisons and integer comparisons. To compare two strings for equality, you use **`=`**; to compare two integers, you use the option **`-eq`**.

Conditions: if, if-else, elif, case

The BASH shell has a set of conditional control structures that allow you to choose what Linux commands to execute. Many of these are similar to conditional control structures found in programming languages, but there are some differences. The **`if`** condition tests the success of a Linux command, not an expression. Furthermore, the end of an **`if-then`** command must be indicated with the keyword **`fi`**, and the end of a **`case`** command is indicated with the keyword **`esac`**. The condition control structures are listed in Table 4.

Condition Control Structures: **`if`**, **`else`**, **`elif`**, **`case`**	**Function**

`if` *command* **`then`**
command
`fi`
	`if` executes an action if its **`test`** command is true.

`if` *command* **`then`**
command
`else`
command
`fi`
	`if-else` executes an action if the exit status of its **`test`** command is true; if false, then the **`else`** action is executed.

`if` *command* **`then`**
command
`elif` *command* **`then`**
command
`else`
command
`fi`
	`elif` allows you to nest **`if`** structures, enabling selection among several alternatives; at the first true **`if`** structure, its commands are executed and control leaves the entire **`elif`** structure.

`case` *string* **`in`**
pattern`)`
command`;;`
`esac`
	`case` matches the string value to any of several patterns; if a pattern is matched, its associated commands are executed.

Table 4. *BASH Shell Control Structures*

Condition Control Structures: `if`, `else`, `elif`, `case`	Function
command **&&** *command*	The logical AND condition returns a true 0 value if both commands return a true 0 value; if one returns a nonzero value, then the AND condition is false and also returns a nonzero value.
command **||** *command*	The logical OR condition returns a true 0 value if one or the other command returns a true 0 value; if both commands return a nonzero value, then the OR condition is false and also returns a nonzero value.
! *command*	The logical NOT condition inverts the return value of the command.
Loop Control Structures: **`while`**, **`until`**, **`for`**, **`for-in`**, **`select`**	

`while` *command*
`do`
command
`done`
	`while` executes an action as long as its **`test`** command is true.

`until` *command*
`do`
command
`done`
	`until` executes an action as long as its **`test`** command is false.

`for` *variable* **`in`** *list-values*
`do`
command
`done`
	`for-in` is designed for use with lists of values; the variable operand is consecutively assigned the values in the list.

`for` *variable*
`do`
command
`done`
	`for` is designed for reference script arguments; the variable operand is consecutively assigned each argument value.

Table 4. *BASH Shell Control Structures* (continued)

Condition Control Structures: `if`, `else`, `elif`, `case`	**Function**
Loop Control Structures: `while`, `until`, `for`, `for-in`, `select` (continued)	
`select` *string* **`in`** *item-list* **`do`** *command* **`done`**	**`select`** creates a menu based on the items in the *item-list*; then it executes the command; the command is usually a **`case`**.

Table 4. *BASH Shell Control Structures* (continued)

if-then

The **`if`** structure places a condition on commands. That condition is the exit status of a specific Linux command. If a command is successful, returning an exit status of 0, then the commands within the **`if`** structure are executed. If the exit status is anything other than 0, the command has failed and the commands within the **`if`** structure are not executed.

The **`if`** command begins with the keyword **`if`** and is followed by a Linux command whose exit condition will be evaluated. This command is always executed. After the command, the keyword **`then`** goes on a line by itself. Any set of commands may then follow. The keyword **`fi`** ends the command. Often, you need to choose between two alternatives based on whether or not a Linux command is successful. The **`else`** keyword allows an **`if`** structure to choose between two alternatives. If the Linux command is successful, those commands following the **`then`** keyword are executed. If the Linux command fails, those commands following the **`else`** keyword are executed. The syntax for the **`if-then-else`** command is shown here:

```
if Linux Command
 then
 Commands
 else
 Commands
fi
```

The **elsels** script in the next example executes the **`ls`** command to list files with two different possible options, either by size or with all file information. If the user enters an **`s`**, files are listed by size; otherwise, all file information is listed.

elsels

```
echo Enter s to list file sizes,
echo otherwise all file information is listed.
echo -n “Please enter option: “
```

```
read choice
if [ “$choice” = s ]
  then
     ls -s
  else
     ls -l
fi
echo Good-bye
```

A run of the **elsels** script is shown here:

```
$ elsels
Enter s to list file sizes,
otherwise all file information is listed.
Please enter option: s
total 2
 1 monday 2 today
$
```

The **`elif`** structure allows you to nest **`if-then-else`** operations. The **`elif`** structure stands for “else if.” With **`elif`**, you can choose between several alternatives. The first alternative is specified with the **`if`** structure, followed by other alternatives, each specified by its own **`elif`** structure. The alternative to the last **`elif`** structure is specified with an **`else`**. If the test for the first **`if`** structure fails, control will be passed to the next **`elif`** structure, and its test will be executed. If it fails, control is passed to the next **`elif`** and its test checked. This continues until a test is true. Then that **`elif`** has its commands executed and control passes out of the **`if`** structure to the next command after the **`fi`** keyword.

The Logical Commands: && and ||

The logical commands perform logical operations on two Linux commands. The syntax is as follows:

```
command && command
command || command
```

In the case of the logical AND, **&&**, if both commands are successful, the logical command is successful. For the logical OR, **`||`**, if either command is successful, the OR is successful and returns an exit status of 0. The logical commands allow you to use logical operations as your test command in control structures.

case

The **`case`** structure chooses among several possible alternatives. The choice is made by comparing a value with several possible patterns. Each possible value is associated with a set of operations. If a match is found, the associated operations are performed.

The **case** structure begins with the keyword **case**, an evaluation of a variable, and the keyword **in**. A set of patterns then follows. Each pattern is a regular expression terminated with a closing parenthesis. After the closing parenthesis, commands associated with this pattern are listed, followed by a double semicolon on a separate line, designating the end of those commands. After all the listed patterns, the keyword **esac** ends the **case** command. The syntax looks like this:

```
case string in
   pattern)
         commands
         ;;
   pattern)
         commands
         ;;
   *)
         default commands
         ;;
 esac
```

A pattern can include any shell special characters. The shell special characters are the *****, **[]**, **?**, and **|**. You can specify a default pattern with a single ***** special character. The ***** special character matches on any pattern and so performs as an effective default option. If all other patterns do not match, the ***** will. In this way, the default option is executed if no other options are chosen. The default is optional. You do not have to put it in.

Loops: while, for-in, for

The BASH shell has a set of loop control structures that allow you to repeat Linux commands. They are the **while**, **for-in**, and **for** structures. Like the BASH **if** structure, **while** and **until** test the result of a Linux command. However, the **for** and **for-in** structures do not perform any test. They simply progress through a list of values, assigning each value in turn to a specified variable. Furthermore, the **while** and **until** structures operate like corresponding structures found in programming languages, whereas the **for** and **for-in** structures are very different. The loop control structures are listed in Table 41-4.

while

The **while** loop repeats commands. A **while** loop begins with the keyword **while** and is followed by a Linux command. The keyword **do** follows on the next line. The end of the loop is specified by the keyword **done**. Here is the syntax for the **while** command:

```
while Linux command
 do
 commands
 done
```

The Linux command used in **`while`** structures is often a **`test`** command indicated by enclosing brackets. In the **myname** script, in the next example, you are asked to enter a name. The name is then printed out. The loop is controlled by testing the value of the variable again using the bracket form of the **`test`** command.

myname

```
again=yes
while [ "$again" = yes ]
 do
   echo -n "Please enter a name: "
   read name
   echo "The name you entered is $name"
   echo -n "Do you wish to continue? "
   read again
 done
 echo Good-bye
```

Here is a run of the **myname** script:

```
$ myname
Please enter a name: George
The name you entered is George
Do you wish to continue? yes
Please enter a name: Robert
The name you entered is Robert
Do you wish to continue? no
Good-bye
```

for-in

The **`for-in`** structure is designed to reference a list of values sequentially. It takes two operands—a variable and a list of values. The values in the list are assigned one by one to the variable in the **`for-in`** structure. Like the **`while`** command, the **`for-in`** structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the **`while`** loop, the body of a **`for-in`** loop begins with the keyword **`do`** and ends with the keyword **`done`**. The syntax for the **`for-in`** loop is shown here:

```
for variable in list of values
 do
 commands
 done
```

In the **mylistfor** script, the user simply outputs a list of each item with today's date. The list of items makes up the list of values read by the **`for-in`** loop. Each item is consecutively assigned to the **`grocery`** variable.

mylistfor
```
tdate=`date +%D`
for grocery in milk cookies apples cheese
  do
   echo "$grocery
   $tdate"
  done
```

A run of the **mylistfor** script follows:

```
$ mylistfor
milk 10/23/00
cookies 10/23/00
apples 10/23/00
cheese 10/23/00
$
```

The **`for-in`** loop is handy for managing files. You can use special characters to generate filenames for use as a list of values in the **`for-in`** loop. For example, the **`*`** special character, by itself, generates a list of all files and directories, and **`*.c`** lists files with the **.c** extension. The special character **`*`** placed in the **`for-in`** loop's value list will generate a list of values consisting of all the filenames in your current directory.

```
for myfiles in *
 do
```

The **cbackup** script makes a backup of each file and places it in a directory called **sourcebak**. Notice the use of the **`*`** special character to generate a list of all filenames with a **.c** extension.

cbackup
```
for backfile in *.c
  do
    cp $backfile sourcebak/$backfile
    echo $backfile
  done
```

A run of the **cbackup** script follows:

```
$ cbackup
io.c
```

```
lib.c
main.c
$
```

for

The **for** structure without a specified list of values takes as its list of values the command line arguments. The arguments specified on the command line when the shell file is invoked become a list of values referenced by the **for** command. The variable used in the **for** command is set automatically to each argument value in sequence. The first time through the loop, the variable is set to the value of the first argument. The second time, it is set to the value of the second argument.

The **for** structure without a specified list is equivalent to the list **$@**. **$@** is a special argument variable whose value is the list of command line arguments. In the next example, a list of C program files is entered on the command line when the shell file **cbackuparg** is invoked. In **cbackuparg**, each argument is automatically referenced by a **for** loop, and **backfile** is the variable used in the **for** loop. The first time through the loop, **$backfile** holds the value of the first argument, **$1**. The second time through, it holds the value of the second argument, **$2**.

cbackuparg

```
for backfile
  do
    cp $backfile sourcebak/$backfile
    echo "$backfile "
  done
```

A run of the **cbackuparg** script is shown here:

```
$ cbackuparg main.c lib.c io.c
main.c
lib.c
io.c
```