

Using Structured Commands

IN THIS CHAPTER

Working with the if-then statement

Nesting ifs

Understanding the test command

Testing compound conditions

Using double brackets and parentheses

Looking at case

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

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

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

Working with the if-then Statement

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

```
if command
then
    commands
fi
```

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

Linux® Command Line and Shell Scripting Bible, Third Edition. Richard Blum, Christine Bresnahan.
© 2015 by John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons Inc.

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

Here's a simple example to demonstrate this concept:

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

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

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

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

Here's another example:

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

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

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

NOTE

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

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

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

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

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

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

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

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

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

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

Exploring the if-then-else Statement

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

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

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

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

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

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

```

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

$

```

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

Nesting ifs

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

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

```

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

```

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

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

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

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

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

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

```

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

```

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

```

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

```

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

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

TIP

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

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

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

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

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

Trying the test Command

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

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

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

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

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

```
test condition
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Numeric comparisons
- String comparisons
- File comparisons

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

Using numeric comparisons

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

TABLE 12-1 The test Numeric Comparisons

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

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

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

The first test condition:

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

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

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

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

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

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

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

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

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

Using string comparisons

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

TABLE 12-2 The test String Comparisons

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

The following sections describe the different string comparisons available.

Looking at string equality

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

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

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

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

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

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

Looking at string order

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

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

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

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

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

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

```
$ cat test9.sh
#!/bin/bash
# mis-using string comparisons
#
val1=baseball
val2=hockey
#
if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./test9.sh
baseball is less than hockey
$
```

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

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

```
$ cat test9b.sh
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing
#
if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./test9b.sh
Testing is less than testing
$
$ sort testfile
testing
Testing
$
```

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

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

NOTE

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

Looking at string size

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

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



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

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

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

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

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

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

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

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

Tip

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

Using file comparisons

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

TABLE 12-3 The test File Comparisons

Comparison	Description
-d file	Checks if file exists and is a directory
-e file	Checks if file exists
-f file	Checks if file exists and is a file

Continues

TABLE 12.3 (continued)

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

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

Checking directories

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

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

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

Checking whether an object exists

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

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

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

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

Checking for a file

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

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

```
The item being checked: /home/Christine
```

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

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

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

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

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

The item being checked: /home/Christine/sentinel

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

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

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

Checking for read access

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

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

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

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

Checking for empty files

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

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

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

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

Checking whether you can write to a file

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

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

```
$ ./test16.sh
```

```
The item being checked: /home/Christine/sentinel
```

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

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

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

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

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

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

```
The item being checked: /home/Christine/sentinel
```

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

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

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

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

Checking whether you can run a file

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

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

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

Checking ownership

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

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

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

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

Checking default group membership

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

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

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

Checking file date

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

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

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

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

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

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

Considering Compound Testing

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

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

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

TIP

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

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

The following shows the AND Boolean operator in use:

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

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

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

Working with Advanced if-then Features

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

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

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

Using double parentheses

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

```
(( expression ))
```

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

TABLE 12-4 The Double Parentheses Command Symbols

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

Continues

TABLE 12.4 (continued)

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

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

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

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

Using double brackets

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

```
[[ expression ]]
```

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

NOTE

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

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

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

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

Considering the case Command

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

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

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

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

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

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

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

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



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

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

Summary

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

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

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

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

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

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

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

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