

# More Structured Commands

## IN THIS CHAPTER

Looping with the `for` statement

Iterating with the `until` statement

Using the `while` statement

Combining loops

Redirecting loop output

In the previous chapter, you saw how to manipulate the flow of a shell script program by checking the output of commands and the values of variables. In this chapter, we continue to look at structured commands that control the flow of your shell scripts. You'll see how you can perform repeating processes, commands that can loop through a set of commands until an indicated condition has been met. This chapter discusses and demonstrates the `for`, `while`, and `until` bash shell looping commands.

## The `for` Command

Iterating through a series of commands is a common programming practice. Often, you need to repeat a set of commands until a specific condition has been met, such as processing all the files in a directory, all the users on a system, or all the lines in a text file.

The bash shell provides the `for` command to allow you to create a loop that iterates through a series of values. Each iteration performs a defined set of commands using one of the values in the series. Here's the basic format of the bash shell `for` command:

```
for var in list
do
    commands
done
```

You supply the series of values used in the iterations in the `list` parameter. You can specify the values in the list in several ways.

In each iteration, the variable *var* contains the current value in the list. The first iteration uses the first item in the list, the second iteration the second item, and so on until all the items in the list have been used.

The *commands* entered between the *do* and *done* statements can be one or more standard bash shell commands. Within the commands, the *\$var* variable contains the current list item value for the iteration.

### NOTE

If you prefer, you can include the *do* statement on the same line as the *for* statement, but you must separate it from the list items using a semicolon: `for var in list; do`.

We mentioned that there are several different ways to specify the values in the list. The following sections show the various ways to do that.

## Reading values in a list

The most basic use of the *for* command is to iterate through a list of values defined within the *for* command itself:

```
$ cat test1
#!/bin/bash
# basic for command

for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo The next state is $test
done
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
$
```

Each time the *for* command iterates through the list of values provided, it assigns the *\$test* variable the next value in the list. The *\$test* variable can be used just like any other script variable within the *for* command statements. After the last iteration, the *\$test* variable remains valid throughout the remainder of the shell script. It retains the last iteration value (unless you change its value):

```
$ cat test1b
#!/bin/bash
```

```
# testing the for variable after the looping
for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo "The next state is $test"
done
echo "The last state we visited was $test"
test=Connecticut
echo "Wait, now we're visiting $test"
$ ./test1b
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
The last state we visited was Colorado
Wait, now we're visiting Connecticut
$
```

The `$test` variable retained its value and allowed us to change the value and use it outside of the `for` command loop, as any other variable would.

## Reading complex values in a list

Things aren't always as easy as they seem with the `for` loop. There are times when you run into data that causes problems. Here's a classic example of what can cause problems for shell script programmers:

```
$ cat badtest1
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if this'll work
do
    echo "word:$test"
done
$ ./badtest1
word:I
word:dont know if thisll
word:work
$
```

Ouch, that hurts. The shell saw the single quotation marks within the list values and attempted to use them to define a single data value, and it really messed things up in the process.

You have two ways to solve this problem:

- Use the escape character (the backslash) to escape the single quotation mark.
- Use double quotation marks to define the values that use single quotation marks.

Neither solution is all that fantastic, but each one helps solve the problem:

```
$ cat test2
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if "this'll" work
do
    echo "word:$test"
done
$ ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
$
```

In the first problem value, you added the backslash character to escape the single quotation mark in the `don't` value. In the second problem value, you enclosed the `this'll` value in double quotation marks. Both methods worked fine to distinguish the value.

Another problem you may run into is multi-word values. Remember that the `for` loop assumes that each value is separated with a space. If you have data values that contain spaces, you run into yet another problem:

```
$ cat badtest2
#!/bin/bash
# another example of how not to use the for command

for test in Nevada New Hampshire New Mexico New York North Carolina
do
    echo "Now going to $test"
done
$ ./badtest1
Now going to Nevada
Now going to New
Now going to Hampshire
Now going to New
Now going to Mexico
Now going to New
Now going to York
```

```
Now going to North
Now going to Carolina
$
```

Oops, that's not exactly what we wanted. The `for` command separates each value in the list with a space. If there are spaces in the individual data values, you must accommodate them using double quotation marks:

```
$ cat test3
#!/bin/bash
# an example of how to properly define values

for test in Nevada "New Hampshire" "New Mexico" "New York"
do
    echo "Now going to $test"
done
$ ./test3
Now going to Nevada
Now going to New Hampshire
Now going to New Mexico
Now going to New York
$
```

Now the `for` command can properly distinguish between the different values. Also, notice that when you use double quotation marks around a value, the shell doesn't include the quotation marks as part of the value.

## Reading a list from a variable

Often what happens in a shell script is that you accumulate a list of values stored in a variable and then need to iterate through the list. You can do this using the `for` command as well:

```
$ cat test4
#!/bin/bash
# using a variable to hold the list

list="Alabama Alaska Arizona Arkansas Colorado"
list=$list " Connecticut"

for state in $list
do
    echo "Have you ever visited $state?"
done
$ ./test4
Have you ever visited Alabama?
Have you ever visited Alaska?
Have you ever visited Arizona?
```

```
Have you ever visited Arkansas?
Have you ever visited Colorado?
Have you ever visited Connecticut?
$
```

The `$list` variable contains the standard text list of values to use for the iterations. Notice that the code also uses another assignment statement to add (or concatenate) an item to the existing list contained in the `$list` variable. This is a common method for adding text to the end of an existing text string stored in a variable.

### Reading values from a command

Another way to generate values for use in the list is to use the output of a command. You use command substitution to execute any command that produces output and then use the output of the command in the `for` command:

```
$ cat test5
#!/bin/bash
# reading values from a file

file="states"

for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
$ ./test5
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
$
```

This example uses the `cat` command in the command substitution to display the contents of the file `states`. Notice that the `states` file includes each state on a separate line, not separated by spaces. The `for` command still iterates through the output of the `cat` command one line at a time, assuming that each state is on a separate line. However, this doesn't solve the problem of having spaces in data. If you list a state with a space in it, the `for` command still takes each word as a separate value. There's a reason for this, which we look at in the next section.

### NOTE

The `test5` code example assigned the filename to the variable using just the filename without a path. This requires that the file be in the same directory as the script. If this isn't the case, you need to use a full pathname (either absolute or relative) to reference the file location.

## Changing the field separator

The cause of this problem is the special environment variable `IFS`, called the *internal field separator*. The `IFS` environment variable defines a list of characters the bash shell uses as field separators. By default, the bash shell considers the following characters as field separators:

- A space
- A tab
- A newline

If the bash shell sees any of these characters in the data, it assumes that you're starting a new data field in the list. When working with data that can contain spaces (such as filenames), this can be annoying, as you saw in the previous script example.

To solve this problem, you can temporarily change the `IFS` environment variable values in your shell script to restrict the characters the bash shell recognizes as field separators. For example, if you want to change the `IFS` value to recognize only the newline character, you need to do this:

```
IFS=$'\n'
```

Adding this statement to your script tells the bash shell to ignore spaces and tabs in data values. Applying this technique to the previous script yields the following:

```
$ cat test5b
#!/bin/bash
# reading values from a file

file="states"

IFS=$'\n'
```

```
for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
$ ./test5b
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
Visit beautiful New Hampshire
Visit beautiful North Carolina
$
```

Now the shell script can use values in the list that contain spaces.

### CAUTION

When working on long scripts, it's possible to change the IFS value in one place, and then forget about it and assume the default value elsewhere in the script. A safe practice to get into is to save the original IFS value before changing it and then restore it when you're finished.

This technique can be coded like this:

```
IFS.OLD=$IFS
IFS=$'\n'
<use the new IFS value in code>
IFS=$IFS.OLD
```

This ensures that the IFS value is returned to the default value for future operations within the script.

Other excellent applications of the IFS environment variable are possible. Suppose you want to iterate through values in a file that are separated by a colon (such as in the `/etc/passwd` file). You just need to set the IFS value to a colon:

```
IFS=:
```

If you want to specify more than one IFS character, just string them together on the assignment line:

```
IFS=$'\n' : ; "
```

This assignment uses the newline, colon, semicolon, and double quotation mark characters as field separators. There's no limit to how you can parse your data using the IFS characters.



## Reading a directory using wildcards

Finally, you can use the `for` command to automatically iterate through a directory of files. To do this, you must use a wildcard character in the file or pathname. This forces the shell to use *file globbing*. File globbing is the process of producing filenames or pathnames that match a specified wildcard character.

This feature is great for processing files in a directory when you don't know all the filenames:

```
$ cat test6
#!/bin/bash
# iterate through all the files in a directory

for file in /home/rich/test/*
do

    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
$ ./test6
/home/rich/test/dirl is a directory
/home/rich/test/myprog.c is a file
/home/rich/test/myprog is a file
/home/rich/test/myscript is a file
/home/rich/test/newdir is a directory
/home/rich/test/newfile is a file
/home/rich/test/newfile2 is a file
/home/rich/test/testdir is a directory
/home/rich/test/testing is a file
/home/rich/test/testprog is a file
/home/rich/test/testprog.c is a file
$
```

The `for` command iterates through the results of the `/home/rich/test/*` listing. The code tests each entry using the `test` command (using the square bracket method) to see if it's a directory, using the `-d` parameter, or a file, using the `-f` parameter (See Chapter 12).

Notice in this example that we did something different in the `if` statement tests:

```
if [ -d "$file" ]
```

In Linux, it's perfectly legal to have directory and filenames that contain spaces. To accommodate that, you should enclose the `$file` variable in double quotation marks. If you don't, you'll get an error if you run into a directory or filename that contains spaces:

```
./test6: line 6: [: too many arguments
./test6: line 9: [: too many arguments
```

The bash shell interprets the additional words as arguments within the `test` command, causing an error.

You can also combine both the directory search method and the list method in the same `for` statement by listing a series of directory wildcards in the `for` command:

```
$ cat test7
#!/bin/bash
# iterating through multiple directories

for file in /home/rich/*.b* /home/rich/badtest
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    else
        echo "$file doesn't exist"
    fi
done
$ ./test7
/home/rich/.backup.timestamp is a file
/home/rich/.bash_history is a file
/home/rich/.bash_logout is a file
/home/rich/.bash_profile is a file
/home/rich/.bashrc is a file
/home/rich/badtest doesn't exist
$
```

The `for` statement first uses file globbing to iterate through the list of files that result from the wildcard character; then it iterates through the next file in the list. You can combine any number of wildcard entries in the list to iterate through.

### CAUTION

Notice that you can enter anything in the list data. Even if the file or directory doesn't exist, the `for` statement attempts to process whatever you place in the list. This can be a problem when working with files and directories. You have no way of knowing if you're trying to iterate through a nonexistent directory: It's always a good idea to test each file or directory before trying to process it.

## The C-Style for Command

If you've done any programming using the C programming language, you're probably surprised by the way the bash shell uses the `for` command. In the C language, a `for` loop normally defines a variable, which it then alters automatically during each iteration. Typically, programmers use this variable as a counter and either increment or decrement the counter by one in each iteration. The bash `for` command can also provide this functionality. This section shows you how to use a C-style `for` command in a bash shell script.

### The C language for command

The C language `for` command has a specific method for specifying a variable, a condition that must remain true for the iterations to continue, and a method for altering the variable for each iteration. When the specified condition becomes false, the `for` loop stops. The condition equation is defined using standard mathematical symbols. For example, consider the following C language code:

```
for (i = 0; i < 10; i++)
{
    printf("The next number is %d\n", i);
}
```

This code produces a simple iteration loop, where the variable `i` is used as a counter. The first section assigns a default value to the variable. The middle section defines the condition under which the loop will iterate. When the defined condition becomes false, the `for` loop stops iterations. The last section defines the iteration process. After each iteration, the expression defined in the last section is executed. In this example, the `i` variable is incremented by one after each iteration.

The bash shell also supports a version of the `for` loop that looks similar to the C-style `for` loop, although it does have some subtle differences, including a couple of things that will confuse shell script programmers. Here's the basic format of the C-style bash `for` loop:

```
for (( variable assignment ; condition ; iteration process ))
```

The format of the C-style `for` loop can be confusing for bash shell script programmers, because it uses C-style variable references instead of the shell-style variable references. Here's what a C-style `for` command looks like:

```
for (( a = 1; a < 10; a++ ))
```

Notice that there are a couple of things that don't follow the standard bash shell `for` method:

- The assignment of the variable value can contain spaces.
- The variable in the condition isn't preceded with a dollar sign.
- The equation for the iteration process doesn't use the `expr` command format.

The shell developers created this format to more closely resemble the C-style `for` command. Although this is great for C programmers, it can throw even expert shell programmers into a tizzy. Be careful when using the C-style `for` loop in your scripts.

Here's an example of using the C-style `for` command in a bash shell program:

```
$ cat test8
#!/bin/bash
# testing the C-style for loop

for (( i=1; i <= 10; i++ ))
do
    echo "The next number is $i"
done
$ ./test8
The next number is 1
The next number is 2
The next number is 3
The next number is 4
The next number is 5
The next number is 6
The next number is 7
The next number is 8
The next number is 9
The next number is 10
$
```

The `for` loop iterates through the commands using the variable defined in the `for` loop (the letter *i* in this example). In each iteration, the `$i` variable contains the value assigned in the `for` loop. After each iteration, the loop iteration process is applied to the variable, which in this example, increments the variable by one.

### Using multiple variables

The C-style `for` command also allows you to use multiple variables for the iteration. The loop handles each variable separately, allowing you to define a different iteration process for each variable. Although you can have multiple variables, you can define only one condition in the `for` loop:

```
$ cat test9
#!/bin/bash
# multiple variables

for (( a=1, b=10; a <= 10; a++, b-- ))
do
    echo "$a - $b"
done
$ ./test9
```

```

1 - 10
2 - 9
3 - 8
4 - 7
5 - 6
6 - 5
7 - 4
8 - 3
9 - 2
10 - 1
$

```

The `a` and `b` variables are each initialized with different values, and different iteration processes are defined. While the loop increases the `a` variable, it decreases the `b` variable for each iteration.

## The while Command

The `while` command is somewhat of a cross between the `if-then` statement and the `for` loop. The `while` command allows you to define a command to test and then loop through a set of commands for as long as the defined test command returns a zero exit status. It tests the `test` command at the start of each iteration. When the `test` command returns a non-zero exit status, the `while` command stops executing the set of commands.

### Basic while format

Here's the format of the `while` command:

```

while test command
do
    other commands
done

```

The `test command` defined in the `while` command is the exact same format as in `if-then` statements (see Chapter 12). As in the `if-then` statement, you can use any normal bash shell command, or you can use the `test` command to test for conditions, such as variable values.

The key to the `while` command is that the exit status of the `test command` specified must change, based on the commands run during the loop. If the exit status never changes, the `while` loop will get stuck in an infinite loop.

The most common use of the `test command` is to use brackets to check a value of a shell variable that's used in the loop commands:

```

$ cat test10
#!/bin/bash

```

```
# while command test

var1=10
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$(( $var1 - 1 ])
done
$ ./test10
10
9
8
7
6
5
4
3
2
1
$
```

The `while` command defines the test condition to check for each iteration:

```
while [ $var1 -gt 0 ]
```

As long as the test condition is true, the `while` command continues to loop through the commands defined. Within the commands, the variable used in the test condition must be modified, or you'll have an infinite loop. In this example, we use shell arithmetic to decrease the variable value by one:

```
var1=$(( $var1 - 1 ]
```

The `while` loop stops when the test condition is no longer true.

### Using multiple test commands

The `while` command allows you to define multiple test commands on the `while` statement line. Only the exit status of the last test command is used to determine when the loop stops. This can cause some interesting results if you're not careful. Here's an example of what we mean:

```
$ cat test11
#!/bin/bash
# testing a multicommand while loop

var1=10

while echo $var1
```

```

        [ $var1 -ge 0 ]
do
    echo "This is inside the loop"
    var1=$(( $var1 - 1 ))
done
$ ./test11
10
This is inside the loop
9
This is inside the loop
8
This is inside the loop
7
This is inside the loop
6
This is inside the loop
5
This is inside the loop
4
This is inside the loop
3
This is inside the loop
2
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1
$

```

Pay close attention to what happened in this example. Two test commands were defined in the while statement:

```

while echo $var1
    [ $var1 -ge 0 ]

```

The first test simply displays the current value of the `var1` variable. The second test uses brackets to determine the value of the `var1` variable. Inside the loop, an `echo` statement displays a simple message, indicating that the loop was processed. Notice when you run the example how the output ends:

```

This is inside the loop
-1
$

```

The while loop executed the `echo` statement when the `var1` variable was equal to zero and then decreased the `var1` variable value. Next, the test commands were executed for

the next iteration. The `echo` test command was executed, displaying the value of the `var1` variable, which is now less than zero. It's not until the shell executes the `test` test command that the `while` loop terminates.

This demonstrates that in a multi-command `while` statement, all the test commands are executed in each iteration, including the last iteration when the last test command fails. Be careful of this. Another thing to be careful of is how you specify the multiple test commands. Note that each test command is on a separate line!

## The until Command

---

The `until` command works in exactly the opposite way from the `while` command. The `until` command requires that you specify a test command that normally produces a non-zero exit status. As long as the exit status of the test command is non-zero, the bash shell executes the commands listed in the loop. When the test command returns a zero exit status, the loop stops.

As you would expect, the format of the `until` command is:

```
until test commands
do
    other commands
done
```

Similar to the `while` command, you can have more than one *test command* in the `until` command statement. Only the exit status of the last command determines if the bash shell executes the *other commands* defined.

The following is an example of using the `until` command:

```
$ cat test12
#!/bin/bash
# using the until command

var1=100

until [ $var1 -eq 0 ]
do
    echo $var1
    var1=$(( $var1 - 25 ])
done
$ ./test12
100
75
50
25
$
```



This example tests the `var1` variable to determine when the `until` loop should stop. As soon as the value of the variable is equal to zero, the `until` command stops the loop. The same caution as for the `while` command applies when you use multiple test commands with the `until` command:

```
$ cat test13
#!/bin/bash
# using the until command

var1=100

until echo $var1
    [ $var1 -eq 0 ]
do
    echo Inside the loop: $var1
    var1=$(( $var1 - 25 ))
done
$ ./test13
100
Inside the loop: 100
75
Inside the loop: 75
50
Inside the loop: 50
25
Inside the loop: 25
0
$
```

The shell executes the test commands specified and stops only when the last command is true.

## Nesting Loops

A loop statement can use any other type of command within the loop, including other loop commands. This is called a *nested loop*. Care should be taken when using nested loops, because you're performing an iteration within an iteration, which multiplies the number of times commands are being run. If you don't pay close attention to this, it can cause problems in your scripts.

Here's a simple example of nesting a `for` loop inside another `for` loop:

```
$ cat test14
#!/bin/bash
```

```
# nesting for loops

for (( a = 1; a <= 3; a++ ))
do
    echo "Starting loop $a:"
    for (( b = 1; b <= 3; b++ ))
    do
        echo "    Inside loop: $b"
    done
done
$ ./test14
Starting loop 1:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 2:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 3:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
$
```

The nested loop (also called the *inner loop*) iterates through its values for each iteration of the outer loop. Notice that there's no difference between the `do` and `done` commands for the two loops. The bash shell knows when the first `done` command is executed that it refers to the inner loop and not the outer loop.

The same applies when you mix loop commands, such as placing a `for` loop inside a `while` loop:

```
$ cat test15
#!/bin/bash
# placing a for loop inside a while loop

var1=5

while [ $var1 -ge 0 ]
do
    echo "Outer loop: $var1"
    for (( var2 = 1; $var2 < 3; var2++ ))
    do
        var3=$(( $var1 * $var2 ))
        echo "    Inner loop: $var1 * $var2 = $var3"
    done
    var1=$(( $var1 - 1 ))
done
$ ./test15
```

```

Outer loop: 5
  Inner loop: 5 * 1 = 5
  Inner loop: 5 * 2 = 10
Outer loop: 4
  Inner loop: 4 * 1 = 4
  Inner loop: 4 * 2 = 8
Outer loop: 3
  Inner loop: 3 * 1 = 3
  Inner loop: 3 * 2 = 6
Outer loop: 2
  Inner loop: 2 * 1 = 2
  Inner loop: 2 * 2 = 4
Outer loop: 1
  Inner loop: 1 * 1 = 1
  Inner loop: 1 * 2 = 2
Outer loop: 0
  Inner loop: 0 * 1 = 0
  Inner loop: 0 * 2 = 0
$

```

Again, the shell distinguished between the `do` and `done` commands of the inner `for` loop from the same commands in the outer `while` loop.

If you really want to test your brain, you can even combine `until` and `while` loops:

```

$ cat test16
#!/bin/bash
# using until and while loops

var1=3

until [ $var1 -eq 0 ]
do
  echo "Outer loop: $var1"
  var2=1
  while [ $var2 -lt 5 ]
  do
    var3=$(echo "scale=4; $var1 / $var2" | bc)
    echo "    Inner loop: $var1 / $var2 = $var3"
    var2=$(( $var2 + 1 ))
  done
  var1=$(( $var1 - 1 ))
done
$ ./test16
Outer loop: 3
  Inner loop: 3 / 1 = 3.0000
  Inner loop: 3 / 2 = 1.5000
  Inner loop: 3 / 3 = 1.0000
  Inner loop: 3 / 4 = .7500

```

```
Outer loop: 2
  Inner loop: 2 / 1 = 2.0000
  Inner loop: 2 / 2 = 1.0000
  Inner loop: 2 / 3 = .6666
  Inner loop: 2 / 4 = .5000
Outer loop: 1
  Inner loop: 1 / 1 = 1.0000
  Inner loop: 1 / 2 = .5000
  Inner loop: 1 / 3 = .3333
  Inner loop: 1 / 4 = .2500
$
```

The outer `until` loop starts with a value of 3 and continues until the value equals 0. The inner `while` loop starts with a value of 1 and continues as long as the value is less than 5. Each loop must change the value used in the test condition, or the loop will get stuck infinitely.

## Looping on File Data

---

Often, you must iterate through items stored inside a file. This requires combining two of the techniques covered:

- Using nested loops
- Changing the `IFS` environment variable

By changing the `IFS` environment variable, you can force the `for` command to handle each line in the file as a separate item for processing, even if the data contains spaces. After you've extracted an individual line in the file, you may have to loop again to extract data contained within it.

The classic example of this is processing data in the `/etc/passwd` file. This requires that you iterate through the `/etc/passwd` file line by line and then change the `IFS` variable value to a colon so you can separate the individual components in each line.

The following is an example of doing just that:

```
#!/bin/bash
# changing the IFS value

IFS.OLD=$IFS
IFS=$'\n'
for entry in $(cat /etc/passwd)
do
    echo "Values in $entry -"
    IFS=:
    for value in $entry
```

```

do
    echo "    $value"
done
done
$

```

This script uses two different IFS values to parse the data. The first IFS value parses the individual lines in the `/etc/passwd` file. The inner `for` loop next changes the IFS value to the colon, which allows you to parse the individual values within the `/etc/passwd` lines.

When you run this script, you get output something like this:

```

Values in rich:x:501:501:Rich Blum:/home/rich:/bin/bash -
rich
x
501
501
Rich Blum
/home/rich
/bin/bash
Values in katie:x:502:502:Katie Blum:/home/katie:/bin/bash -
katie
x
506
509
Katie Blum
/home/katie
/bin/bash

```

The inner loop parses each individual value in the `/etc/passwd` entry. This is also a great way to process comma-separated data, a common way to import spreadsheet data.

## Controlling the Loop

You might be tempted to think that after you start a loop, you're stuck until the loop finishes all its iterations. This is not true. A couple of commands help us control what happens inside of a loop:

- The `break` command
- The `continue` command

Each command has a different use in how to control the operation of a loop. The following sections describe how you can use these commands to control the operation of your loops.

### The break command

The `break` command is a simple way to escape a loop in progress. You can use the `break` command to exit any type of loop, including `while` and `until` loops.

You can use the `break` command in several situations. This section shows each of these methods.

#### Breaking out of a single loop

When the shell executes a `break` command, it attempts to break out of the loop that's currently processing:

```
$ cat test17
#!/bin/bash
# breaking out of a for loop

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed"
$ ./test17
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
$
```

The `for` loop should normally have iterated through all the values specified in the list. However, when the `if-then` condition was satisfied, the shell executed the `break` command, which stopped the `for` loop.

This technique also works for `while` and `until` loops:

```
$ cat test18
#!/bin/bash
# breaking out of a while loop

var1=1

while [ $var1 -lt 10 ]
do
    if [ $var1 -eq 5 ]
```

```

        then
            break
        fi
        echo "Iteration: $var1"
        var1=$(( $var1 + 1 ))
    done
    echo "The while loop is completed"
$ ./test18
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
The while loop is completed
$

```

The while loop terminated when the if-then condition was met, executing the break command.

### Breaking out of an inner loop

When you're working with multiple loops, the break command automatically terminates the innermost loop you're in:

```

$ cat test19
#!/bin/bash
# breaking out of an inner loop

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -eq 5 ]
        then
            break
        fi
        echo "    Inner loop: $b"
    done
done
$ ./test19
Outer loop: 1
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
Outer loop: 2
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3

```

```
        Inner loop: 4
Outer loop: 3
        Inner loop: 1
        Inner loop: 2
        Inner loop: 3
        Inner loop: 4
$
```

The `for` statement in the inner loop specifies to iterate until the `b` variable is equal to 100. However, the `if-then` statement in the inner loop specifies that when the `b` variable value is equal to 5, the `break` command is executed. Notice that even though the inner loop is terminated with the `break` command, the outer loop continues working as specified.

### Breaking out of an outer loop

There may be times when you're in an inner loop but need to stop the outer loop. The `break` command includes a single command line parameter value:

```
break n
```

where *n* indicates the level of the loop to break out of. By default, *n* is 1, indicating to break out of the current loop. If you set *n* to a value of 2, the `break` command stops the next level of the outer loop:

```
$ cat test20
#!/bin/bash
# breaking out of an outer loop

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -gt 4 ]
        then
            break 2
        fi
        echo "    Inner loop: $b"
    done
done
$ ./test20
Outer loop: 1
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
$
```

Now when the shell executes the `break` command, the outer loop stops.



## The continue command

The `continue` command is a way to prematurely stop processing commands inside of a loop but not terminate the loop completely. This allows you to set conditions within a loop where the shell won't execute commands. Here's a simple example of using the `continue` command in a `for` loop:

```
$ cat test21
#!/bin/bash
# using the continue command

for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteration number: $var1"
done
$ ./test21
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
$
```

When the conditions of the `if-then` statement are met (the value is greater than 5 and less than 10), the shell executes the `continue` command, which skips the rest of the commands in the loop, but keeps the loop going. When the `if-then` condition is no longer met, things return to normal.

You can use the `continue` command in `while` and `until` loops, but be extremely careful with what you're doing. Remember that when the shell executes the `continue` command, it skips the remaining commands. If you're incrementing your test condition variable in one of those conditions, bad things happen:

```
$ cat badtest3
#!/bin/bash
# improperly using the continue command in a while loop

var1=0

while echo "while iteration: $var1"
```

want to make sure you redirect the output of this script to the `more` command so you can see things. Everything seems to be going just fine until the `if-then` condition is met, and the shell executes the `continue` command. When the shell executes the `continue` command, it skips the remaining commands in the `while` loop. Unfortunately, that's where the `var1` counter variable that is tested in the `while` test command is incremented. That means that the variable isn't incremented, as you can see from the continually displaying of `1`.

```
continue n
```

where *n* defines the loop level to continue. Here's an example of continuing an outer for loop:

```
$ cat test22
#!/bin/bash
# continuing an outer loop

for (( a = 1; a <= 5; a++ ))
do
    echo "Iteration $a:"
    for (( b = 1; b < 3; b++ ))
    do
        if [ $a -gt 2 ] && [ $a -lt 4 ]
        then
            continue 2
        fi
        var3=$(( $a * $b ))
        echo "    The result of $a * $b is $var3"
    done
done
$ ./test22
Iteration 1:
    The result of 1 * 1 is 1
    The result of 1 * 2 is 2
Iteration 2:
    The result of 2 * 1 is 2
    The result of 2 * 2 is 4
Iteration 3:
Iteration 4:
    The result of 4 * 1 is 4
    The result of 4 * 2 is 8
Iteration 5:
    The result of 5 * 1 is 5
    The result of 5 * 2 is 10
$
```

The if-then statement:

```
if [ $a -gt 2 ] && [ $a -lt 4 ]
then
    continue 2
fi
```

uses the `continue` command to stop processing the commands inside the loop but continue the outer loop. Notice in the script output that the iteration for the value 3 doesn't process any inner loop statements, because the `continue` command stopped the processing, but it continues with the outer loop processing.

# Processing the Output of a Loop

---

Finally, you can either pipe or redirect the output of a loop within your shell script. You do this by adding the processing command to the end of the `done` command:

```
for file in /home/rich/*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif
        echo "$file is a file"
    fi
done > output.txt
```

Instead of displaying the results on the monitor, the shell redirects the results of the `for` command to the file `output.txt`.

Consider the following example of redirecting the output of a `for` command to a file:

```
$ cat test23
#!/bin/bash
# redirecting the for output to a file

for (( a = 1; a < 10; a++ ))
do
    echo "The number is $a"
done > test23.txt
echo "The command is finished."
$ ./test23
The command is finished.
$ cat test23.txt
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
$
```

The shell creates the file `test23.txt` and redirects the output of the `for` command only to the file. The shell displays the `echo` statement after the `for` command just as normal.

This same technique also works for piping the output of a loop to another command:

```
$ cat test24
#!/bin/bash
```

```
# piping a loop to another command

for state in "North Dakota" Connecticut Illinois Alabama Tennessee
do
    echo "$state is the next place to go"
done | sort
echo "This completes our travels"
$ ./test24
Alabama is the next place to go
Connecticut is the next place to go
Illinois is the next place to go
North Dakota is the next place to go
Tennessee is the next place to go
This completes our travels
$
```

The state values aren't listed in any particular order in the `for` command list. The output of the `for` command is piped to the `sort` command, which changes the order of the `for` command output. Running the script indeed shows that the output was properly sorted within the script.

## Practical Examples

Now that you've seen how to use the different ways to create loops in shell scripts, let's look at some practical examples of how to use them. Looping is a common way to iterate through data on the system, whether it's files in folders or data contained in a file. Here are a couple of examples that demonstrate using simple loops to work with data.

### Finding executable files

When you run a program from the command line, the Linux system searches a series of folders looking for that file. Those folders are defined in the `PATH` environment variable. If you want to find out just what executable files are available on your system for you to use, just scan all the folders in the `PATH` environment variable. That may take some time to do manually, but it's a breeze working out a small shell script to do that.

The first step is to create a `for` loop to iterate through the folders stored in the `PATH` environment variable. When you do that, don't forget to set the `IFS` separator character:

```
IFS=:
for folder in $PATH
do
```

Now that you have the individual folders in the `$folder` variable, you can use another `for` loop to iterate through all the files inside that particular folder:

```
for file in $folder/*
do
```

The last step is to check whether the individual files have the executable permission set, which you can do using the `if-then` test feature:

```
if [ -x $file ]
then
    echo "    $file"
fi
```

And there you have it! Putting all the pieces together into a script looks like this:

```
$ cat test25
#!/bin/bash
# finding files in the PATH

IFS=:
for folder in $PATH
do
    echo "$folder:"
    for file in $folder/*
    do
        if [ -x $file ]
        then
            echo "    $file"
        fi
    done
done
$
```

When you run the code, you get a listing of the executable files that you can use from the command line:

```
$ ./test25 | more
/usr/local/bin:
/usr/bin:
    /usr/bin/Mail
    /usr/bin/Thunar
    /usr/bin/X
    /usr/bin/Xorg
    /usr/bin/[
    /usr/bin/a2p
    /usr/bin/abiword
    /usr/bin/ac
    /usr/bin/activation-client
    /usr/bin/addr2line
...
```

The output shows all the executable files found in all the folders defined in the `PATH` environment variable, which is quite a few!

## Creating multiple user accounts

The goal of shell scripts is to make life easier for the system administrator. If you happen to work in an environment with lots of users, one of the most boring tasks can be creating new user accounts. Fortunately, you can use the `while` loop to make your job a little easier!

Instead of having to manually enter `useradd` commands for every new user account you need to create, you can place the new user accounts in a text file and create a simple shell script to do that work for you. The format of the text file that we'll use looks like this:

```
userid,user name
```

The first entry is the `userid` you want to use for the new user account. The second entry is the full name of the user. The two values are separated by a comma, making this a comma-separated file format, or `.csv`. This is a very common file format used in spreadsheets, so you can easily create the user account list in a spreadsheet program and save it in `.csv` format for your shell script to read and process.

To read the file data, we're going to use a little shell scripting trick. We'll actually set the `IFS` separator character to a comma as the test part of the `while` statement. Then to read the individual lines, we'll use the `read` command. That looks like this:

```
while IFS=',' read -r userid name
```

The `read` command does the work of moving onto the next line of text in the `.csv` text file, so we don't need another loop to do that. The `while` command exits when the `read` command returns a `FALSE` value, which happens when it runs out of lines to read in the file. Tricky!

To feed the data from the file into the `while` command, you just use a redirection symbol at the end of the `while` command.

Putting everything together results in this script:

```
$ cat test26
#!/bin/bash
# process new user accounts

input="users.csv"
while IFS=',' read -r userid name
do
    echo "adding $userid"
    useradd -c "$name" -m $userid
done < "$input"
$
```

The `$input` variable points to the data file and is used as the redirect data for the `while` command. The `users.csv` file looks like this:

```
$ cat users.csv
rich,Richard Blum
```

```
christine,Christine Bresnahan
barbara,Barbara Blum
tim,Timothy Bresnahan
$
```

To run the problem, you must be the root user account, because the `useradd` command requires root privileges:

```
# ./test26
adding rich
adding christine
adding barbara
adding tim
#
```

Then by taking a quick look at the `/etc/passwd` file, you can see that the accounts have been created:

```
# tail /etc/passwd
rich:x:1001:1001:Richard Blum:/home/rich:/bin/bash
christine:x:1002:1002:Christine Bresnahan:/home/christine:/bin/bash
barbara:x:1003:1003:Barbara Blum:/home/barbara:/bin/bash
tim:x:1004:1004:Timothy Bresnahan:/home/tim:/bin/bash
#
```

Congratulations, you've saved yourself lots of time in adding user accounts!

## Summary

---

Looping is an integral part of programming. The bash shell provides three looping commands that you can use in your scripts.

The `for` command allows you to iterate through a list of values, either supplied within the command line, contained in a variable, or obtained by using file globbing, to extract file and directory names from a wildcard character.

The `while` command provides a method to loop based on the condition of a command, using either ordinary commands or the `test` command, which allows you to test conditions of variables. As long as the command (or condition) produces a zero exit status, the `while` loop continues to iterate through the specified set of commands.

The `until` command also provides a method to iterate through commands, but it bases its iterations on a command (or condition) producing a non-zero exit status. This feature allows you to set a condition that must be met before the iteration stops.



You can combine loops in shell scripts, producing multiple layers of loops. The bash shell provides the `continue` and `break` commands, which allow you to alter the flow of the normal loop process based on different values within the loop.

The bash shell also allows you to use standard command redirection and piping to alter the output of a loop. You can use redirection to redirect the output of a loop to a file or piping to redirect the output of a loop to another command. This provides a wealth of features with which you can control your shell script execution.

The next chapter discusses how to interact with your shell script user. Often, shell scripts aren't completely self-contained. They require some sort of external data that must be supplied at the time you run them. The next chapter discusses different methods with which you can provide real-time data to your shell scripts for processing.