MARUDHARKESARIJAINCOLLEGEFORWOMEN,VANIYAMBADI PG

DEPARTMENT OF COMPUTER APPLICATIONS

CLASS :I MCA

SUBJECT CODE :23PCA12

SUBJECTNAME:LINUXANDSHELLPROGRAMMING

SYLLABUS

Unit-II

More Structured Commands: Looping with for statement-Iterating with the until statement-Using the while statement-Combining loops-Redirecting loop output. **Handling User Input:** Passing parameters-Tracking parameters-Being shifty-Working with options-Standardizing options-Getting user input. **Script Control:** Handling signals-Running scripts in the background-Forbidding hang-ups -Controlling a Job-Modifying script priority-Automating script execution.

(Book-1,Chapters:13,14,and16)

MoreStructuredCommands

INTHISCHAPTER

Looping with the for statement

Iterating with the until statement

Using the while statement

Combiningloops

Redirectingloopoutput

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 conditionhas been met. This chapter discusses and demonstrates the `for`, `while`, and `until`bash shell looping commands.

TheforCommand

Iterating through a series of commands is a common programming practice. Often, you need to repeatasetofcommandsuntilaspecificconditionhasbeenmet,suchasprocessingallthefilesin adirectory,alltheusersonasystem,orallthelinesinatextfile.

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:

```
forvarinlist
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*enteredbetweenthe`do`and`done`statementscanbeoneormorestandard bash shell commands. Within the commands, the `$var`variable contains the current list item value for the iteration.

NOTE
Ifyouprefer,youcanincludethe`do`statementonthesamelineasthe`for`statement,butyoumustseparateit fromthelistitemsusingasemicolon:`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.

Readingvaluesinalist

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

```
$cattest1
#!/bin/bash
#basicforcommand

fortestinAlabamaAlaskaArizonaArkansasCaliforniaColorado do
   echoThenextstateis$test done
$./test1
ThenextstateisAlabama The
next state is Alaska
ThenextstateisArizona
ThenextstateisArkansas
ThenextstateisCalifornia
ThenextstateisColorado
$
```

Each timethe`for`commanditeratesthroughthelistofvaluesprovided,itassignsthe `$test`variable the next value in the list. The `$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):

```
$cattest1b #!/bin/bash
```

```
#testingtheforvariableafterthelooping

fortestinAlabamaAlaskaArizonaArkansasCaliforniaColorado do
   echo"Thenextstateis$test"done
echo"Thelaststatewevisitedwas$test"
test=Connecticut
echo"Wait,nowwe'revisiting$test"
$./test1b
ThenextstateisAlabama The
next state is Alaska
ThenextstateisArizona
ThenextstateisArkansas
ThenextstateisCalifornia
ThenextstateisColorado
ThelaststatewevisitedwasColorado
Wait,nowwe'revisitingConnecticut
$
```

The$testvariable retained its value and allowed us to change the value and use it outside of the forcommand loop, as any other variable would.

Readingcomplexvaluesinalist

Thingsaren'talwaysaseasyastheyseemwiththeforloop.Therearetimeswhenyou run into data that causes problems. Here's a classic example of what can cause problems for shell script programmers:

```
$catbadtest1
#!/bin/bash
#anotherexampleofhownottousetheforcommand

fortestinIdon'tknowifthis'llwork do
   echo"word:$test"do
ne
$./badtest1
word:I
word:dontknowifthisll
word:work
$
```

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

Youhavetwowaystosolvethisproblem:

- Usetheescapecharacter(thebackslash)toescapethesinglequotationmark.
- 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:

```
$cattest2
#!/bin/bash
#anotherexampleofhownottousetheforcommand

fortestinIdon\'tknowif"this'll"work do
   echo"word:$test"don
e
$ ./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:

```
$catbadtest2
#!/bin/bash
#anotherexampleofhownottousetheforcommand

fortestinNevadaNewHampshireNewMexicoNewYorkNorthCarolina do
   echo"Nowgoingto$test"don
e
$./badtest1
NowgoingtoNevada
Now going to New
NowgoingtoHampshire
Now going to New
NowgoingtoMexico
Now going to New
Now going to York
```

```
Now going to North
NowgoingtoCarolina
$
```

Oops,that’snot 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:

```
$cattest3
#!/bin/bash
#anexampleofhowtoproperlydefinevalues

fortestinNevada"NewHampshire""NewMexico""NewYork"do
   echo"Nowgoingto$test"don
e
$./test3
NowgoingtoNevada
NowgoingtoNewHampshire
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.

Readingalistfromavariable

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:

```
$cattest4
#!/bin/bash
#usingavariabletoholdthelist

list="AlabamaAlaskaArizonaArkansasColorado"lis
t=$list" Connecticut"

forstatein$list
do
   echo"Haveyouevervisited$state?" done
$./test4
HaveyouevervisitedAlabama?
HaveyouevervisitedAlaska?
HaveyouevervisitedArizona?
```

```
HaveyouevervisitedArkansas?
HaveyouevervisitedColorado?
HaveyouevervisitedConnecticut?
$
```

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 addingtexttotheendofanexistingtextstringstoredinavariable.

Readingvaluesfromacommand

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

```
$cattest5
#!/bin/bash
#readingvaluesfromafile

file="states"

forstatein$(cat$file) do
   echo"Visitbeautiful$state"
done
$catstates
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
VisitbeautifulConnecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
$
```

Thisexampleusesthe`cat`commandinthecommandsubstitutiontodisplaythecontents of the file states. Notice that the states file includes each state on a separate line, not sepa-rated by spaces. The `for`command still iterates through the output of the `cat`commandonelineatatime,assumingthateachstateisonaseparateline.However,thisdoesn'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 lookat in the next section.

NOTE

The`test5`codeexampleassignedthefilenametothevariableusingjustthefilenamewithoutapath.Thisrequires 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 abso-lute or relative) to reference the file location.

Changingthefieldseparator

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:

- Aspace
- Atab
- Anewline

Ifthebashshellseesanyofthesecharactersinthedata,itassumesthatyou'restarting a new data field in the list. When working with data that can contain spaces (such as file-names), 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:

```
$cattest5b #!/bin/bash
 #readingvaluesfromafile

 file="states"

 IFS=$'\n'
```

```
forstatein$(cat$file) do
   echo"Visitbeautiful$state"
done
$./test5b
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
VisitbeautifulConnecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
VisitbeautifulNewHampshire
VisitbeautifulNorthCarolina
$
```

Nowtheshellscriptcanusevaluesinthelistthatcontainspaces.

CAUTION

Whenworkingonlongscripts,it'spossibletochangetheIFSvalueinoneplace,andthenforgetaboutitandassume thedefaultvalueelsewhereinthescript.AsafepracticetogetintoistosavetheoriginalIFSvaluebeforechangingit and then restore it when you're finished.

Thistechniquecanbecodedlikethis:

```
IFS.OLD=$IFS
IFS=$'\n'
<usethenewIFSvalueincode> IFS=$IFS.OLD
```

ThisensuresthattheIFSvalueisreturnedtothedefaultvalueforfutureoperationswithinthescript.

Other excellent applications of the IFS environment variable are possible. Suppose youwant 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 theassignment 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.

Readingadirectoryusingwildcards

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 shellto 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:

```
$cattest6
#!/bin/bash
#iteratethroughallthefilesinadirectory

forfilein/home/rich/test/* do

   if[-d"$file"]
   then
      echo"$fileisadirectory"eli
   f [ -f "$file" ]
   then
      echo"$fileisafile"
   fi
done
$./test6
/home/rich/test/dir1isadirectory
/home/rich/test/myprog.cisafile
/home/rich/test/myprogisafile
/home/rich/test/myscriptisafile
/home/rich/test/newdirisadirectory
/home/rich/test/newfileisafile
/home/rich/test/newfile2isafile
/home/rich/test/testdirisadirectory
/home/rich/test/testingisafile
/home/rich/test/testprogisafile
/home/rich/test/testprog.cisafile
$
```

The`for`commanditeratesthroughtheresultsofthe`/home/rich/test/*`listing. Thecodetestseachentryusingthe`test`command(usingthesquarebracketmethod) toseeifit'sadirectory,usingthe`–d`parameter,orafile,usingthe`–f`parameter(See Chapter 12).

Noticeinthisexamplethatwedidsomethingdifferentinthe`if`statementtests:

```
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 youdon't, you'll get an error if you run into a directory or filename that contains spaces:

```
./test6:line6:[:toomanyarguments
./test6:line9:[:toomanyarguments
```

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

Youcanalsocombineboththedirectorysearchmethodandthelistmethodinthesame `for`statementbylistingaseriesofdirectorywildcardsinthe`for`command:

```
$cattest7
#!/bin/bash
#iteratingthroughmultipledirectories

forfilein/home/rich/.b*/home/rich/badtest do
   if[-d"$file"] then
      echo"$fileisadirectory"elif
   [ -f "$file" ]
   then
      echo"$fileisafile"els
   e
     echo"$filedoesn'texist"fi
done
$./test7
/home/rich/.backup.timestampisafile
/home/rich/.bash_historyisafile
/home/rich/.bash_logoutisafile
/home/rich/.bash_profileisafile
/home/rich/.bashrcisafile
/home/rich/badtestdoesn'texist
$
```

The `for`statement first uses file globbing to iterate through the list of files that result from thewildcardcharacter;thenititeratesthroughthenextfileinthelist.Youcancombine any number of wildcard entries in the list to iterate through.

CAUTION
Noticethatyoucanenteranythinginthelistdata.Evenifthefileordirectorydoesn'texist,the`for`statement attempts to process whatever you place in the list. This can be a problem when working with files and directories. Youhavenowayofknowingifyou'retryingtoiteratethroughanonexistentdirectory:It'salwaysagoodideatotesteach file or directory before trying to process it.

TheC-StyleforCommand

If you've done any programming using the C programming language, you're probably sur- prisedby the way the bash shell uses the `for`command. In the C language, a `for`loop nor- mally 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.

TheClanguageforcommand

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

```
for(i=0;i<10;i++)
{
   printf("Thenextnumberis%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 condi- tion under which the loop will iterate. When the defined condition becomes false, the `for` loopstopsiterations.Thelastsectiondefinestheiterationprocess.Aftereachiteration, the expression defined in the last section is executed. In this example, the `i`variable isincremented by one after each iteration.

Thebash 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((variableassignment;condition;iterationprocess))
```

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++))
```

Noticethatthereareacoupleofthingsthatdon'tfollowthestandardbashshell`for` method:

- Theassignmentofthevariablevaluecancontainspaces.
- Thevariableintheconditionisn'tprecededwithadollarsign.
- Theequationfortheiteration process doesn't use the `expr`command format.

Theshelldeveloperscreatedthis 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'sanexampleofusingtheC-style`for`commandinabashshellprogram:

```
$cattest8
#!/bin/bash
#testingtheC-styleforloop

for((i=1;i<=10;i++)) do
   echo"Thenextnumberis$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(theletter*i*inthisexample).Ineachiteration,the`$i`variablecontainsthevalueassigned 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.

Usingmultiplevariables

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 processfor each variable. Although you can have multiple variables, you can define only one condi- tion in the `for`loop:

```
$cattest9
#!/bin/bash
#multiplevariables

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.

ThewhileCommand

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.

Basicwhileformat

Here'sfheformatofthe`while`command:

```
whiletestcommand
do
 othercommands
done
```

The *`testcommand`*definedinthe`while`commandistheexactsameformatasin`if-then` statements(seeChapter12).Asinthe`if-then`statement,youcanuseanynormal`bash`shell command,or you can use the `test`command to test for conditions, such as variable values.

Thekeytothe`while`commandisthattheexitstatusofthe*`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.

Themost common use of the *`testcommand`*is to use brackets to check a value of a shell variable that's used in the loop commands:

```
$cattest10
#!/bin/bash
```

```
#whilecommandtest

var1=10
while[$var1-gt0] do
   echo$var1
   var1=$[$var1-1]
done
$./test10
10
9
8
7
6
5
4
3
2
1
$
```

The`while`commanddefinesthetestconditiontocheckforeachiteration:

```
while[$var1-gt0]
```

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 mustbe modified, or you'll have an infinite loop. In this example, we use shell arithmetic todecrease the variable value by one:

```
var1=$[$var1-1]
```

The`while`loopstopswhenthetestconditionisnolongertrue.

Usingmultipletestcommands

The`while`commandallowsyoutodefinemultiple testcommandsonthe`while`statement line.Onlytheexitstatusofthelasttestcommandisusedtodeterminewhentheloop stops. This can cause some interesting results if you're not careful. Here's an example of what we mean:

```
$cattest11 #!/bin/bash
#testingamulticommandwhileloop var1=10

whileecho$var1
```

```
      [$var1-ge0]
do
   echo"Thisisinsidetheloop"var1
   =$[ $var1 - 1 ]
done
$./test11
10
Thisisinsidetheloop 9
Thisisinsidetheloop 8
Thisisinsidetheloop 7
Thisisinsidetheloop 6
Thisisinsidetheloop 5
Thisisinsidetheloop 4
Thisisinsidetheloop 3
Thisisinsidetheloop 2
Thisisinsidetheloop 1
Thisisinsidetheloop 0
Thisisinsidetheloop
-1
$
```

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

```
whileecho$var1
      [$var1-ge0]
```

The first test simply displays the current value of the var1variable. The second test uses brackets to determine the value of the var1variable. Inside the loop, an echostatement displays a simple message, indicating that the loop was processed. Notice when you run the example how the output ends:

```
Thisisinsidetheloop
-1
$
```

Thewhileloopexecutedtheechostatementwhenthe var1variable was equal to zero andthendecreasedthevar1variablevalue.Next,thetestcommandswereexecutedfor the next iteration. The echotest command was executed, displaying the value of the var1 variable, which is now less than zero. It's not until the shell executes the testtest command that the whileloop terminates.

This demonstrates that in a multi-command whilestatement, all the test commands are executedineachiteration,includingthelastiterationwhenthelasttestcommandfails. Be careful of this. Another thing to be careful of is how you specify the multiple test com-mands. Note that each test command is on a separate line!

TheuntilCommand

The untilcommand works in exactly the opposite way from the whilecommand. The untilcommand 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 shellexecutes the commands listed in the loop. When the test command returns a zero exit sta- tus, the loop stops.

Asyouwouldexpect,theformatoftheuntilcommandis:

```
untiltestcommands
do
   othercommands
done
```

Similar to the whilecommand, 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.

Thefollowingisanexampleofusingtheuntilcommand:

```
$cattest12 #!/bin/bash
#usingtheuntilcommand

var1=100

until[$var1-eq0] do
   echo$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 withthe`until`command:

```
$cattest13
#!/bin/bash
#usingtheuntilcommand

var1=100

untilecho$var1
      [$var1-eq0]
do
   echoInsidetheloop:$var1
   var1=$[ $var1 - 25 ]
done
$./test13
100
Insidetheloop:100 75
Insidetheloop:75 50
Insidetheloop:50 25
Insidetheloop:25 0
$
```

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

NestingLoops

Aloopstatementcanuseanyothertypeofcommandwithintheloop,includingother 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'sasimpleexampleofnestinga`for`loop insideanother `for`loop:

```
$cattest14
#!/bin/bash
```

```
#nestingforloops

for((a=1;a<=3;a++)) do
   echo"Startingloop$a:"
   for((b=1;b<=3;b++)) do
      echo "   Insideloop:$b"
   done
done
$ ./test14
Startingloop1:
   Insideloop:1
   Insideloop:2
   Insideloop:3
Startingloop2:
   Insideloop:1
   Insideloop:2
   Insideloop:3
Startingloop3:
   Insideloop:1
   Insideloop:2
   Insideloop:3
$
```

The nested loop (also called the *inner loop*) iterates through its values for each iteration oftheouterloop.Noticethatthere'snodifferencebetweenthe`do`and`done`commandsfor thetwo loops. The bash shell knows when the first `done`command is executed that it refers to the inner loop and not the outer loop.

Thesameapplieswhenyoumixloopcommands,suchasplacinga`for`loopinsidea`while` loop:

```
$cattest15 #!/bin/bash
#placingaforloopinsideawhileloop var1=5

while[$var1-ge0] do
   echo"Outerloop:$var1"
   for((var2=1;$var2<3;var2++)) do
      var3=$[$var1*$var2]
      echo"Innerloop:$var1*$var2=$var3"done
   var1=$[$var1-1]
done
$./test15
```

```
  Outerloop:5
   Innerloop: 5 * 1 = 5
   Innerloop: 5 * 2 = 10
  Outerloop:4
   Innerloop: 4 * 1 = 4
   Innerloop: 4 * 2 = 8
  Outerloop:3
   Innerloop: 3 * 1 = 3
   Innerloop: 3 * 2 = 6
  Outerloop:2
   Innerloop: 2 * 1 = 2
   Innerloop: 2 * 2 = 4
  Outerloop:1
   Innerloop: 1 * 1 = 1
   Innerloop: 1 * 2 = 2
  Outerloop:0
   Innerloop: 0 * 1 = 0
   Innerloop: 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.

Ifyoureallywanttotestyourbrain,youcanevencombine`until`and`while`loops:

```
$cattest16
#!/bin/bash
#usinguntilandwhileloops

var1=3

until[$var1-eq0] do
   echo"Outerloop:$var1"var2=1
   while[$var2-lt5] do
      var3=$(echo"scale=4;$var1/$var2"|bc)
      echo "
               Innerloop:$var1/$var2=$var3"var2
      =$[ $var2 + 1 ]
   done
   var1=$[$var1-1]
done
$ ./test16
Outerloop:3
   Inner loop: 3 / 1 = 3.0000
   Inner loop: 3 / 2 = 1.5000
   Inner loop: 3 / 3 = 1.0000
   Inner loop: 3 / 4 = .7500
```

```
Outerloop:2
    Innerloop:  2 / 1 = 2.0000
    Innerloop:  2 / 2 = 1.0000
    Innerloop:  2 / 3 = .6666
    Innerloop:  2 / 4 = .5000
Outerloop:1
   Innerloop:1/1=1.0000
   Innerloop:1/2=.5000
   Innerloop:1/3=.3333
   Innerloop: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 stuckinfinitely.

LoopingonFileData

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

- Usingnestedloops
- Changingthe`IFS`environmentvariable

Bychanging 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. Afteryou've extracted an individual line in the file, you may have to loop again to extract datacontained within it.

Theclassic 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.

Thefollowingisanexampleofdoingjustthat:

```
#!/bin/bash
#changingtheIFSvalue

IFS.OLD=$IFS
IFS=$'\n'
forentryin$(cat/etc/passwd) do
   echo"Valuesin$entry–"
   IFS=:
   forvaluein$entry
```

```
   do
      echo"    $value"
   done
done
$
```

Thisscriptusestwo different IFSvalues to parse the data. The first IFSvalue parses the individuallinesinthe/etc/passwdfile.Theinnerforloopnextchangesthe IFSvalue to the colon, which allows you to parse the individual values within the /etc/passwd lines.

Whenyourunthisscript,yougetoutputsomethinglikethis:

```
Valuesinrich:x:501:501:RichBlum:/home/rich:/bin/bash- rich
   x
   501
   501
   RichBlum
   /home/rich
   /bin/bash
Valuesinkatie:x:502:502:KatieBlum:/home/katie:/bin/bash- katie
   x
   506
   509
   KatieBlum
   /home/katie
   /bin/bash
```

Theinnerloopparses each individual value in the /etc/passwdentry. This is also a great way to process comma-separated data, a common way to import spreadsheet data.

ControllingtheLoop

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:

- Thebreakcommand
- Thecontinuecommand

Each command has a different use in how to control the operation of a loop. The following sectionsdescribehowyoucanusethesecommandstocontroltheoperationofyourloops.

Thebreakcommand

The`break`commandisasimple way to escapea loop in progress. Youcan use the `break` commandtoexitanytypeofloop,including`while`and`until`loops.

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

Breakingoutofasingleloop

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

```
$cattest17 #!/bin/bash
#breakingoutofaforloop

forvar1in12345678910 do
   if[$var1-eq5] then
      break
   fi
   echo"Iterationnumber:$var1"
done
echo"Theforloopiscompleted"
$ ./test17
Iterationnumber:1
Iterationnumber:2
Iterationnumber:3
Iterationnumber:4
Theforloopiscompleted
$
```

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

Thistechniquealsoworksfor`while`and`until`loops:

```
$cattest18 #!/bin/bash
#breakingoutofawhileloop var1=1

while[$var1-lt10] do
   if[$var1-eq5]
```

```
      then
         break
      fi
      echo"Iteration:$var1"var1=$
      [ $var1 + 1 ]
   done
   echo"Thewhileloopiscompleted"
   $ ./test18
   Iteration: 1
   Iteration: 2
   Iteration: 3
   Iteration: 4
   Thewhileloopiscompleted
   $
```

Thewhileloopterminatedwhentheif-thenconditionwasmet,executingthebreak command.

Breakingoutofaninnerloop

Whenyou'reworkingwithmultipleloops,thebreakcommandautomaticallyterminates the innermost loop you're in:

```
$cattest19
#!/bin/bash
#breakingoutofaninnerloop

for((a=1;a<4;a++)) do
   echo"Outerloop:$a"
   for((b=1;b<100;b++)) do
      if[$b-eq5]
      then
         break
      fi
      echo "   Innerloop:$b"
   done
done
$ ./test19
Outerloop:1
   Innerloop:1
   Innerloop:2
   Innerloop:3
   Innerloop:4
Outerloop:2
   Innerloop:1
   Innerloop:2
   Innerloop:3
```

```
   Innerloop:4
Outerloop:3
   Innerloop:1
   Innerloop:2
   Innerloop:3
   Innerloop: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.

Breakingoutofanouterloop

Theremaybetimeswhenyou'reinaninnerloopbutneedtostoptheouterloop.The `break`commandincludesasinglecommandlineparametervalue:

```
breakn
```

where *n* indicates the level of the loop to break out of. By default, *n* is 1, indicating tobreak 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:

```
$cattest20 #!/bin/bash
#breakingoutofanouterloop

for((a=1;a<4;a++)) do
   echo"Outerloop:$a"
   for((b=1;b<100;b++)) do
      if[$b-gt4] then
         break2
      fi
      echo "   Innerloop:$b"
   done
done
$ ./test20
Outerloop:1
   Innerloop:1
   Innerloop:2
   Innerloop:3
   Innerloop:4
$
```

Nowwhentheshellexecutesthe`break`command,the outerloopstops.

Thecontinuecommand

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:

```
$cattest21
#!/bin/bash
#usingthecontinuecommand

for((var1=1;var1<15;var1++)) do
   if[$var1-gt5]&&[$var1-lt10] then
      continue
   fi
   echo"Iterationnumber:$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 andless 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 longermet, things return to normal.

Youcanusethe`continue`commandin`while`and`until`loops,but be extremely careful withwhat 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 inone of those conditions, bad things happen:

```
$catbadtest3
#!/bin/bash
#improperlyusingthecontinuecommandinawhileloop var1=0

whileecho"whileiteration:$var1"
```

```
        [$var1-lt15]
do
   if[$var1-gt5]&&[$var1-lt10] then
      continue
   fi
   echo "
         Insideiterationnumber:$var1"va
   r1=$[ $var1 + 1 ]
done
$./badtest3|more
whileiteration:0
   Insideiterationnumber:0
while iteration: 1
   Insideiterationnumber:1
while iteration: 2
   Insideiterationnumber:2
while iteration: 3
   Insideiterationnumber:3
while iteration: 4
   Insideiterationnumber:4
while iteration: 5
   Insideiterationnumber:5
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
$
```

You'llwantto make sure you redirect the output of this script to the `more`command so you can stop 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`countervariablethatistestedinthe`while`testcommandisincremented.That means that the variable isn't incremented, as you can see from the continually displaying output.

Aswiththe `break`command, the `continue`command allows you to specify what level of loop to continue with a command line parameter:

```
continuen
```

wherendefinesthelooplevelto continue.Here'sanexampleofcontinuinganouter`for` loop:

```
$cattest22
#!/bin/bash
#continuinganouterloop

for((a=1;a<=5;a++)) do
   echo"Iteration$a:"
   for((b=1;b<3;b++)) do
      if[$a-gt2]&&[$a-lt4] then
         continue2
      fi
      var3=$[$a*$b]
      echo "   Theresultof$a*$bis$var3"done
done
$ ./test22
Iteration1:
   Theresult of 1 * 1 is 1
   Theresult of 1 * 2 is 2
Iteration2:
   Theresultof2*1is2
   Theresult of 2 * 2 is 4
Iteration3:
Iteration4:
   Theresult of 4 * 1 is 4
   Theresult of 4 * 2 is 8
Iteration5:
   Theresultof5*1is5
   Theresult of 5 * 2 is 10
$
```

The`if-then`statement:

```
if[$a-gt2]&&[$a-lt4] then
         continue2
      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'tprocessanyinnerloopstatements,becausethe`continue`commandstoppedtheprocess - ing, but it continues with the outer loop processing.

ProcessingtheOutputofaLoop

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:

```
forfilein/home/rich/* do
   if[-d"$file"] then
      echo"$fileisadirectory" elif
      echo"$fileisafile"
   fi
done>output.txt
```

Insteadofdisplayingtheresultsonthemonitor,theshellredirectstheresultsofthe`for` commandtothefile`output.txt`.

Considerthefollowingexampleofredirectingtheoutputofa`for`commandtoafile:

```
$cattest23 #!/bin/bash
#redirectingtheforoutputtoafile

for((a=1;a<10;a++)) do
   echo"Thenumberis$a"don
e > test23.txt
echo"Thecommandisfinished."
$./test23
Thecommandisfinished.
$cattest23.txt
Thenumberis1
Thenumberis2
Thenumberis3
Thenumberis4
Thenumberis5
Thenumberis6
Thenumberis7
Thenumberis8
Thenumberis9
$
```

Theshellcreatesthe file `test23.txt`and redirects the output of the `for`command only tothefile.Theshelldisplaysthe`echo`statementafterthe`for`commandjustasnormal.

Thissametechniquealsoworksforpipingtheoutputofalooptoanothercommand:

```
$cattest24 #!/bin/bash
```

```
#pipingalooptoanothercommand

forstatein"NorthDakota"ConnecticutIllinoisAlabamaTennessee do
   echo"$stateisthenextplacetogo"done |
sort
echo"Thiscompletesourtravels"
$./test24
Alabamaisthenextplacetogo
Connecticutisthenextplacetogo
Illinoisisthenextplacetogo
NorthDakotaisthenextplacetogo
Tennesseeisthenextplacetogo
Thiscompletesourtravels
$
```

The state values aren't listed in any particular order in the `for`command list. The outputofthe `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.

PracticalExamples

Nowthatyou'veseenhowtousethedifferentwaystocreateloopsinshellscripts,let's look at some practical examples of how to use them. Looping is a common way to iterate throughdataonthesystem,whetherit'sfilesinfoldersordatacontainedinafile.Hereare a couple of examples that demonstrate using simple loops to work with data.

Findingexecutablefiles

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.

Thefirst 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=:
forfolderin$PATH
do
```

Nowthatyouhavetheindividualfoldersinthe`$folder`variable,youcanuseanother `for`looptoiteratethroughallthefilesinsidethatparticularfolder:

```
forfilein$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
```

Andthereyouhaveit!Puttingallthepiecestogetherintoascriptlookslikethis:

```
$cattest25
#!/bin/bash
#findingfilesinthePATH

IFS=:
forfolderin$PATH do
   echo"$folder:"
   forfilein$folder/* do
      if[-x$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 thecommand 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!

Creatingmultipleuseraccounts

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 whileloop to make your job a little easier!

Instead of having to manually enter useraddcommands 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 scripttodothatworkforyou.Theformatofthetextfilethatwe'lluselookslikethis:

```
userid,username
```

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, soyou can easily create the user account list in a spreadsheet program and save it in .csv for- mat 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 theIFSseparatorcharacter to a comma as the test part of the whilestatement. Then to read the individual lines, we'll use the readcommand. That looks like this:

```
whileIFS=','read-ruseridname
```

The readcommand does the work of moving onto the next line of text in the .csv text file,sowedon'tneedanotherlooptodothat.Thewhilecommandexitswhenthereadcommand returns a FALSEvalue, which happens when it runs out of lines to read in the file. Tricky!

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

Puttingeverythingtogetherresultsinthisscript:

```
$cattest26
#!/bin/bash
#processnewuseraccounts

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

The $inputvariablepointstothedatafileandisusedastheredirectdataforthewhile command.Theusers.csvfilelookslikethis:

```
$ cat users.csv
rich,RichardBlum
```

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

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

```
# ./test26
adding rich
addingchristine
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'vesavedyourselflotsoftimeinaddinguseraccounts!

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 commandline,containedinavariable,orobtainedbyusingfileglobbing,toextractfile 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 conditionsof 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 providesthe`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 outputofaloop.Youcanuseredirectiontoredirecttheoutputofalooptoafileorpiping to redirect the output of a loop to another command. This provides a wealth of featureswith 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 whichyou can provide real-time data to your shell scripts for processing.

HandlingUserInput

INTHISCHAPTER

Passing parameters

Tracking parameters

Beingshifty

Working with options

Standardizingoptions

Getting user input

So far you've seen how to write scripts that interact with data, variables, and files on theLinux system. Sometimes, you need to write a script that has to interact with the person running the script. The bash shell provides a few different methods for retrieving data from people, including command line parameters (data values added after the command), command line options (single-letter values that modify the behavior of the command), and the capability to read input directly from the keyboard. This chapter discusses how to incorporate these different methods into your bash shell scripts to obtain data from the person running your script.

PassingParameters

The most basic method of passing data to your shell script is to use *command line parameters*. Command line parameters allow you to add data values to the command line when you execute the script:

```
$./addem1030
```

This example passes two command line parameters (`10`and `30`) to the script `addem`. The script handles the command line parameters using special variables. The following sections describe how to use command line parameters in your bash shell scripts.

Readingparameters

The bash shell assigns special variables, called *positional parameters,* to all of the command line parameters entered. This includes the name of the script the shell is executing. The positional parameter variables are standard numbers, with *$0*being the script's name, *$1* being the first parameter, *$2*being the second parameter, and so on, up to *$9*for the ninth parameter.

Here'sasimpleexampleofusingonecommandlineparameterinashellscript:

```
$cattest1.sh
#!/bin/bash
#usingonecommandlineparameter #
factorial=1
for((number=1;number<=$1;number++)) do
   factorial=$[$factorial*$number] done
echoThefactorialof$1is$factorial
$
$./test1.sh5
Thefactorialof5is120
$
```

You can use the *$1*variable just like any other variable in the shell script. The shell script automatically assigns the value from the command line parameter to the variable; you don't need to do anything with it.

If you need to enter more command line parameters, each parameter must be separated by a space on the command line:

```
$cattest2.sh
#!/bin/bash
#testingtwocommandlineparameters #
total=$[$1*$2]
echoThefirstparameteris$1.
echoThesecondparameteris$2.
echoThetotalvalueis$total.
$
$./test2.sh25
Thefirstparameteris2.
Thesecondparameteris5.
The total value is 10.
$
```

Theshell assigns each parameter to the appropriate variable.

In the preceding example, the command line parameters used were both numerical values. You can also use text strings in the command line:

```
$cattest3.sh
#!/bin/bash
#testingstringparameters #
echoHello$1,gladtomeetyou.
$
$./test3.shRich
HelloRich,gladtomeetyou.
$
```

The shell passes the string value entered into the command line to the script. However, you'llhaveaproblemifyoutrytodothiswithatextstringthatcontainsspaces:

```
$./test3.shRichBlum
HelloRich,gladtomeetyou.
$
```

Remember that each of the parameters is separated by a space, so the shell interpreted the space as just separating the two values. To include a space as a parameter value, you must use quotation marks (either single or double quotation marks):

```
$./test3.sh'RichBlum'
HelloRichBlum,gladtomeetyou.
$
$./test3.sh"RichBlum"
HelloRichBlum,gladtomeetyou.
$
```

NOTE

Thequotationmarksusedwhenyoupasstextstringsasparametersarenotpartofthedata.Theyjustdelineatethe beginningandtheendofthedata.

If your script needs more than nine command line parameters, you can continue, but the variable names change slightly. After the ninth variable, you must use braces around the variable number, such as *${10}*. Here's an example of doing that:

```
$cattest4.sh
#!/bin/bash
#handlinglotsofparameters #
total=$[${10}*${11}]
echoThetenthparameteris${10}
echoTheeleventhparameteris${11} echo
The total is $total
```

```
$
$./test4.sh123456789101112
The tenth parameter is 10
Theeleventhparameteris11 The
total is 110
$
```

This technique allows you to add as many command line parameters to your scripts as you could possibly need.

Readingthescriptname

You can use the *$0*parameter to determine the script name the shell started from the command line. This can come in handy if you're writing a utility that can have multiplefunctions.

```
$cattest5.sh
#!/bin/bash
#Testingthe$0parameter #
echoThezeroparameterissetto:$0 #
$
$bashtest5.sh
Thezeroparameterissetto:test5.sh
$
```

However, there is a potential problem. When using a different command to run the shell script, the command becomes entangled with the script name in the `$0`parameter:

```
$./test5.sh
Thezeroparameterissetto:./test5.sh
$
```

There is another potential problem. When the actual string passed is the full script path, andnotjustthescript'sname,the`$0`variablegetssettothefullscriptpathandname:

```
$bash/home/Christine/test5.sh
Thezeroparameterissetto:/home/Christine/test5.sh
$
```

If you want to write a script that performs different functions based on just the script'sname, you'll have to do a little work. You need to be able to strip off whatever path is used to run the script. Also, you need to be able to remove any entangled commands from the script.

Fortunately,there'sahandylittlecommandavailablethatdoesjustthat.The`basename` commandreturnsjustthescript'snamewithoutthepath:

```
$cattest5b.sh
#!/bin/bash
```

```
#Usingbasenamewiththe$0parameter #
name=$(basename$0)
echo
echoThescriptnameis:$name #
$bash/home/Christine/test5b.sh

Thescriptnameis:test5b.sh
$
$./test5b.sh

Thescriptnameis:test5b.sh
$
```

Now that's much better. You can use this technique to write scripts that perform different functions based on the script name used. Here's a simple example:

```
$cattest6.sh
#!/bin/bash
#TestingaMulti-functionscript #
name=$(basename$0)
#
if[$name="addem"]
then
   total=$[$1+$2]
#
elif[$name="multem"] then
   total=$[$1*$2]
fi
#
echo
echoThecalculatedvalueis$total #
$
$cptest6.shaddem
$chmodu+xaddem
$
$ln-stest6.shmultem
$
$ls-l*em
-rwxrw-r--.1ChristineChristine224Jun3023:50addem
lrwxrwxrwx.1ChristineChristine      8Jun3023:50multem->test6.sh
$
$./addem25

Thecalculatedvalueis7
```

```
$
$./multem25

Thecalculatedvalueis10
$
```

Theexamplecreates two separate filenames from the test6.shscript, one by just copying the file to a new script (addem) and the other by using a symbolic link (see Chapter 3) to create the new script (multem). In both cases, the script determines the script's base name and performs the appropriate function based on that value.

Testingparameters

Be careful when using command line parameters in your shell scripts. If the script is run without the parameters, bad things can happen:

```
$./addem2
./addem:line8:2+:syntaxerror:operandexpected(error token is
 "")
Thecalculatedvalueis
$
```

Whenthescriptassumesthereisdatainaparametervariable,andnodataispresent, most likely you'll get an error message from your script. This is a poor way to write scripts. Always check your parameters to make sure the data is there before using it:

```
$cattest7.sh
#!/bin/bash
#testingparametersbeforeuse #
if[-n"$1"] then
   echoHello$1,gladtomeetyou. else
   echo"Sorry,youdidnotidentifyyourself."
fi
$
$./test7.shRich
HelloRich,gladtomeetyou.
$
$./test7.sh
Sorry,youdidnotidentifyyourself.
$
```

In this example, the -ntest evaluation was used to check for data in the $1command line parameter. In the next section, you'll learn another way to check command line parameters.

UsingSpecialParameterVariables

A few special bash shell variables track command line parameters. This section describes what they are and how to use them.

Countingparameters

As you saw in the last section, you should verify command line parameters before using them in your script. For scripts that use multiple command line parameters, this checking can get tedious.

Instead of testing each parameter, you can count how many parameters were entered on the command line. The bash shell provides a special variable for this purpose.

The special $#variable contains the number of command line parameters included whenthe script was run. You can use this special variable anywhere in the script, just like a nor-mal variable:

```
$cattest8.sh
#!/bin/bash
#gettingthenumberofparameters #
echoTherewere$#parameterssupplied.
$
$./test8.sh
Therewere0parameterssupplied.
$
$./test8.sh12345
Therewere5parameterssupplied.
$
$./test8.sh12345678910
Therewere10parameterssupplied.
$
$./test8.sh"RichBlum"
Therewere1parameterssupplied.
$
```

Now you have the ability to test the number of parameters present before trying to use them:

```
$cattest9.sh
#!/bin/bash
#Testingparameters
#
if[$#-ne2] then
   echo
```

```
   echoUsage:test9.shab
   echo
else
   total=$[$1+$2]
   echo
   echoThetotalis$total
   echo
fi
#
$
$bashtest9.sh

Usage:test9.shab

$bashtest9.sh10

Usage:test9.shab

$bashtest9.sh1015

Thetotalis25

$
```

The`if-then`statementusesthe `-ne`evaluation to perform a numeric test of the command line parameters supplied. If the correct number of parameters isn't present, an error message displays showing the correct usage of the script.

This variable also provides a cool way of grabbing the last parameter on the command line without having to know how many parameters were used. However, you need to use a little trick to get there.

Ifyouthinkthisthrough,youmightthinkthatbecausethe *$#*variablecontainsthevalue of the number of parameters, using the variable *${$#}*would represent the last command line parameter variable. Try that and see what happens:

```
$catbadtest1.sh
#!/bin/bash
#testinggrabbinglastparameter #
echoThelastparameterwas${$#}
$
$./badtest1.sh10
Thelastparameterwas15354
$
```

Wow, what happened? Obviously, something went wrong. It turns out that you can't use the dollar sign within the braces. Instead, you must replace the dollar sign with an exclamation mark. Odd, but it works:

```
$cattest10.sh
#!/bin/bash
#Grabbingthelastparameter #
params=$#
echo
echoThelastparameteris$params
echoThelastparameteris${!#} echo
#
$
$bashtest10.sh12345

Thelastparameteris5
Thelastparameteris5

$
$bashtest10.sh

Thelastparameteris0
Thelastparameteristest10.sh

$
```

Perfect.Thisscriptalsoassignedthe *$#*variablevaluetothevariable*params*andthen used that variable within the special command line parameter variable format as well. Both versions worked. It's also important to notice that, when there weren't any parameters onthecommandline,the *$#*valuewaszero,whichiswhatappearsinthe`params`variable, but the *${!#}*variable returns the script name used on the command line.

Grabbingallthedata

In some situations you want to grab all the parameters provided on the command line. Insteadof having to mess with using the *$#*variable to determine how many parameters are on the command line and having to loop through all of them, you can use a couple of other special variables.

The *$**and *$@*variables provide easy access to all your parameters. Both of these variables include all the command line parameters within a single variable.

The *$**variable takes all the parameters supplied on the command line as a single word. The word contains each of the values as they appear on the command line. Basically, instead of treating the parameters as multiple objects, the *$**variable treats them all as one parameter.

The *$@*variable, on the other hand, takes all the parameters supplied on the command lineas separate words in the same string. It allows you to iterate through the values, separating out each parameter supplied. This is most often accomplished using the `for`command.

It can easily get confusing to figure out how these two variables operate. Let's look at the difference between the two:

```
$cattest11.sh
#!/bin/bash
#testing$*and$@ #
echo
echo"Usingthe\$*method:$*"echo
echo"Usingthe\$@method:$@"
$
$./test11.shrichbarbarakatiejessica

Usingthe$*method:richbarbarakatiejessica

Usingthe$@method:richbarbarakatiejessica
$
```

Notice that on the surface, both variables produce the same output, showing all the command line parameters provided at once.

Thefollowingexampledemonstrateswherethedifferencesare:

```
$cattest12.sh
#!/bin/bash
#testing$*and$@ #
echo
count=1
#
forparamin"$*"do
   echo"\$*Parameter#$count=$param"count=$[
   $count + 1 ]
done
#
echo
count=1
#
forparamin"$@"do
   echo"\$@Parameter#$count=$param"count=$[
   $count + 1 ]
done
$
$./test12.shrichbarbarakatiejessica

$*Parameter#1=richbarbarakatiejessica

$@Parameter#1=rich
```

```
$@Parameter#2=barbara
$@Parameter#3=katie
$@Parameter#4=jessica
$
```

Now we're getting somewhere. By using the `for`command to iterate through the special variables, you can see how they each treat the command line parameters differently. The *$**variable treated all the parameters as a single parameter, while the *$@*variable treated each parameter separately. This is a great way to iterate through command line parameters.

BeingShifty

Anothertool you have in your bash shell tool belt is the `shift`command. The bash shell provides the `shift`command to help you manipulate command line parameters. The `shift`commandliterallyshiftsthecommandlineparametersintheirrelativepositions.

When you use the `shift`command, it moves each parameter variable one position to the left by default. Thus, the value for variable *$3*is moved to *$2*, the value for variable *$2*is moved to *$1*, and the value for variable *$1*is discarded (note that the value for variable *$0*,theprogramname,remainsunchanged).

This is another great way to iterate through command line parameters, especially if youdon't know how many parameters are available. You can just operate on the first parameter, shift the parameters over, and then operate on the first parameter again.

Here'sa short demonstration ofhow this works:

```
$cattest13.sh
#!/bin/bash
#demonstratingtheshiftcommand
echo
count=1
while[-n"$1"] do
   echo"Parameter#$count=$1"count
   =$[ $count + 1 ]
   shift
done
$
$./test13.shrichbarbarakatiejessica

Parameter #1 = rich
Parameter#2=barbara
Parameter #3 = katie
Parameter#4=jessica
$
```

Thescript performs a `while`loop, testing the length of the first parameter's value. When the first parameter's length is zero, the loop ends. After testing the first parameter, the`shift`command is used to shift all the parameters one position.

TIP

Becarefulwhenworkingwiththe`shift`command.Whenaparameterisshiftedout,itsvalueislostandcan'tbe recovered.

Alternatively,youcanperformamultiplelocationshiftbyprovidingaparametertothe `shift`command.Justprovidethenumberofplacesyouwanttoshift:

```
$cattest14.sh
#!/bin/bash
#demonstratingamulti-positionshift #
echo
echo"Theoriginalparameters:$*"shift
2
echo"Here'sthenewfirstparameter:$1"
$
$./test14.sh12345

Theoriginalparameters:12345
Here'sthenewfirstparameter:3
$
```

Byusing values in the `shift`command, you can easily skip over parameters you don't need.

WorkingwithOptions

If you've been following along in the book, you've seen several bash commands that provide both parameters and options. *Options* are single letters preceded by a dash that alter the behavior of a command. This section shows three methods for working with options in your shell scripts.

Findingyouroptions

On the surface, there's nothing all that special about command line options. They appearon the command line immediately after the script name, just the same as command line parameters. In fact, if you want, you can process command line options the same way you process command line parameters.

Processingsimpleoptions

In the `test13.sh`script earlier, you saw how to use the `shift`command to work your way down the command line parameters provided with the script program. You can use this same technique to process command line options.

Asyou extract each individual parameter, use the `case`statement (see Chapter 12) to determine when a parameter is formatted as an option:

```
$cattest15.sh
#!/bin/bash
#extractingcommandlineoptionsasparameters #
echo
while[-n"$1"] do
   case"$1"in
     -a)echo"Foundthe-aoption";;
     -b)echo"Foundthe-boption";;
     -c)echo"Foundthe-coption";;
      *)echo"$1isnotanoption";; esac
   shift
done
$
$./test15.sh-a-b-c-d

Foundthe-aoption
Foundthe-boption
Foundthe-coption
-disnotanoption
$
```

The`case`statement checks each parameter for valid options. When one is found, the appropriate commands are run in the `case`statement.

This method works, no matter in what order the options are presented on the command line:

```
$./test15.sh-d-c-a

-disnotanoption
Foundthe-coption
Foundthe-aoption
$
```

The`case`statementprocesses each option as it finds it in the command line parameters. If any other parameters are included on the command line, you can include commands in the catch-all part of the `case`statement to process them.

Separatingoptionsfromparameters

Oftenyou'llrunintosituationswhereyou'llwanttousebothoptionsandparametersforashell script.ThestandardwaytodothisinLinuxistoseparatethetwowithaspecialcharactercode thattellsthescriptwhentheoptionsarefinishedandwhenthenormalparametersstart.

For Linux, this special character is the double dash (--). The shell uses the double dash to indicate the end of the option list. After seeing the double dash, your script can safely process the remaining command line parameters as parameters and not options.

Tocheckforthedoubledash,simplyaddanotherentryinthe `case`statement:

```
$cattest16.sh
#!/bin/bash
#extractingoptionsandparameters
echo
while[-n"$1"] do
   case"$1"in
      -a)echo"Foundthe-aoption";;
      -b)echo"Foundthe-boption";;
      -c)echo"Foundthe-coption";;
      --)shift
          break;;
       *)echo"$1isnotanoption";; esac
   shift
done
#
count=1
forparamin$@ do
   echo"Parameter#$count:$param"co
   unt=$[ $count + 1 ]
done
$
```

This script uses the `break`command to break out of the `while`loop when it encounters the double dash. Because we're breaking out prematurely, we need to ensure that we stick in another `shift`command to get the double dash out of the parameter variables.

Forthefirsttest,tryrunningthescriptusinganormalsetofoptionsandparameters:

```
$./test16.sh-c-a-btest1test2test3

Found the -c option
Found the -a option
Found the -b option
test1isnotanoption
```

```
test2isnotanoption
test3isnotanoption
$
```

The results show that the script assumed that all the command line parameters were options when it processed them. Next, try the same thing, only this time using the doubledash to separate the options from the parameters on the command line:

```
$./test16.sh-c-a-b--test1test2test3

Foundthe-coption
Foundthe-aoption
Foundthe-boption
Parameter#1:test1
Parameter#2:test2
Parameter#3:test3
$
```

When the script reaches the double dash, it stops processing options and assumes that any remaining parameters are command line parameters.

Processingoptionswithvalues

Some options require an additional parameter value. In these situations, the command line looks something like this:

```
$./testing.sh-atest1-b-c-dtest2
```

Your script must be able to detect when your command line option requires an additional parameter and be able to process it appropriately. Here's an example of how to do that:

```
$cattest17.sh
#!/bin/bash
#extractingcommandlineoptionsandvalues echo
while[-n"$1"] do
   case"$1"in
      -a)echo"Foundthe-aoption";;
      -b)param="$2"
          echo"Foundthe-boption,withparametervalue$param"shift ;;
      -c)echo"Foundthe-coption";;
      --)shift
          break;;
       *)echo"$1isnotanoption";; esac
   shift
done
```

```
#
count=1
forparamin"$@"do
   echo"Parameter#$count:$param"co
   unt=$[ $count + 1 ]
done
$
$./test17.sh-a-btest1-d

Foundthe-aoption
Foundthe-boption,withparametervaluetest1
-disnotanoption
$
```

In this example, the `case`statement defines three options that it processes. The `-b`option also requires an additional parameter value. Because the parameter being processed is *$1*, you know that the additional parameter value is located in *$2*(because all the parameters are shifted after they are processed). Just extract the parameter value from the *$2*vari-able. Of course, because we used two parameter spots for this option, you also need to set the `shift`command to shift one additional position.

Just as with the basic feature, this process works no matter what order you place the options in (just remember to include the appropriate option parameter with the each option):

```
$./test17.sh-btest1-a-d
Foundthe-boption,withparametervaluetest1 Found
the -a option
-disnotanoption
$
```

Now you have the basic ability to process command line options in your shell scripts, but there are limitations. For example, this doesn't work if you try to combine multiple options in one parameter:

```
$./test17.sh-ac
-acisnotanoption
$
```

It is a common practice in Linux to combine options, and if your script is going to be user-friendly, you'll want to offer this feature for your users as well. Fortunately, there's another method for processing options that can help you.

Usingthegetoptcommand

The`getopt`commandisagreattooltohavehandywhenprocessingcommandlineoptionsand parameters. It reorganizes the command line parameters to make parsing them in your script easier.

Lookingatthecommandformat

The`getopt`commandcan take a list of command line options and parameters, in any form, and automatically turn them into the proper format. It uses the following command format:

```
getoptoptstringparameters
```

The *`optstring`*is the key to the process. It defines the valid option letters that can be used in the command line. It also defines which option letters require a parameter value.

First,listeachcommandlineoptionletteryou'regoingtouseinyourscriptinthe *`optstring`*.Thenplaceacolonaftereachoptionletterthatrequiresaparametervalue. The`getopt`commandparsesthesuppliedparametersbasedonthe*`optstring`*youdefine.

> **TIP**
> A more advanced version of the `getopt`command, called `getopts`(notice it is plural), is available. The `getopts` command is covered later in this chapter. Because of their nearly identical spelling, it's easy to get these two com-mands confused. Be careful!

Here'sasimpleexampleofhow`getopt`works:

```
$getoptab:cd-a-btest1-cdtest2test3
 -a-btest1-c-d--test2test3
$
```

The *`optstring`*defines four valid option letters, `a`, `b`, `c`, and `d`. A colon (:) is placed behind theletter `b`in order to require option `b`to have a parameter value. When the `getopt`command runs, it examines the provided parameter list (`-a-btest1-cdtest2test3`) and parses it based on the supplied *`optstring`*. Notice that it automatically separated the `-cd`optionsinto two separate options and inserted the double dash to separate the additional parameters on the line.

Ifyouspecifyaparameteroptionnotinthe*`optstring`*,bydefaultthe`getopt`command producesanerrormessage:

```
$getoptab:cd-a-btest1-cdetest2test3
getopt:invalidoption--e
 -a-btest1-c-d--test2test3
$
```

Ifyouprefertojustignoretheerrormessages,use`getopt`withthe`-q`option:

```
$getopt-qab:cd-a-btest1-cdetest2test3
 -a-b'test1'-c-d--'test2''test3'
$
```

Notethatthe`getopt`command optionsmustbelistedbeforethe*`optstring`*.Nowyou should be ready to use this command in your scripts to process command line options.

Usinggetoptinyourscripts

Youcan use the getoptcommand in your scripts to format any command line options or parameters entered for your script. It's a little tricky, however, to use.

The trick is to replace the existing command line options and parameters with the formattedversion produced by the getoptcommand. The way to do that is to use the set command.

Yousawthesetcommandback in Chapter 6. The setcommand works with the different variables in the shell.

Oneofthesetcommandoptionsis the double dash (--). The double dash instructs setto replacethecommandline parameter variables with the values on the setcommand's command line.

The trick then is to feed the original script command line parameters to the getoptcommand and then feed the output of the getoptcommand to the setcommand to replace the original command line parameters with the nicely formatted ones from getopt. This looks something like this:

```
set--$(getopt-qab:cd"$@")
```

Now the values of the original command line parameter variables are replaced with the output from the getoptcommand, which formats the command line parameters for us.

Using this technique, we can now write scripts that handle our command line parameters for us:

```
$cattest18.sh
#!/bin/bash
#Extractcommandlineoptions&valueswithgetopt #
set--$(getopt-qab:cd"$@") #
echo
while[-n"$1"] do
   case"$1"in
   -a)echo"Foundthe-aoption";;
   -b)param="$2"
       echo"Foundthe-boption,withparametervalue$param"shift ;;
   -c)echo"Foundthe-coption";;
   --)shift
       break;;
    *)echo"$1isnotanoption";; esac
   shift
```

```
done
#
count=1
forparamin"$@"do
   echo"Parameter#$count:$param"co
   unt=$[ $count + 1 ]
done
#
$
```

You'll notice this is basically the same script as in `test17.sh`. The only thing that changed is the addition of the `getopt`command to help format our command line parameters.

Nowwhenyourunthescriptwithcomplexoptions,thingsworkmuchbetter:

```
$./test18.sh-ac

Foundthe-aoption
Foundthe-coption
$
```

Andofcourse,alltheoriginalfeaturesworkjustfineaswell:

```
$./test18.sh-a-btest1-cdtest2test3test4

Foundthe-aoption
Foundthe-boption,withparametervalue'test1'Found
the -c option
Parameter#1:'test2'
Parameter#2:'test3'
Parameter#3:'test4'
$
```

Nowthingsarelookingprettyfancy.However,there'sstillonesmallbugthatlurksinthe `getopt`command.Checkoutthisexample:

```
$./test18.sh-a-btest1-cd"test2test3"test4

Foundthe-aoption
Foundthe-boption,withparametervalue'test1'Found
the -c option
Parameter#1:'test2
Parameter#2:test3'
Parameter#3:'test4'
$
```

The`getopt`commandisn't good at dealing with parameter values with spaces and quotation marks. It interpreted the space as the parameter separator, instead of following the double quotation marks and combining the two values into one parameter. Fortunately, this problem has another solution.

Advancingtogetopts

The `getopts`command (notice that it is plural) is built into the bash shell. It looks much like its `getopt`cousin, but has some expanded features.

Unlike `getopt`, which produces one output for all the processed options and parameters foundin the command line, the `getopts`command works on the existing shell parameter variables sequentially.

It processes the parameters it detects in the command line one at a time each time it's called. When it runs out of parameters, it exits with an exit status greater than zero. Thismakes it great for using in loops to parse all the parameters on the command line.

Here'stheformatofthe`getopts`command:

```
getoptsoptstringvariable
```

The*optstring*valueis similar to the one used in the `getopt`command. Valid option letters are listed in the *optstring*, along with a colon if the option letter requires a parameter value. To suppress error messages, start the *optstring*with a colon. The `getopts` command places the current parameter in the *variable*defined in the command line.

The*getopts*commandusestwoenvironmentvariables.The*OPTARG*environmentvariable containsthe value to be used if an option requires a parameter value. The *OPTIND*environment variable contains the value of the current location within the parameter list where `getopts`left off. This allows you to continue processing other command line parameters after finishing the options.

Let'slookatasimpleexamplethatusesthe`getopts`command:

```
$cattest19.sh
#!/bin/bash
#simpledemonstrationofthegetoptscommand #
echo
whilegetopts:ab:copt do
   case"$opt"in
      a) echo"Foundthe-aoption";;
      b) echo"Foundthe-boption,withvalue$OPTARG";;
      c) echo"Foundthe-coption";;
      *)echo"Unknownoption:$opt";; esac
done
$
```

```
$./test19.sh-abtest1-c

Foundthe-aoption
Foundthe-boption,withvaluetest1
Found the -c option
$
```

The `while`statement defines the `getopts`command, specifying what command line options to look for, along with the variable name (*`opt`*) to store them in for each iteration.

You'll notice something different about the `case`statement in this example. When the `getopts`command parses the command line options, it strips off the leading dash, so you don't need leading dashes in the `case`definitions.

The`getopts`commandoffers several nice features. For starters, you can include spaces in your parameter values:

```
$./test19.sh-b"test1test2"-a

Foundthe-boption,withvaluetest1test2
Found the -a option
$
```

Another nice feature is that you can run the option letter and the parameter value together without a space:

```
$./test19.sh-abtest1

Foundthe-aoption
Foundthe-boption,withvaluetest1
$
```

The`getopts`commandcorrectlyparsedthe `test1`value from the `–b`option. In addition, the `getopts`command bundles any undefined option it finds in the command line into a single output, the question mark:

```
$./test19.sh-d

Unknownoption:?
$
$./test19.sh-acde

Foundthe-aoption
Foundthe-coption
Unknown option: ?
Unknownoption:?
$
```

Any option letter not defined in the *`optstring`*value is sent to your code as a question mark.

The`getopts`commandknows when to stop processing options and leave the parameters foryoutoprocess.As `getopts`processes each option, it increments the *`OPTIND`*environment variable by one. When you've reached the end of the `getopts`processing, you can use the *`OPTIND`*value with the `shift`command to move to the parameters:

```
$cattest20.sh
#!/bin/bash
#Processingoptions&parameterswithgetopts #
echo
whilegetopts:ab:cdopt do
   case"$opt"in
   a) echo"Foundthe-aoption";;
   b) echo"Foundthe-boption,withvalue$OPTARG";;
   c) echo"Foundthe-coption";;
   d) echo"Foundthe-doption";;
   *)echo"Unknownoption:$opt";; esac
done
#
shift$[$OPTIND-1] #
echo
count=1
forparamin"$@"do
   echo"Parameter$count:$param"co
   unt=$[ $count + 1 ]
done
#
$
$./test20.sh-a-btest1-dtest2test3test4

Foundthe-aoption
Foundthe-boption,withvaluetest1
Found the -d option

Parameter1:test2
Parameter2:test3
Parameter3:test4
$
```

Now you have a full-featured command line option and parameter processing utility you can use in all your shell scripts!

StandardizingOptions

When you create your shell script, obviously you're in control of what happens. It's completely up to you as to which letter options you select to use and how you select to use them.

However,afewletteroptionshaveachievedasomewhatstandardmeaninginthe Linux world. If you leverage these options in your shell script, your scripts will be more user-friendly.

Table14-1showssomeofthecommonmeaningsforcommandlineoptionsusedinLinux.

TABLE14-1 **CommonLinuxCommandLineOptions**

Option	Description
`-a`	Showsallobjects
`-c`	Producesacount
`-d`	Specifiesadirectory
`-e`	Expandsanobject
`-f`	Specifiesafiletoreaddatafrom
`-h`	Displaysahelpmessageforthecommand
`-i`	Ignorestextcase
`-l`	Producesalongformatversionoftheoutput
`-n`	Usesanon-interactive(batch)mode
`-o`	Specifiesanoutputfiletoredirectalloutputto
`-q`	Runsinquietmode
`-r`	Processesdirectoriesandfilesrecursively
`-s`	Runsinsilentmode
`-v`	Producesverboseoutput
`-x`	Excludesanobject
`-y`	Answersyestoallquestions

You'll probably recognize most of these option meanings just from working with the various `bash`commandsthroughoutthebook. Using the same meaning for your options helps users interact with your script without having to worry about manuals.

GettingUserInput

Although providing command line options and parameters is a great way to get data from your script users, sometimes your script needs to be more interactive. Sometimes you need to ask a question while the script is running and wait for a response from the person running your script. The bash shell provides the `read`command just for this purpose.

Readingbasics

The`read`commandaccepts input either from standard input (such as from the keyboard) or from another file descriptor. After receiving the input, the `read`command places the data into a variable. Here's the `read`command at its simplest:

```
$cattest21.sh
#!/bin/bash
#testingthereadcommand #
echo-n"Enteryourname:"read
name
echo"Hello$name,welcometomyprogram."#
$
$./test21.sh
Enteryourname:RichBlum
HelloRichBlum,welcometomyprogram.
$
```

That's pretty simple. Notice that the `echo`command that produced the prompt uses the `–n` option. This suppresses the newline character at the end of the string, allowing the script user to enter data immediately after the string, instead of on the next line. This gives yourscripts a more form-like appearance.

Infact, the `read`command includes the `–p`option, which allows you to specify a prompt directly in the `read`command line:

```
$cattest22.sh
#!/bin/bash
#testingtheread-poption #
read-p"Pleaseenteryourage:"age days=$[
$age * 365 ]
echo"Thatmakesyouover$daysdaysold!"#
$
$./test22.sh
Pleaseenteryourage:10
Thatmakesyouover3650daysold!
$
```

You'llnotice in the first example that when a name was entered, the readcommand assignedboththefirstnameandlastnametothesamevariable.Thereadcommand assigns all data entered at the prompt to a single variable, or you can specify multiple variables. Each data value entered is assigned to the next variable in the list. If the list of variables runs out before the data does, the remaining data is assigned to the last variable:

```
$cattest23.sh
#!/bin/bash
#enteringmultiplevariables #
read-p"Enteryourname:"firstlast
echo"Checkingdatafor$last,$first…"
$
$./test23.sh
Enteryourname:RichBlum
CheckingdataforBlum,Rich...
$
```

Youcanalsospecify no variables on the readcommand line. If you do that, the readcommand places any data it receives in the special environment variable *REPLY*:

```
$cattest24.sh
#!/bin/bash
#TestingtheREPLYEnvironmentvariable #
read-p"Enteryourname:"echo
echoHello$REPLY,welcometomyprogram. #
$
$./test24.sh
Enteryourname:Christine

HelloChristine,welcometomyprogram.
$
```

The REPLYenvironment variable contains all the data entered in the input, and it can be used in the shell script as any other variable.

Timingout

Be careful when using the readcommand. Your script may get stuck waiting for the script user to enter data. If the script must go on regardless of whether any data was entered, you can use the -toption to specify a timer. The -toption specifies the number of seconds for thereadcommand to wait for input. When the timer expires, the readcommand returns a non-zero exit status:

```
$cattest25.sh
#!/bin/bash
```

```
#timingthedataentry #
ifread-t5-p"Pleaseenteryourname:"name then
   echo"Hello$name,welcometomyscript" else
   echo
   echo"Sorry,tooslow!"
fi
$
$./test25.sh
Pleaseenteryourname:Rich
HelloRich,welcometomyscript
$
$./test25.sh
Pleaseenteryourname:
Sorry,tooslow!
$
```

Because the `read`command exits with a non-zero exit status if the timer expires, it's easytousethe standard structured statements, such as an `if-then`statement or a `while`loop totrackwhathappened.Inthisexample,whenthetimerexpires,the`if`statementfails, and the shell executes the commands in the `else`section.

Insteadof timing the input, you can also set the `read`command to count the input characters. When a preset number of characters has been entered, it automatically exits, assigning the entered data to the variable:

```
$cattest26.sh
#!/bin/bash
#gettingjustonecharacterofinput #
read-n1-p"Doyouwanttocontinue[Y/N]?"answer case
$answer in
Y|y)echo
       echo"fine,continueon…";;
N|n)echo
       echoOK,goodbye
       exit;;
esac
echo"Thisistheendofthescript"
$
$./test26.sh
Doyouwanttocontinue[Y/N]?Y
fine,continueon…
Thisistheendofthescript
$
$./test26.sh
```

```
Doyouwanttocontinue[Y/N]?n
OK,goodbye
$
```

This example uses the -noption with the value of 1, instructing the readcommand to accept only a single character before exiting. As soon as you press the single character to answer,the readcommand accepts the input and passes it to the variable. You don't need to press the Enter key.

Readingwithnodisplay

Sometimes you need input from the script user, but you don't want that input to display on the monitor. The classic example is when entering passwords, but there are plenty of other types of data that you need to hide.

The-soptionprevents the data entered in the readcommand from being displayed on the monitor; actually, the data is displayed, but the readcommand sets the text color to the same as the background color. Here's an example of using the -soption in a script:

```
$cattest27.sh
#!/bin/bash
#hidinginputdatafromthemonitor #
read-s-p"Enteryourpassword:"pass echo
echo"Isyourpasswordreally$pass?"
$
$./test27.sh
Enteryourpassword:
IsyourpasswordreallyT3st1ng?
$
```

The data typed at the input prompt doesn't appear on the monitor but is assigned to the variable for use in the script.

Readingfromafile

Finally, you can also use the readcommand to read data stored in a file on the Linux system. Each call to the readcommand reads a single line of text from the file. When no more linesareleftinthefile,thereadcommandexitswithanon-zeroexitstatus.

The tricky part is getting the data from the file to the readcommand. The most common methodisto pipethe result of the catcommand of the file directly to a whilecommand that contains the readcommand. Here's an example:

```
$cattest28.sh
#!/bin/bash
```

```
#readingdatafromafile #
count=1
cattest|whilereadline do
   echo"Line$count:$line"count=$
   [ $count + 1]
done
echo"Finishedprocessingthefile"
$
$cattest
Thequickbrowndogjumpsoverthelazyfox.
Thisisatest,thisisonlyatest.
ORomeo,Romeo!WhereforeartthouRomeo?
$
$./test28.sh
Line1:Thequickbrowndogjumpsoverthelazyfox.
Line2:Thisisatest,thisisonlyatest.
Line3:ORomeo,Romeo!WhereforeartthouRomeo?
Finishedprocessingthefile
$
```

The`while`commandloop continues processing lines of the file with the `read`command, until the `read`command exits with a non-zero exit status.

Summary

Thischaptershowedthreemethodsforretrievingdatafromthescriptuser.Command line parameters allow users to enter data directly on the command line when they run the script. The script uses positional parameters to retrieve the command line parameters and assign them to variables.

The `shift`command allows you to manipulate the command line parameters by rotating them within the positional parameters. This command allows you to easily iterate through the parameters without knowing how many parameters are available.

You can use three special variables when working with command line parameters. The shell sets the `$#`variable to the number of parameters entered on the command line. The `$*` variable contains all the parameters as a single string, and the `$@`variable contains all the parametersasseparatewords.Thesevariablescomeinhandywhenyou'retryingtoprocess long parameter lists.

Besides parameters, your script users can use command line options to pass information to your script. Command line options are single letters preceded by a dash. Different options can be assigned to alter the behavior of your script.

Thebashshellprovidesthreewaystohandlecommandlineoptions.

The first way is to handle them just like command line parameters. You can iterate through the options using the positional parameter variables, processing each option as it appearson the command line.

Anotherway to handle command line options is with the `getopt`command. This command converts command line options and parameters into a standard format that you can process in your script. The `getopt`command allows you to specify which letters it recognizes as options and which options require an additional parameter value. The `getopt`command processes the standard command line parameters and outputs the options and parametersin the proper order.

The final method for handling command line options is via the `getopts`command (note that it's plural). The `getopts`command provides more advanced processing of the command line parameters. It allows for multi-value parameters, along with identifying options not defined by the script.

An interactive method to obtain data from your script users is the `read`command. The `read`commandallowsyourscriptstoqueryusersforinformationandwait.The`read`com- mand places any data entered by the script user into one or more variables, which you can use within the script.

Severaloptions are available for the `read`command that allow you to customize the data input into your script, such as using hidden data entry, applying timed data entry, andrequesting a specific number of input characters.

In the next chapter, we look further into how bash shell scripts output data. So far, you'veseen how to display data on the monitor and redirect it to a file. Next, we explore a fewotheroptionsthatyouhaveavailablenotonlytodirectdatatospecificlocationsbutalso to direct specific types of data to specific locations. This will help make your shell scripts look professional!

ScriptControl

INTHISCHAPTER

Handlingsignals

Runningscriptsinthebackground

Forbiddinghang-ups

Controlling a Job

Modifying script priority

Automatingscriptexecution

Asyoustartbuildingadvancedscripts,you'llprobablywonderhowtorunandcontrolthem on your Linux system. So far in this book, the only way we've run scripts is directly from the command line interface in real-time mode. This isn't the only way to run scripts in Linux.

Quite a few options are available for running your shell scripts. There are also options for controlling your scripts. Various control methods include sending signals to your script, modifying a script's priority, and switching the run mode while a script is running. This chapter examines the different ways you can control your shell scripts.

HandlingSignals

Linux uses signals to communicate with processes running on the system. Chapter 4 described the different Linux signals and how the Linux system uses these signals to stop, start, and kill pro-cesses. You can control the operation of your shell script by programming the script to perform cer- tain commands when it receives specific signals.

Signalingthebashshell

There are more than 30 Linux signals that can be generated by the system and applications. Table 16-1 lists the most common Linux system signals that you'll run across in your shell script writing.

TABLE16-1 **LinuxSignals**

Signal	Value	Description
1	SIGHUP	Hangsuptheprocess
2	SIGINT	Interrupts theprocess
3	SIGQUIT	Stopstheprocess
9	SIGKILL	Unconditionallyterminatestheprocess
15	SIGTERM	Terminatestheprocessifpossible
17	SIGSTOP	Unconditionallystops,butdoesn’tterminate,theprocess
18	SIGTSTP	Stopsor pausestheprocess,butdoesn’tterminate
19	SIGCONT	Continuesastoppedprocess

Bydefault,the bash shell ignores any `SIGQUIT(3)` and `SIGTERM(15)` signals it receives (soaninteractiveshellcannotbeaccidentallyterminated).However,thebashshelldoes notignoreany`SIGHUP (1)` and`SIGINT (2)` signalsitreceives.

If the bash shell receives a `SIGHUP`signal, such as when you leave an interactive shell, it exits. Before it exits, however, it passes the `SIGHUP`signal to any processes started by the shell, including any running shell scripts.

Witha `SIGINT`signal, the shell is just interrupted. The Linux kernel stops giving the shell processingtimeon the CPU. When this happens, the shell passes the `SIGINT`signal to any processes started by the shell to notify them of the situation.

As you probably have noticed, the shell passes these signals on to your shell script program for processing. However, a shell script’s default behavior does not govern these signals,which may have an adverse effect on the script’s operation. To avoid this situation, you can program your script to recognize signals and perform commands to prepare the script forthe consequences of the signal.

Generatingsignals

The bash shell allows you to generate two basic Linux signals using key combinations on thekeyboard.Thisfeaturecomesinhandyifyouneedtostoporpausearunawayscript.

Interruptingaprocess

The Ctrl+C key combination generates a `SIGINT`signal and sends it to any processes currently running in the shell. You can test this by running a command that normally takes along time to finish and pressing the Ctrl+C key combination:

```
$sleep100
^C
$
```

TheCtrl+Ckey combination sends a `SIGINT`signal, which simply stops the current process running in the shell. The `sleep`command pauses the shell's operation for the specified number of seconds and returns the shell prompt. By pressing the Ctrl+C key combination before the time passed, the `sleep`command terminated prematurely.

Pausingaprocess

Instead of terminating a process, you can pause it in the middle of whatever it's doing. Sometimes,thiscanbeadangerousthing(forexample,ifascripthasafilelockopenon a crucial system file), but often it allows you to peek inside what a script is doing without actually terminating the process.

TheCtrl+Z key combination generates a `SIGTSTP`signal, stopping any processes running in theshell.Stoppingaprocessisdifferentthanterminatingtheprocess.Stoppingtheprocess leaves the program in memory and able to continue running from where it left off. In the "Controlling the Job" section later in this chapter, you learn how to restart a process that's been stopped.

When you use the Ctrl+Z key combination, the shell informs you that the process has been stopped:

```
$sleep100
^Z
[1]+Stopped                    sleep100
$
```

Thenumber in the square brackets is the *jobnumber*assigned by the shell. The shell refers to each process running in the shell as a *job* and assigns each job a unique job number within the current shell. It assigns the first started process job number 1, the second job number 2, and so on.

If you have a stopped job assigned to your shell session, bash warns you if you try to exit the shell:

```
$sleep100
^Z
[1]+Stopped                    sleep100
$exit
exit
Therearestoppedjobs.
$
```

Youcanviewthestoppedjobsusingthe`ps`command:

```
$sleep100
^Z
[1]+Stopped                    sleep100
$
$ps-l
```

```
F S UID   PID  PPID  C PRI NI  ADDRSZ  WCHAN  TTY       TIME CMD
0 S 501  2431  2430  0  80  0 -27118  wait   pts/0 00:00:00 bash
0 T 501  2456  2431  0  80  0 -25227  signal pts/0 00:00:00 sleep
0 R 501  2458  2431  0  80  0 -27034  -      pts/0 00:00:00 ps
$
```

IntheScolumn(process state), the pscommand shows the stopped job's state as T. This indicates the command is either being traced or is stopped.

If you really want to exit the shell with a stopped job still active, just type the exitcommand again. The shell exits, terminating the stopped job. Alternately, now that you know the PID of the stopped job, you can use the killcommand to send a SIGKILLsignal to terminate it:

```
$kill-92456
$
[1]+Killed                    sleep100
$
```

When you kill the job, initially you don't get any response. However, the next time you do something that produces a shell prompt (such as pressing the Enter key), you'll see a message indicating that the job was killed. Each time the shell produces a prompt, it also displays the status of any jobs that have changed states in the shell. After you kill a job, the next timeyouforcetheshelltoproduceaprompt,itdisplaysamessageshowingthatthejobwaskilled while running.

Trappingsignals

Instead of allowing your script to leave signals ungoverned, you can trap them when they appear and perform other commands. The trapcommand allows you to specify which Linux signals your shell script can watch for and intercept from the shell. If the script receives a signal listed in the trapcommand, it prevents it from being processed by the shell and instead handles it locally.

Theformatofthetrapcommandis:

trap*commands signals*

On the trapcommand line, you just list the commands you want the shell to execute, along with a space-separated list of signals you want to trap. You can specify the signals either by their numeric value or by their Linux signal name.

Here'sasimpleexample of using the trapcommand to capture the SIGINTsignal and gov- ern the script's behavior when the signal is sent:

```
$cattest1.sh
#!/bin/bash
#Testingsignaltrapping #
```

```
trap"echo'Sorry!IhavetrappedCtrl-C'"SIGINT #
echoThisisatestscript #
count=1
while[$count-le10] do
   echo"Loop#$count"s
   leep 1
   count=$[$count+1] done
#
echo"Thisistheendofthetestscript"#
```

The `trap`command used in this example displays a simple text message each time it detects the `SIGINT`signal. Trapping this signal makes this script impervious to the user attempting to stop the program by using the bash shell keyboard Ctrl+C command:

```
$./test1.sh
Thisisatestscript
Loop #1
Loop#2
Loop#3
Loop#4
Loop#5
^CSorry!IhavetrappedCtrl-C
Loop #6
Loop#7
Loop#8
^CSorry!IhavetrappedCtrl-C
Loop #9
Loop#10
Thisistheendofthetestscript
$
```

Each time the Ctrl+C key combination was used, the script executed the `echo`statement specified in the `trap`command instead of not managing the signal and allowing the shellto stop the script.

Trappingascriptexit

Besides trapping signals in your shell script, you can trap them when the shell script exits. This is a convenient way to perform commands just as the shell finishes its job.

Totraptheshellscriptexiting,justaddthe`EXIT`signaltothe`trap`command:

```
$cattest2.sh
#!/bin/bash
```

```
#Trappingthescriptexit #
trap"echoGoodbye..."EXIT #
count=1
while[$count-le5] do
   echo"Loop#$count"s
   leep 1
   count=$[$count+1] done
#
$
$./test2.sh
Loop#1
Loop#2
Loop#3
Loop#4
Loop #5
Goodbye...
$
```

When the script gets to the normal exit point, the trap is triggered, and the shell executesthecommandyouspecifyonthe `trap`command line. The `EXIT`trap also works if you pre- maturely exit the script:

```
$./test2.sh
Loop#1
Loop#2
Loop#3
^CGoodbye...

$
```

Because the `SIGINT`signal isn't listed in the `trap`command list, when the Ctrl+C key combination is used to send that signal, the script exits. However, before the script exits,because the `EXIT`is trapped, the shell executes the `trap`command.

Modifyingorremovingatrap

Tohandletrapsdifferentlyinvarioussectionsofyourshellscript,yousimplyreissuethe `trap`commandwithnewoptions:

```
$cattest3.sh
#!/bin/bash
#Modifyingasettrap #
trap"echo'Sorry...Ctrl-Cistrapped.'"SIGINT
```

```
#
count=1
while[$count-le5] do
   echo"Loop#$count"s
   leep 1
   count=$[$count+1] done
#
trap"echo'Imodifiedthetrap!'"SIGINT #
count=1
while[$count-le5] do
   echo"SecondLoop#$count"sleep
   1
   count=$[$count+1] done
#
$
```

After the signal trap is modified, the script manages the signal or signals differently.However, if a signal is received before the trap is modified, the script processes it per the original`trap`command:

```
$./test3.sh
Loop#1
Loop#2
Loop#3
^CSorry...Ctrl-Cistrapped. Loop
#4
Loop#5
SecondLoop#1
SecondLoop#2
^CImodifiedthetrap!
Second Loop #3
SecondLoop#4
SecondLoop#5
$
```

Youcan also remove a set trap. Simply add two dashes after the `trap`command and a list of the signals you want to return to default behavior:

```
$cattest3b.sh
#!/bin/bash
#Removingasettrap #
trap"echo'Sorry...Ctrl-Cistrapped.'"SIGINT #
```

```
count=1
while[$count-le5] do
   echo"Loop#$count"s
   leep 1
   count=$[$count+1] done
#
#Removethetrap
trap -- SIGINT
echo"Ijustremovedthetrap"#
count=1
while[$count-le5] do
   echo"SecondLoop#$count"sleep
   1
   count=$[$count+1] done
#
$./test3b.sh
Loop#1
Loop#2
Loop#3
Loop#4
Loop#5
Ijustremovedthetrap
Second Loop #1
SecondLoop#2
SecondLoop#3
^C
$
```

TIP

Youcanuseasingledashinsteadofadoubledashafterthe`trap`commandtoreturnsignalstotheirdefaultbehav- ior. Both the single and double dash work properly.

After the signal trap is removed, the script handles the `SIGINT`signal in its default manner, terminating the script. However, if a signal is received before the trap is removed, the script processes it per the original `trap`command:

```
$./test3b.sh
Loop#1
Loop#2
Loop#3
^CSorry...Ctrl-Cistrapped. Loop
#4
```

```
Loop#5
Ijustremovedthetrap
Second Loop #1
SecondLoop#2
^C
$
```

In this example, the first Ctrl+C key combination was used to attempt to terminate thescript prematurely. Because the signal was received before the trap was removed, the script executed the command specified in the trap. After the script executed the trap removal,then Ctrl+C could prematurely terminate the script.

RunningScriptsinBackgroundMode

Sometimes, running a shell script directly from the command line interface is inconvenient. Some scripts can take a long time to process, and you may not want to tie up thecommand line interface waiting. While the script is running, you can't do anything else inyour terminal session. Fortunately, there's a simple solution to that problem.

When you use the `ps`command, you see a whole bunch of different processes running ontheLinuxsystem.Obviously,alltheseprocessesarenotrunningonyourterminalmonitor.Thisiscalledrunningprocessesinthe***background*.**Inbackgroundmode,aprocessruns without being associated with a `STDIN`, `STDOUT`, and `STDERR`on a terminal session (see Chapter 15).

You can exploit this feature with your shell scripts as well, allowing them to run behind the scenes and not lock up your terminal session. The following sections describe how to run your scripts in background mode on your Linux system.

Runninginthebackground

Running a shell script in background mode is a fairly easy thing to do. To run a shell scriptin background mode from the command line interface, just place an ampersand symbol (&) after the command:

```
$cattest4.sh
#!/bin/bash
#Testrunninginthebackground #
count=1
while[$count-le10] do
   sleep1
   count=$[$count+1] done
```

```
#
$
$./test4.sh&
[1]3231
$
```

When you place the ampersand symbol after a command, it separates the command from the bash shell and runs it as a separate background process on the system. The first thing that displays is the line:

```
[1]3231
```

The number in the square brackets is the job number assigned by the shell to the background process. The next number is the Process ID (PID) the Linux system assigns to the process. Every process running on the Linux system must have a unique PID.

As soon as the system displays these items, a new command line interface prompt appears. You are returned to the shell, and the command you executed runs safely in background mode. At this point, you can enter new commands at the prompt.

Whenthebackgroundprocessfinishes,itdisplaysamessageontheterminal:

```
[1]   Done                    ./test4.sh
```

This shows the job number and the status of the job (`Done`), along with the command used to start the job.

Be aware that while the background process is running, it still uses your terminal monitor for`STDOUT`and`STDERR`messages:

```
$cattest5.sh
#!/bin/bash
#Testrunninginthebackgroundwithoutput #
echo"Startthetestscript"count=1
while[$count-le5] do
   echo"Loop#$count"s
   leep 5
   count=$[$count+1] done
#
echo"Testscriptiscomplete"#
$
$./test5.sh&
[1]3275
```

```
$Startthetestscript
Loop #1
Loop#2
Loop#3
Loop#4
Loop#5
Testscriptiscomplete

[1]   Done                    ./test5.sh
$
```

You'll notice from the example that the output from the `test5.sh`script displays. The outputintermixeswith the shell prompt, which is why `Startthetestscript`appears next to the `$`prompt.

Youcanstillissuecommandswhilethisoutputisoccurring:

```
$./test5.sh&
[1]3319
$Startthetestscript
Loop #1
Loop#2
Loop#3
lsmyprog*
myprogmyprog.c
$Loop#4
Loop#5
Testscriptiscomplete

[1]+Done                      ./test5.sh
$$
```

While the `test5.sh`script is running in the background, the command `lsmyprog*` was entered. The script's output, the typed command, and the command's output all intermixed with each other's output display. This can be confusing! It is a good idea to redirect `STDOUT`and `STDERR`for scripts you will be running in the background (Chapter 15) to avoid this messy output.

Runningmultiplebackgroundjobs

You can start any number of background jobs at the same time from the command line prompt:

```
$./test6.sh&
[1]3568
$ThisisTestScript#1

$./test7.sh&
```

```
[2]3570
$ThisisTestScript#2

$./test8.sh&
[3]3573
$And...anotherTestscript

$./test9.sh&
[4]3576
$Then...therewasonemoretestscript

$
```

Each time you start a new job, the Linux system assigns it a new job number and PID. You can see that all the scripts are running using the `ps`command:

```
$ps
  PIDTTY           TIMECMD
 2431pts/0     00:00:00bash
 3568pts/0     00:00:00test6.sh
 3570pts/0     00:00:00test7.sh
 3573pts/0     00:00:00test8.sh
 3574pts/0     00:00:00sleep
 3575pts/0     00:00:00sleep
 3576pts/0     00:00:00test9.sh
 3577pts/0     00:00:00sleep
 3578pts/0     00:00:00sleep
 3579pts/0     00:00:00ps
$
```

Youmustbecarefulwhenusingbackgroundprocessesfromaterminalsession.Noticeinthe output from the `ps`command that each of the background processes is tied to the terminal session (pts/0) terminal. If the terminal session exits, the background process also exits.

NOTE

Earlierinthischapterwementionedthatwhenyouattempttoexitaterminalsession,awarningisissuedifthere arestoppedprocesses.However,withbackgroundprocesses,onlysometerminalemulatorsremindyouthataback-ground job is running, before you attempt to exit the terminal session.

If you want your script to continue running in background mode after you have logged offthe console, there's something else you need to do. The next section discusses that process.

RunningScriptswithoutaHang-Up

Sometimes, you may want to start a shell script from a terminal session and let the script run in background mode until it finishes, even if you exit the terminal session. You can dothis by using the `nohup`command.

The`nohup`commandrunsanother command blocking any `SIGHUP`signals that are sent to the process. This prevents the process from exiting when you exit your terminal session.

Theformatusedfor the `nohup`command isas follows:

```
$nohup./test1.sh&
[1]3856
$nohup:ignoringinputandappendingoutputto'nohup.out'

$
```

As with a normal background process, the shell assigns the command a job number, and the Linux system assigns a PID number. The difference is that when you use the `nohup`command, the script ignores any `SIGHUP`signals sent by the terminal session if you close the session.

Becausethe`nohup`command disassociates the process from the terminal, the process loses the `STDOUT`and `STDERR`output links. To accommodate any output generated by the command,the`nohup`commandautomaticallyredirects`STDOUT`and`STDERR`messagestoafile, called `nohup.out`.

NOTE

Ifyourunanothercommandusing`nohup`,theoutputisappendedtotheexisting `nohup.out`file.Becarefulwhen running multiple commands from the same directory, because all the output is sent to the same `nohup.out`file, which can get confusing.

The `nohup.out`file contains all the output that would normally be sent to the terminal monitor. After the process finishes running, you can view the `nohup.out`file for the output results:

```
$catnohup.out
This isatestscript
Loop 1
Loop 2
Loop 3
Loop 4
Loop 5
Loop 6
Loop 7
Loop 8
Loop 9
Loop 10
This istheendofthetestscript
$
```

Theoutputappearsinthe`nohup.out`filejustasiftheprocessranonthecommandline.

ControllingtheJob

Earlier in this chapter, you saw how to use the Ctrl+C key combination to stop a job running in the shell. Afteryoustopajob,theLinuxsystemletsyoueitherkillorrestartit. Youcan kill the process by using the `kill`command. Restarting a stopped process requires that you send it a `SIGCONT`signal.

The function of starting, stopping, killing, and resuming jobs is called *jobcontrol*. With job control, you have full control over how processes run in your shell environment. This section describes the commands used to view and control jobs running in your shell.

Viewingjobs

Thekeycommandforjobcontrolisthe `jobs`command.The`jobs`commandallowsyouto view the current jobs being handled by the shell:

```
$cattest10.sh
#!/bin/bash
#Testjobcontrol #
echo"ScriptProcessID:$$"#
count=1
while[$count-le10] do
   echo"Loop#$count"s
   leep 10
   count=$[$count+1] done
#
echo"Endofscript..."#
$
```

The script uses the `$$`variable to display the PID that the Linux system assigns to the script; then it goes into a loop, sleeping for 10 seconds at a time for each iteration.

You can start the script from the command line interface and then stop it using the Ctrl+Z key combination:

```
$./test10.sh
ScriptProcessID:1897
Loop #1
Loop#2
^Z
[1]+Stopped                  ./test10.sh
$
```

Using the same script, another job is started as a background process, using the ampersand symbol. To make life a little easier, the output of that script is redirected to a file so itdoesn't appear on the screen:

```
$./test10.sh>test10.out&
[2]1917
$
```

The`jobs`commandenablesyouto view the jobs assigned to the shell. The `jobs`command shows both the stopped and the running jobs, along with their job numbers and the commands used in the jobs:

```
$jobs
[1]+Stopped                   ./test10.sh
[2]-Running                   ./test10.sh>test10.out&
$
```

Youcanviewthevarious jobs' PIDs byadding the `-l`parameter(lowercase L) on the`jobs` command:

```
$jobs-l
[1]+1897Stopped                   ./test10.sh
[2]-1917Running                   ./test10.sh>test10.out&
$
```

The`jobs`commandusesafewdifferentcommandlineparameters,asshowninTable16-2.

TABLE16-2 **ThejobsCommandParameters**

Parameter	Description
`-l`	ListsthePIDoftheprocessalongwiththejobnumber
`-n`	Listsonlyjobsthathavechangedtheirstatussincethelastnotificationfromthe shell
`-p`	ListsonlythePIDsofthejobs
`-r`	Listsonlytherunningjobs
`-s`	Listsonlystoppedjobs

You probably noticed the plus and minus signs in the `jobs`command output. The job with the plus sign is considered the default job. It would be the job referenced by any job control commands if a job number wasn't specified in the command line.

The job with the minus sign is the job that would become the default job when the current default job finishes processing. There will be only one job with the plus sign and one job withtheminussignatanytime,nomatterhowmanyjobsarerunningintheshell.

The following is an example showing how the next job in line takes over the default status, when the default job is removed. Three separate processes are started in the background. The jobscommand listing shows the three processes, their PID, and their status. Note that the default process (the one listed with the plus sign) is the last process started, job #3.

```
$./test10.sh>test10a.out&
[1]1950
$./test10.sh>test10b.out&
[2]1952
$./test10.sh>test10c.out&
[3]1955
$
$jobs-l
[1]   1950Running                     ./test10.sh>test10a.out&
[2]-1952Running                       ./test10.sh>test10b.out&
[3]+1955Running                       ./test10.sh>test10c.out&
$
```

Usingthekillcommand to send a SIGHUPsignal to the default process causes the job to terminate.Inthenextjobslisting,thejobthatpreviouslyhadtheminussignnowhas the plus sign and is the default job:

```
$kill1955
$
[3]+Terminated                ./test10.sh>test10c.out
$
$jobs-l
[1]-1950Running                       ./test10.sh>test10a.out&
[2]+1952Running                       ./test10.sh>test10b.out&
$
$kill1952
$
[2]+Terminated                ./test10.sh>test10b.out
$
$jobs-l
[1]+1950Running                       ./test10.sh>test10a.out&
$
```

Although changing a background job to the default process is interesting, it doesn't seem very useful. In the next section, you learn how to use commands to interact with thedefault process using no PID or job number.

Restartingstoppedjobs

Under bash job control, you can restart any stopped job as either a background process or a foregroundprocess.Aforegroundprocesstakesovercontroloftheterminalyou'reworking on, so be careful about using that feature.

Torestartajobinbackgroundmode,usethe`bg`command:

```
$./test11.sh
^Z
[1]+Stopped                  ./test11.sh
$
$bg
[1]+./test11.sh&
$
$jobs
[1] +Running                 ./test11.sh&
$
```

Because the job was the default job, indicated by the plus sign, only the `bg`command was needed to restart it in background mode. Notice that no PID is listed when the job is moved into background mode.

Ifyouhaveadditionaljobs,youneedtousethejobnumberalongwiththe`bg`command:

```
$./test11.sh
^Z
[1] +Stopped                 ./test11.sh
$
$./test12.sh
^Z
[2] +Stopped                 ./test12.sh
$
$bg2
[2]+./test12.sh&
$
$jobs
[1]+Stopped                  ./test11.sh
[2]-Running                  ./test12.sh&
$
```

The command `bg2`was used to send the second job into background mode. Notice that when the `jobs`command was used, it listed both jobs with their status, even though the default job is not currently in background mode.

Torestartajobinforegroundmode,usethe`fg`command,alongwiththejobnumber:

```
$fg2
./test12.sh
Thisisthescript'send...
$
```

Because the job is running in foreground mode, the command line interface prompt does not appear until the job finishes.

BeingNice

In a multitasking operating system (which Linux is), the kernel is responsible for assigning CPU time for each process running on the system. The *schedulingpriority* is the amount of CPU time the kernel assigns to the process relative to the other processes. By default, all processes started from the shell have the same scheduling priority on the Linux system.

The scheduling priority is an integer value, from -20 (the highest priority) to +19 (the low-estpriority).Bydefault,thebashshellstartsallprocesseswithaschedulingpriorityof0.

TIP

It'sconfusingtorememberthat-20,thelowestvalue,isthehighestpriorityand19,thehighestvalue,isthelow- est priority. Just remember the phrase, "Nice guys finish last." The "nicer" or higher you are in value, the lower your chance of getting the CPU.

Sometimes, you want to change the priority of a shell script, either lowering its priority soit doesn't take as much processing power away from other processes or giving it a higher priority so it gets more processing time. You can do this by using the `nice`command.

Usingthenicecommand

The`nice`command allows you to set the scheduling priority of a command as you start it. To make a command run with less priority, just use the `-n`command line option for `nice` to specify a new priority level:

```
$nice-n10./test4.sh>test4.out&
[1]4973
$
$ps-p4973-opid,ppid,ni,cmd
  PIDPPIDNICMD
 4973472110/bin/bash./test4.sh
$
```

Notice that you must use the `nice`command on the same line as the command you are starting. The output from the `ps`command confirms that the nice value (column `NI`) has been set to `10`.

The`nice`command causes the script to run at a lower priority. However, if you try to increase the priority of one of your commands, you might be in for a surprise:

```
$nice-n-10./test4.sh>test4.out&
[1]4985
$nice:cannotsetniceness:Permissiondenied

 [1]+Done                          nice-n-10./test4.sh>test4.out
$
```

The nicecommand prevents normal system users from increasing the priority of their commands. Notice that the job does run, even though the attempt to raise its priority with thenicecommandfailed.

Youdon't have to use the −noption with the nicecommand. You can simply type the priority preceded by a dash:

```
$nice-10./test4.sh>test4.out&
[1]4993
$
$ps-p4993-opid,ppid,ni,cmd
  PIDPPIDNICMD
 4993472110/bin/bash./test4.sh
$
```

However, this can get confusing when the priority is a negative number, because you must have a double-dash. It's best just to use the −noption to avoid confusion.

Usingtherenicecommand

Sometimes, you'd like to change the priority of a command that's already running on the system. That's what the renicecommand is for. It allows you to specify the PID of a running process to change its priority:

```
$./test11.sh&
[1]5055
$
$ps-p5055-opid,ppid,ni,cmd
  PIDPPIDNICMD
 50554721     0/bin/bash./test11.sh
$
$renice-n10-p5055
5055:oldpriority0,newpriority10
$
$ps-p5055-opid,ppid,ni,cmd
  PIDPPIDNICMD
 5055472110/bin/bash./test11.sh
$
```

Therenicecommand automatically updates the scheduling priority of the running process. As with the nicecommand, the renicecommand has some limitations:

- Youcanonlyreniceprocessesthatyouown.
- Youcanonlyreniceyourprocessestoalowerpriority.
- Therootusercanreniceanyprocesstoanypriority.

If you want to fully control running processes, you must be logged in as the root account or usethesudocommand.

RunningLikeClockwork

When you start working with scripts, you may want to run a script at a preset time, usually at a time when you're not there. The Linux system provides a couple of ways to run a scriptat a preselected time: the `at`command and the `cron`table. Each method uses a different technique for scheduling when and how often to run scripts. The following sections describe each of these methods.

Schedulingajobusingtheatcommand

The `at`command allows you to specify a time when the Linux system will run a script. The `at`commandsubmitsajobtoaqueuewithdirectionsonwhentheshellshouldrunthe job. The `at`daemon, `atd`, runs in the background and checks the job queue for jobs to run. Most Linux distributions start this daemon automatically at boot time.

The `atd`daemon checks a special directory on the system (usually `/var/spool/at`) for jobs submitted using the `at`command. By default, the `atd`daemon checks this directory every 60 seconds. When a job is present, the `atd`daemon checks the time the job is set to be run. If the time matches the current time, the `atd`daemon runs the job.

Thefollowing sections describe how to use the `at`command to submit jobs to run and how to manage these jobs.

Understandingtheatcommandformat

Thebasic`at`commandformatisprettysimple:

```
at[-ffilename]time
```

Bydefault, the `at`command submits input from `STDIN`to the queue. You can specify a filename used to read commands (your script file) using the `−f`parameter.

The *time*parameter specifies when you want the Linux system to run the job. If you specify a time that has already passed, the `at`command runs the job at that time on the nextday.

You can get pretty creative with how you specify the time. The `at`command recognizes lots of different time formats:

- Astandardhourandminute,suchas10:15
- AnAM/PMindicator,suchas10:15PM
- Aspecificnamedtime,suchasnow,noon,midnight,orteatime(4PM)

Inadditiontospecifyingthetimetorunthejob,youcanalsoincludeaspecificdate,using a few different date formats:

- Astandarddateformat,suchasMMDDYY,MM/DD/YY,orDD.MM.YY
- Atextdate,suchasJul4orDec25,withorwithouttheyear
- Atimeincrement:
  - Now+25minutes
  - 10:15PMtomorrow
  - 10:15+7days

Whenyouuse the `at`command, the job is submitted into a ***jobqueue***. The job queue holds the jobs submitted by the `at`command for processing. There are 26 different job queues available for different priority levels. Job queues are referenced using lowercase letters, ***a*** through ***z***, and uppercase letters ***A*** through ***Z***.

NOTE
Afewyearsago,the`batch`commandwasanothermethodthatallowedascripttoberunatalatertime.The `batch`command was unique because you could schedule a script to run when the system was at a lower usage level. However, nowadays, the `batch`command is just simply a script, `/usr/bin/batch`, that calls the `at`command and submits your job to the `b` queue.

Thehigheralphabetically the job queue, the lower the priority (higher `nice`value) the job willrununder.Bydefault,`at`jobsaresubmittedtothe`at`job`a`queue.Ifyouwanttorun a job at a lower priority, you can specify a different queue letter using the `−q`parameter.

Retrievingjoboutput

When the job runs on the Linux system, there's no monitor associated with the job.Instead, the Linux system uses the e-mail address of the user who submitted the job as `STDOUT`and`STDERR`.Anyoutputdestined to `STDOUT`or `STDERR`is mailed to the user via the mail system.

Here'sasimple example using the `at`command to schedule a job to run on a CentOS distribution:

```
$cattest13.sh
#!/bin/bash
#Testusingatcommand #
echo"Thisscriptranat$(date+%B%d,%T)" echo
sleep5
echo"Thisisthescript'send..."#
```

```
$at-ftest13.shnow
job7at2015-07-1412:38
$
```

The at command displays the job number assigned to the job along with the time the job is scheduled to run. The -f option tells what script file to use and the now time designation directs at to run the script immediately.

Using e-mail for the at command's output is inconvenient at best. The at command sendse-mailviathesendmailapplication.Ifyoursystemdoesnotusesendmail,youwon'tget anyoutput!Therefore,it'sbest to redirect STDOUT and STDERR in your scripts (see Chapter 15) when using the at command, as the following example shows:

```
$cattest13b.sh
#!/bin/bash
#Testusingatcommand #
echo"Thisscriptranat$(date+%B%d,%T)">test13b.out echo >>
test13b.out
sleep5
echo"Thisisthescript'send...">>test13b.out #
$
$at-M-ftest13b.shnow
job8at2015-07-1412:48
$
$cattest13b.out
ThisscriptranatJuly14,12:48:18

Thisisthescript'send...
$
```

If you don't want to use e-mail or redirection with at, it is best to add the -M option to suppress any output generated by jobs using the at command.

Listingpendingjobs

The atq commandallowsyouto view what jobs arepending on the system:

```
$at-M-ftest13b.shteatime
job17at2015-07-1416:00
$
$at-M-ftest13b.shtomorrow
job18at2015-07-1513:03
$
$at-M-ftest13b.sh13:30
job19at2015-07-1413:30
$
$at-M-ftest13b.shnow
```

```
job20at2015-07-1413:03
$
$atq
20      2015-07-1413:03=Christine
18      2015-07-1513:03aChristine
17      2015-07-1416:00aChristine
19      2015-07-1413:30aChristine
$
```

The job listing shows the job number, the date and time the system will run the job, and the job queue the job is stored in.

Removingjobs

After you know the information about what jobs are pending in the job queues, you can use the `atrm`command to remove a pending job:

```
$atq
18       2015-07-15 13:03 a Christine
17       2015-07-14 16:00 a Christine
19       2015-07-14 13:30 a Christine
$
$atrm  18
$
$atq
17       2015-07-14 16:00 a Christine
19       2015-07-14 13:30 a Christine
$
```

Just specify the job number you want to remove. You can only remove jobs that you submit for execution. You can't remove jobs submitted by others.

Schedulingregularscripts

Using the `at`command to schedule a script to run at a preset time is great, but what if you need that script to run at the same time every day or once a week or once a month? Insteadofhavingtocontinuallysubmit`at`jobs,youcanuseanotherfeatureoftheLinuxsystem.

TheLinuxsystem uses the `cron`program to allow you to schedule jobs that need to run onaregularbasis. The `cron`program runs in the background and checks special tables, called *cron tables*, for jobs that are scheduled to run.

Lookingatthecrontable

The`cron`tableuses a special format for allowing you to specify when a job should be run. The format for the `cron`table is:

```
minhourdayofmonthmonthdayofweekcommand
```

The cron table allows you to specify entries as specific values, ranges of values (such as 1–5), or as a wildcard character (the asterisk). For example, if you want to run a command at 10:15 on every day, you would use this cron table entry:

```
1510***command
```

Thewildcardcharacterusedin the *dayofmonth*, *month*, and *dayofweek* fields indicates thatcron willexecute the command every day of every month at 10:15. To specify a command to run at 4:15 PM every Monday, you would use the following:

```
1516**1command
```

You can specify the *dayofweek* entry as either a three-character text value (mon, tue, wed,thu,fri,sat,sun)orasanumericvalue,with0beingSundayand6beingSaturday.

Here's another example: to execute a command at 12 noon on the first day of every month, you would use the following format:

```
00121**command
```

The *dayofmonth* entryspecifiesadatevalue(1–31)forthemonth.

NOTE

Theastutereadermightbewonderingjusthowyouwouldbeabletosetacommandtoexecuteonthelastdayof every month because you can't set the *dayofmonth* value to cover every month. This problem has plagued Linux and Unix programmers, and has spawned quite a few different solutions. A common method is to add an if-then statement that uses the date command to check if tomorrow's date is 01:

```
0012***if[`date+%d-dtomorrow`=01];then;command
```

Thischeckseverydayat12noontoseeifit'sthelastdayofthemonth,andifso, cron runsthecommand.

The command list must specify the full command pathname or shell script to run. You can add any command line parameters or redirection symbols you like, as a regular command line:

```
1510***/home/rich/test4.sh>test4out
```

The cron program runs the script using the user account that submitted the job. Thus, you must have the proper permissions to access the command and output files specified in the command listing.

Buildingthecrontable

Each system user can have their own cron table (including the root user) for running scheduledjobs.Linux provides the crontab command for handling the cron table. To list anexisting cron table,usethe −l parameter:

```
$crontab-l
nocrontabforrich
$
```

Bydefault,eachuser'scrontablefiledoesn'texist.Toaddentriestoyour crontable, usethe –eparameter. When you do that, the crontabcommand starts a text editor (see Chapter 10) with the existing crontable (or an empty file if it doesn't yet exist).

Viewingcrondirectories

Whenyoucreateascriptthathaslesspreciseexecutiontimeneeds,itiseasiertouseone ofthe pre-configured cronscript directories. There are four basic directories: hourly, daily, monthly, and weekly.

```
$ls/etc/cron.*ly
/etc/cron.daily:
cups      makewhatis.cronprelink        tmpwatch
logrotatemlocate.cron     readahead.cron

/etc/cron.hourly:
0anacron

/etc/cron.monthly:
readahead-monthly.cron

/etc/cron.weekly:
$
```

Thus, if you have a script that needs to be run one time per day, just copy the script to thedaily directory and cronexecutes it each day.

Lookingattheanacronprogram

Theonlyproblem with the cronprogram is that it assumes that your Linux system is operational 24 hours a day, 7 days a week. Unless you're running Linux in a server environment, this may not necessarily be true.

IftheLinuxsystemisturnedoffatthetimeajobisscheduledtoruninthecrontable, the job doesn't run. The cronprogram doesn't retroactively run missed jobs when the system is turned back on. To resolve this issue, many Linux distributions also include the anacronprogram.

If anacrondetermines that a job has missed a scheduled running, it runs the job as soonaspossible.ThismeansthatifyourLinuxsystemisturnedoffforafewdays,whenit starts back up, any jobs scheduled to run during the time it was off are automatically run.

This feature is often used for scripts that perform routine log maintenance. If the systemisalwaysoffwhenthescriptshouldrun,thelogfileswouldnevergettrimmedandcould grow to undesirable sizes. With anacron, you're guaranteed that the log files are trimmed at least each time the system is started.

Theanacronprogramdealsonlywithprogramslocatedinthecrondirectories,suchas /etc/cron.monthly. It uses timestamps to determine if the jobs have been run at the proper scheduled interval. A timestamp file exists for each crondirectory and is located in /var/spool/anacron:

```
$sudocat/var/spool/anacron/cron.monthly
20150626
$
```

Theanacronprogramhasitsowntable(usuallylocatedat/etc/anacrontab)tocheck the job directories:

```
$sudocat/etc/anacrontab
# /etc/anacrontab: configuration file for anacron

#Seeanacron(8)andanacrontab(5)fordetails.

SHELL=/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
#themaximalrandomdelayaddedtothebasedelayofthejobs RANDOM_DELAY=45
#thejobswillbestartedduringthefollowinghoursonly
START_HOURS_RANGE=3-22

#periodindays      delayinminutes     job-identifier   command
1       5       cron.daily                nicerun-parts/etc/cron.daily
7       25      cron.weekly               nicerun-parts/etc/cron.weekly
@monthly45      cron.monthly           nicerun-parts/etc/cron.monthly
$
```

Thebasicformatoftheanacrontableisslightlydifferentfromthatofthecrontable:

perioddelayidentifiercommand

The period entry defines how often the jobs should be run, specified in days. The anacron program uses this entry to check against the jobs' timestamp file. The delay entry specifies howmany minutes after the system starts the anacronprogram should run missed scripts. The command entry contains the run-partsprogram and a cronscript directory name. Therun-partsprogramisresponsiblefor runningany scriptin thedirectory passedto it.

Notice that anacrondoes not run the scripts located in /etc/cron.hourly. This is becausethe anacronprogram does not deal with scripts that have execution time needs of less than daily.

The identifier entry is a unique non-blank character string — for example, `cron-weekly`.It is used to uniquely identify the job in log messages and error e-mails.

Startingscriptswithanewshell

The ability to run a script every time a user starts a new bash shell (even just when a specific user starts a bash shell) can come in handy. Sometimes, you want to set shell features for a shell session or just ensure that a specific file has been set.

Recall the startup files run when a user logs into the bash shell (covered in detail in Chapter6). Also, remember that not every distribution has all the startup files. Essentially, the first file found in the following ordered list is run and the rest are ignored:

- `$HOME/.bash_profile`
- `$HOME/.bash_login`
- `$HOME/.profile`

Therefore,youshouldplaceanyscriptsyouwantrunatlogintimeinthefirstfilelisted.

The bash shell runs the `.bashrc`file any time a new shell is started. You can test this by adding a simple echo statement to the `.bashrc`file in your home directory and starting a new shell:

```
$cat.bashrc
#.bashrc

#Sourceglobaldefinitions
if[-f/etc/bashrc];then
        ./etc/bashrc
fi

#Userspecificaliasesandfunctions echo
"I'm in a new shell!"
$
$bash
I'minanewshell!
$
$exit
exit
$
```

The `.bashrc`fileisalsotypicallyrunfromoneofthebashstartupfiles.Becausethe `.bashrc`file runs both when you log into the bash shell and when you start a bash shell, if youneedascripttoruninbothinstances,placeyourshellscriptinsidethisfile.

Summary

The Linux system allows you to control your shell scripts by using signals. The bash shell acceptssignalsandpassesthemontoanyprocessrunningundertheshellprocess.Linuxsig- nals allow you to easily kill a runaway process or temporarily pause a long-running process.

Youcanusethe `trap`statement in your scripts to catch signals and perform commands. This featureprovidesasimplewaytocontrolwhetherausercaninterruptyourscriptwhileit's running.

By default, when you run a script in a terminal session shell, the interactive shell is suspended until the script completes. You can cause a script or command to run in background mode by adding an ampersand sign (&) after the command name. When you run a script or command in background mode, the interactive shell returns, allowing you to continue entering more commands. Any background processes run using this method are still tied to the terminal session. If you exit the terminal session, the background processes also exit.

To prevent this from happening, use the `nohup`command. This command intercepts any signals intended for the command that would stop it — for example, when you exit the terminal session. This allows scripts to continue running in background mode even if you exit the terminal session.

When you move a process to background mode, you can still control what happens to it. The jobs command allows you to view processes started from the shell session. After you know thejobIDofabackgroundprocess,youcanusethe`kill`commandtosendLinuxsignals to the process or use the `fg`command to bring the process back to the foreground in theshell session. You can suspend a running foreground process by using the Ctrl+Z key combination and place it back in background mode, using the `bg`command.

The `nice`and `renice`commands allow you to change the priority level of a process. By giving a process a lower priority, you allow the CPU to allocate less time to it. This comes in handy when running long processes that can take lots of CPU time.

Inadditiontocontrollingprocesseswhilethey'rerunning,youcanalsodeterminewhenapro- cessstartsonthesystem.Insteadofrunningascriptdirectlyfromthecommandlineinterface prompt,youcanscheduletheprocesstorunatanalternativetime.Youcanaccomplishthisin severaldifferentways.The`at`commandenablesyoutorunascriptonceatapresettime.The `cron`programprovides an interface that can runscripts at a regularly scheduledinterval.

Finally, the Linux system provides script files for you to use for scheduling your scripts to run whenever a user starts a new bash shell. Similarly, the startup files, such as `.bashrc`, are located in every user's home directory to provide a location to place scripts and commands that run with a new shell.

In the next chapter, we look at how to write script functions. Script functions allow you towrite code blocks once and then use them in multiple locations throughout your script.