developerWorks®

Learn Linux, 101: Customize or write simple scripts

Use the power of Linux scripting

Ian Shields
Linux Author
Freelance

18 January 2016

Learn how to customize existing scripts or write simple new bash scripts using standard shell syntax, looping and control structures, and tests for success or failure. You can use the material in this tutorial to study for the LPI 102 exam for Linux system administrator certification, or to learn for fun.

View more content in this series

Overview

In this tutorial, learn to customize existing scripts or write simple new bash scripts. Learn to:

- Use standard looping and control constructs
- Use command substitution
- Test return values from commands for success or failure
- Conditionally send mail to the superuser
- Make sure that the right shell interprets your script
- Manage the location, ownership, execution, and suid rights of scripts

Programming with the Linux shell

In this tutorial, I step beyond simple command execution and minimal testing via `&&` and `||`. I show you how to use the bash shell control structures to add the power of programming to shell scripts. You first learn how to perform various tests that you can use to make control decisions. Then I show you how to use `if`-`then`-`else`, `for`, `while`, and `case` control structures to exploit those test results. Finally, I cover important issues regarding who has the right to run your scripts and how to notify the superuser (root) when your script is not running under direct control of a user at a terminal.

About this series

This series of tutorials helps you learn Linux system administration tasks. You can also use the material in these tutorials to prepare for the Linux Professional Institute's LPIC-1: Linux Server Professional Certification exams.

See "*Learn Linux, 101*: A roadmap for LPIC-1" for a description of and link to each tutorial in this series. The roadmap is in progress and reflects the version 4.0 objectives of the LPIC-1 exams as updated April 15, 2015. As tutorials are completed, they will be added to the roadmap.

This tutorial helps you prepare for Objective 105.2 in Topic 105 of the Linux Server Professional (LPIC-1) exam 102. The objective has a weight of 4.

Prerequisites

To get the most from the tutorials in this series, you need:

- A basic knowledge of Linux
- Familiarity with GNU and UNIX® commands
- A working Linux system on which you can practice the commands covered in this tutorial

This tutorial builds on material covered in the tutorials for Topic 103 of Exam 101. In addition, you need to be familiar with the material covered in "*Learn Linux, 101*: Customize and use the shell environment."

Sometimes different versions of a program format output differently, so your results might not always look exactly like the listings and figures shown here. The examples in this tutorial are largely distribution independent. Unless otherwise noted, the examples in this article use Ubuntu 15.10, with a 4.2.0 kernel.

Variable assignment and arithmetic

When you learn any programming language, you learn how to assign values to variables. In prior tutorials in this series, you assigned string values to variables. Bash supports shell arithmetic using integers. You can evaluate an expression as an arithmetic value and assign it to a variable by using the `let` builtin. You can explicitly declare a variable as an integer variable, and then future assignments to it are evaluated as integer expressions. Listing 1 shows examples of both methods and some subtle differences.

Listing 1. Variable assignments and arithmetic

```
ian@attic-u15:~$ x=3+4
ian@attic-u15:~$ let y=5*10
ian@attic-u15:~$ declare -i z=5*4/3
ian@attic-u15:~$ echo $x $y $z
3+4 50 6
ian@attic-u15:~$ # Use declare -p to show more information
ian@attic-u15:~$ declare -p x y z
declare -- x="3+4"
declare -- y="50"
declare -i z="6"
```

Notice that only the variable `z` is declared to be integer.

You can use most of the C or C++ arithmetic operators in shell arithmetic, including bitwise and logical operators. You can use pre- and postincrement operators, and the usual C or C++ assignment operators such as `+=`, `&&=`, and `|=`. Use parentheses if you need to group operations. If you want, you can use `let` and `declare` to assign multiple variables in one line. If you want to use a variable value in an arithmetic expression, you don't need to use `$` before the variable name, although you can if you want. Listing 2 shows some more examples of arithmetic in bash.

Listing 2. More arithmetic assignment examples

```
ian@attic-u15:~$ declare -i p q r
ian@attic-u15:~$ let p=" x + 7 " q=" (y * 2**4) / 100 "
ian@attic-u15:~$ q=" 2**z - (50 /3 ) + 7%4 "
ian@attic-u15:~$ r=4
ian@attic-u15:~$ r+=" q + ( 17 > 4) "
ian@attic-u15:~$ echo $p $q $r
14 51 56
ian@attic-u15:~$ declare -p p q r
declare -i p="14"
declare -i q="51"
declare -i r="56"
ian@attic-u15:~$ let t=3 u=p+q
ian@attic-u15:~$ echo $t $u
3 65
ian@attic-u15:~$ declare -p t u
declare -- t="3"
declare -- u="65"
```

Notice that you can't have white space to the left of the `=` sign, and that anything to the right of it that includes white space must be enclosed in single or double quotation marks. You can use the `((  ))` construct to make an assignment while relaxing these rules. You do not need to escape operators between `((` and `))`. Listing 3 shows what happens if you have white space in the wrong place and how using the `((  ))` can ease the problem.

Listing 3. Arithmetic, white space, and `((  ))`

```
ian@attic-u15:~$ declare -i t
ian@attic-u15:~$ t= 3**3 % 5
3**3: command not found
ian@attic-u15:~$ t = 3**3 % 5
t: command not found
ian@attic-u15:~$ (( t = 3**3 % 5 ))
ian@attic-u15:~$ echo $t
2
ian@attic-u15:~$ # Logical expression using unescaped shell meta characters
ian@attic-u15:~$ (( u = ( 3 > 5 ) || ( 4 < 6 ) ))
ian@attic-u15:~$ echo $u
1
```

Tests and testing

Now that you know how to assign values to variables and pass parameters, you need to know how to test those values and parameters. You already know that `$?` contains the return status from a shell command. This value is also set for variable declarations and assignments, and for the tests that I am about to show you. The `test` command is a builtin command that performs various tests and sets the return status to `0` (success or true) or `1` (failure or false). Later in this tutorial, I show you how to use the return status to make decisions, such as in `if-then-else` constructs.

`test` and `[`

In the simple `add2path` function from the previous tutorial, ("*Learn Linux, 101*: Customize and use the shell environment"), I introduced you to the `test` command and showed you how to add a directory to your `PATH` variable if it is not already there. See Listing 4.

Listing 4. The `add2path` function

```
ian@attic-u15:~$ type  add2path
add2path is a function
add2path ()
{
    local augpath augdir;
    augpath=":$PATH:";
    augdir=":$1:";
    test "$augpath" = "${augpath/$augdir}" && PATH="$1:$PATH"
}
```

The `test` builtin command returns `0` (true) or `1` (false) depending on the evaluation of an expression *`expr`*. You can also use square brackets; `test` *`expr`* and `[` *`expr`* `]` are equivalent. You can examine the return value by displaying `$?`. Then, you can use the return value as you have before with && and ||. Or you can test the return value by using the various conditional constructs that I cover later in this tutorial. Listing 5 shows some simple test examples.

Listing 5. Some simple tests

```
ian@attic-u15:~$ test 3 -gt 4 && echo true || echo false
false
ian@attic-u15:~$ [ "abc" != "def" ];echo $?
0
ian@attic-u15:~$ [ "abc" = "def" ];echo $?
1
ian@attic-u15:~$ test -d "$HOME" ;echo $?
0
```

The first example in Listing 5 uses the `-gt` operator to perform an arithmetic comparison between two literal values. The second and third examples use the alternative `[ ]` syntax to compare two strings for inequality and then equality, echoing the value of `$?` in each case. The final example uses the `-d` unary operator to check that the `HOME` variable is the name of a directory.

You compare arithmetic values by using one of `-eq` (equal), `-ne` (not equal), `-lt` (less than), `-le` (less than or equal), `-gt` (greater than), or `-ge` (greater than or equal).

You use `=` to compare strings for equality, `!=` to compare strings for inequality, and `<` and `>` to determine whether the first string sorts before or after the second one. The unary operator `-z` tests for a null string; `-n` or no operator at all returns true (`0`) if a string is not empty.

The `<` and `>` operators are also used by the shell for redirection, so you must escape them by using `\<` or `\>`. Listing 6 shows some more examples of string tests.

Listing 6. More string tests

```
ian@attic-u15:~$ test "abc" = "def" ;echo $?
1
ian@attic-u15:~$ [ "abc" != "def" ];echo $?
0
ian@attic-u15:~$ [ "abc" \< "def" ];echo $?
0
ian@attic-u15:~$ [ "abc" \> "def" ];echo $?
1
ian@attic-u15:~$ [ "abc" \< "abc" ];echo $?
1
ian@attic-u15:~$ [ "abc" \> "abc" ];echo $?
1
ian@attic-u15:~$ [ -z "abc" ]; echo $?
1
ian@attic-u15:~$ [ -n "abc" ]; echo $?
0
```

You can use many tests on filesystem objects. Table 1 shows some of the common tests. If the object tested exists and has the specified attribute, the result is true (`0`).

Table 1. Common file tests

Unary operator	Characteristic
`-d`	Directory
`-e` or `-a`	Exists
`-f`	Regular file
`-h` or `-L`	Symbolic link
`-p`	Named pipe
`-r`	Readable by you
`-s`	Not empty
`-S`	Socket
`-w`	Writable by you
`-N`	Has been modified since last being read

You can also use the binary operators shown in Table 2 to compare two files.

Table 2. File-comparison tests

Binary operator	Characteristic
`-nt`	Test if file 1 is newer than file 2. The modification timestamp is used for this comparison.
`-ot`	Test if file 1 is older than file 2. The modification timestamp is used for this comparison.
`-ef`	Test if file 1 is a hard link to file 2.

You can use other tests to check things such as the permissions settings of a file. See the bash man pages for more details, or use `help test` to see brief information on the `test` builtin. You can use the `help` command for other builtins too.

You can use the unary `-o` operator to test if various shell options are set. As shown in Listing 7, `test -o option` returns true (`0`) if the option is set; otherwise, it returns false (`1`).

Listing 7. Testing shell options

```
ian@attic-u15:~$ # Setting and testing the unset option
ian@attic-u15:~$ set +o nounset
ian@attic-u15:~$ echo $MYTESTVAR

ian@attic-u15:~$ [ -o nounset ];echo $?
1
ian@attic-u15:~$ # You can also set/unset nounset using set -u or set +u
ian@attic-u15:~$ set -u
ian@attic-u15:~$ echo $MYTESTVAR
bash: MYTESTVAR: unbound variable
ian@attic-u15:~$ test -o nounset; echo $?
0
```

You use the `-a` binary option to combine expressions with logical AND, and the `-o` binary option to combine expressions with logical OR. The unary `!` operator inverts the sense of the test. Use parentheses to group expressions or override the default precedence. Remember that the shell normally runs an expression between parentheses in a subshell, so you must escape the parentheses by using `\(` and `\)`, or enclose these operators in single or double quotation marks if you don't want an expression to run in a subshell. Listing 8 illustrates the application of de Morgan's laws to an expression.

Listing 8. Combining and grouping tests

```
ian@attic-u15:~$ test "a" != "$HOME" -a 3 -ge 4 ; echo $?
1
ian@attic-u15:~$ [ ! \( "a" = "$HOME" -o 3 -lt 4 \) ]; echo $?
1
ian@attic-u15:~$ [ ! \( "a" = "$HOME" -o '(' 3 -lt 4 ')' ")" ]; echo $?
1
ian@attic-u15:~$ # Be careful. ! has higher priority that -a or -o
ian@attic-u15:~$ [ ! \( "a" = "$HOME" \) -o '(' 3 -lt 4 ')'  ]; echo $?
0
```

The `test` command is powerful, but the need to escape and the difference between string and arithmetic comparisons can make it unwieldy. Fortunately, bash has two other ways to set the return code for arithmetic and logical expressions, and they'll seem more natural if you're familiar with C, C++, or Java syntax.

Return status from `(( ))` and `[[ ]]`

The `(( ))` compound command that you saw at the beginning of this tutorial evaluates an arithmetic expression and sets the exit status to `1` if the expression evaluates to zero, or to `0` if the expression evaluates to a nonzero value. Note that the `let` command sets the return status based on whether the last argument evaluates to zero or a nonzero value. Listing 9 shows some examples.

Listing 9. Return status from `(( ))`

```
ian@attic-u15:~$ let x=2 y=2**3 z=y*3;echo $? $x $y $z
0 2 8 24
ian@attic-u15:~$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 3 8 16
ian@attic-u15:~$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 4 8 13
ian@attic-u15:~$ (( w - w )) ;echo $?
1
```

The `[[ ]]` compound command executes a conditional expression and sets the return status to `0` (true) or `1` (false). As with `(( ))`, you can use a more natural syntax with the `[[ ]]` compound command for filename and string tests. By using parentheses and logical operators, you can combine tests that the `test` command can run. See Listing 10.

Listing 10. Return status from `[[ ]]`

```
ian@attic-u15:~$ [[ ( -d "$HOME" ) && ( -w "$HOME" ) ]]; echo $?
0
ian@attic-u15:~$ [[ ( -d "$HOME" ) && ( -w "$HOME" ) ]] &&
> echo "home is a writable directory"
home is a writable directory
```

You can use the `[[ ]]` compound to do pattern matching on strings when the `==` or `!=` operators are used. The match behaves as for shell wildcard globbing, as shown in Listing 11.

Listing 11. Wildcard tests with `[[ ]]`

```
ian@attic-u15:~$ [[ "abc def .d,x--" == a[abc]*\ ?d* ]]; echo $?
0
ian@attic-u15:~$ [[ "abc def c" == a[abc]*\ ?d* ]]; echo $?
1
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* ]]; echo $?
1
```

Within `[[ ]]`, `==` and `=` have the same meaning, so you can use either one. Use `=~` if you want the pattern to be a regular expression rather than shell wildcard globbing. See Listing 12.

Listing 12. Regular expression pattern matching with `[[ ]]`

```
ian@attic-u15:~$ # Wildcard globbing does not match this pattern
ian@attic-u15:~$ [[ "abc def c" == a[abc]*\ ?d* ]]; echo $?
1
ian@attic-u15:~$ # But regular expression matching does
ian@attic-u15:~$ [[ "abc def c" =~ a[abc]*\ ?d* ]]; echo $?
0
```

You can even do arithmetic tests within `[[ ]]` compounds, but be careful. Unless they are within a nested `(( ))` compound, the `<` and `>` operators compare the operands as strings and test their order in the current collating sequence. Listing 13 illustrates this behavior with some examples.

Listing 13. Arithmetic tests within `[[ ]]`

```
ian@attic-u15:~$ # Set warning in case we use an unbound variable
ian@attic-u15:~$ # Otherwise names are interpreted as strings
ian@attic-u15:~$ set -u
ian@attic-u15:~$ # First expression is false
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* ]]; echo $?
1
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* || (( 3 > 2 )) ]]; echo $?
0
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 -gt 2 ]]; echo $?
0
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 > 2 ]]; echo $?
0
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* || a > 2 ]]; echo $?
0
ian@attic-u15:~$ [[ "abc def d,x" == a[abc]*\ ?d* || a -gt 2 ]]; echo $?
bash: a: unbound variable
ian@attic-u15:~$ # Restore default
ian@attic-u15:~$ set +u
```

Conditionals

You can accomplish a huge amount of programming with the tests that I've shown you so far and the `&&` and `||` control operators. In addition, bash includes the more familiar `if`-`then`-`else` and `case` constructs. After I show you these constructs and then looping constructs, your toolbox will be much expanded.

Using `if`-`then`-`else` statements

Although the tests that you've seen so far return only `0` or `1` values, commands can return other values. I show you more about testing those values later in this tutorial.

The bash `if` command is a compound command that tests the return status (`$?`) of a test or command and branches based on whether the return status is true (`0`) or false (not `0`). The `if` command in bash has a `then` clause containing a list of commands to be executed if the test or command returns `0`. The command also has one or more optional `elif` clauses. Each of these optional `elif` clauses has an additional test and a `then` clause with an associated list of commands. A final `else` clause, with its associated list of commands, is optional. A final `else` clause is run if neither the original test nor any of the tests used in the `elif` clauses is true. A terminal `fi`, which marks the end of the construct, is required.

Using what you have learned so far in these tutorials, you could now build a simple calculator to evaluate arithmetic expressions, as shown in Listing 14.

Listing 14. Evaluating expressions with `if`-`then`-`else`

```
ian@attic-u15:~$ function mycalc ()
> {
>   local x
>   if [ $# -lt 1 ]; then
>     echo "This function evaluates arithmetic for you if you give it some"
>   elif (( $* )); then
>   let x="$*"
>     echo "$* = $x"
>   else
>     echo "$* = 0 or is not an arithmetic expression"
>   fi
> }
ian@attic-u15:~$ mycalc 3 + 4
3 + 4 = 7
ian@attic-u15:~$ mycalc 3 + 4**3
3 + 4**3 = 67
ian@attic-u15:~$ mycalc 3 + (4**3 /2)
bash: syntax error near unexpected token `('
ian@attic-u15:~$ mycalc 3 + "(4**3 /2)"
3 + (4**3 /2) = 35
ian@attic-u15:~$ mycalc xyz
xyz = 0 or is not an arithmetic expression
ian@attic-u15:~$ mycalc xyz + 3 + "(4**3 /2)" + abc
xyz + 3 + (4**3 /2) + abc = 35
```

The calculator uses the `local` statement to declare `x` as a local variable that is available only within the scope of the `mycalc` function. The `let` builtin command has several possible options, as does the `declare` command to which it is closely related. Check the man pages for bash, or use `help let` for more information.

As you see in Listing 14, your expressions must be correctly escaped if they use shell metacharacters such as `(`, `)`, `*`, `>`, and `<`. Nevertheless, you now have a handy little calculator for evaluating arithmetic as the shell does it.

Notice the last two examples in Listing 14. It is not an error to pass `xyz` to `mycalc`, but it evaluates to `0` unless you previously assigned a value to the variable `xyz`. In the final example, the function is not smart enough to identify the character values and thus be able to warn you that `xyz` and `abc` are quietly treated as variables with a value of `0`. You could use a string pattern-matching test such as `[[ ! ("$*" == *[a-zA-Z]* ]]` (or the appropriate form for your locale) to eliminate any expression containing alphabetic characters, but that would prevent you from using shell variables as input. It would also prevent you from using hexadecimal notation in your input, because hex notation — such as `0x0f`, representing decimal 15 — can contain letters. In fact, you can use bases up to 64 in the shell (via *`base`*`#`*`value`* notation), so your input could legitimately include any alphabetic character, plus `_` and `@`. For the special cases of octal and hexadecimal, you use the more common notation of a leading 0 for octal and leading 0x or 0X for hexadecimal. Listing 15 shows some examples.

Listing 15. Calculating with different bases

```
ian@attic-u15:~$ mycalc 015
015 = 13
ian@attic-u15:~$ mycalc 0xff
0xff = 255
ian@attic-u15:~$ mycalc 29#37
29#37 = 94
ian@attic-u15:~$ mycalc 64#1az
64#1az = 4771
ian@attic-u15:~$ mycalc 64#1azA
64#1azA = 305380
ian@attic-u15:~$ mycalc 64#1azA_@
64#1azA_@ = 1250840574
ian@attic-u15:~$ mycalc 64#1az*64**3 + 64#A_@
64#1az*64**3 + 64#A_@ = 1250840574
```

Additional laundering of the input is beyond the scope of this tutorial, so use your calculator with appropriate care.

The `elif` statement is a convenience that helps you simplify indentation in your scripts. Listing 16 shows how the `type` command for the `mycalc` function displays an equivalent form for the `elif` statements of Listing 14.

Listing 16. Type mycalc

```
ian@attic-u15:~$ type mycalc
mycalc is a function
mycalc ()
{
    local x;
    if [ $# -lt 1 ]; then
        echo "This function evaluates arithmetic for you if you give it some";
    else
        if (( $* )); then
            let x="$*";
            echo "$* = $x";
        else
            echo "$* = 0 or is not an arithmetic expression";
        fi;
    fi
}
```

Case statements

Use the `case` compound command to simplify testing when you have a list of possibilities and you want to take action based on whether a value matches a particular possibility. The `case` compound is introduced by `case WORD in` and terminated by `esac` (case spelled backward). Each `case` consists of a single pattern, or multiple patterns separated by `|`, followed by `)`, a list of statements, and finally a pair of semicolons (`;;`).

For example, imagine a store that serves coffee, decaffeinated coffee (decaf), tea, or soda. The function in Listing 17 might be used to determine the response to an order.

Listing 17. Using the `case` command

```
ian@attic-u15:~$ type myorder
myorder is a function
myorder ()
{
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
        ;;
        "tea")
            echo "Hot tea on its way"
        ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
        ;;
        *)
            echo "Sorry, we don't serve that here"
        ;;
    esac
}
ian@attic-u15:~$ myorder decaf
Hot coffee coming right up
```

```
ian@attic-u15:~$ myorder tea
Hot tea on its way
ian@attic-u15:~$ myorder milk
Sorry, we don't serve that here
```

Note the use of `*` to match anything that has not already been matched.

Another bash construct similar to `case` is the `select` statement, which is not covered here. You use it to print a list of items output to a terminal, and your user selects from the list. See the bash man pages, or type `help select` to learn more about `select`.

Of course, such a simple approach to the drink-ordering system has many problems; you can't order two drinks at once, and the function doesn't handle anything but lowercase input. Can you do a case-insensitive match? The answer is yes, so I'll show you how.

Return values

Bash has a `shopt` builtin that you can use to set or unset many shell options. One such option is `nocasematch`, which, if set, tells the shell to ignore case in string matching. Your first thought might be to use the `-o` operand that you learned about with the `test` command. Unfortunately, `nocasematch` is not one of the options that you can test with `-o`, so you must take a different approach.

The tests that you learned earlier are not the only things with return values. An `if` statement, for example, tests the return value of the underlying `test` command for being true (`0`) or false (other than `0`). Even if you use a command other than a test, success is indicated by a return value of `0` and failure by a nonzero return value. The `shopt` command, like most UNIX and Linux commands, sets a return value that you can examine by using `$?`.

Armed with this knowledge, you can now test the `nocasematch` option, set it if it is not already set, and then revert the setting to the user's preference when your function terminates. The `shopt` command has four convenient options: `-pqsu` to print the current value, don't print anything, set the option, or unset the option. The `-p` and `-q` options set a return value of `0` to indicate that the shell option is set, and `1` to indicate that it is unset. The `-p` option prints out the command required to set the option to its current value, whereas the `-q` option simply sets a return value of `0` or `1`. Listing 18 shows examples of the basic usage you need for modifying the `myorder` function, using the pattern matching you saw earlier with `[[ ]]`.

Listing 18. Using `shopt`

```
ian@attic-u15:~$ # nocasematch starts out unset
ian@attic-u15:~$ shopt -p nocasematch ; echo $?
shopt -u nocasematch
1
ian@attic-u15:~$ # test it
ian@attic-u15:~$ [[ "abc" = "ABC" ]] ;echo $?
1
ian@attic-u15:~$ # set nocasematch
ian@attic-u15:~$ shopt -s nocasematch ; echo $?
0
ian@attic-u15:~$ # test the pattern again
ian@attic-u15:~$ [[ "abc" = "ABC" ]] ;echo $?
0
ian@attic-u15:~$ # restore nocasematch
ian@attic-u15:~$ shopt -u nocasematch ; echo $?
0
```

As shown in Listing 19, your modified `myorder` function can now use the return value from `shopt` to:

1. Set a local variable representing the current state of the `nocasematch` option.
2. Set the option.
3. Run the `case` command.
4. Reset the `nocasematch` option to its original value.

Listing 19. Testing return values from the `shopt` command

```
ian@attic-u15:~$ type myorder
myorder is a function
myorder ()
{
    local restorecase;
    if shopt -q nocasematch; then
        restorecase="-s";
    else
        restorecase="-u";
        shopt -s nocasematch;
    fi;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
        ;;
        "tea")
            echo "Hot tea on its way"
        ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
        ;;
        *)
            echo "Sorry, we don't serve that here"
        ;;
    esac;
    shopt $restorecase nocasematch
}
ian@attic-u15:~$ shopt -p nocasematch
shopt -u nocasematch
ian@attic-u15:~$ # nocasematch is currently unset
ian@attic-u15:~$ myorder DECAF
Hot coffee coming right up
ian@attic-u15:~$ myorder Soda
Your ice-cold soda will be ready in a moment
ian@attic-u15:~$ shopt -p nocasematch
shopt -u nocasematch
```

```
ian@attic-u15:~$ # nocasematch is unset again after running the myorder function
```

If you want your function (or script) to return a value that other functions or commands can test, use the return statement in your function. Listing 20 shows how to return `0` for a drink that you can serve and `1` if the customer requests something else.

Listing 20. Setting your own return values from functions

```
ian@attic-u15:~$ type myorder
myorder is a function
myorder ()
{
    local restorecase;
    rc=0;
    if shopt -q nocasematch; then
        restorecase="-s";
    else
        restorecase="-u";
        shopt -s nocasematch;
    fi;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
        ;;
        "tea")
            echo "Hot tea on its way"
        ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
        ;;
        *)
            echo "Sorry, we don't serve that here";
            rc=1
        ;;
    esac;
    shopt $restorecase nocasematch;
    return $rc
}
ian@attic-u15:~$ myorder coffee;echo $?
Hot coffee coming right up
0
ian@attic-u15:~$ myorder milk;echo $?
Sorry, we don't serve that here
1
```

If you don't specify your own return value, the return value is that of the last command executed. Functions and scripts have a habit of being reused in situations that you never anticipated, so it is good practice to set your own value.

Commands can return values other than `0` and `1`, and sometimes you need the extra information. For example, the `grep` command returns `0` if the pattern is matched and `1` if it is not, but it also returns `2` if the pattern is invalid or if the file specification doesn't match any files. If you need to distinguish return values other than success (`0`) or failure (nonzero), you will probably use a `case` command or perhaps an `if` command with several `elif` parts.

Command substitution

If you surround a command with `$(` and `)`, or with a pair of backticks `` ` ``, you can substitute the command's output as input to another command. This technique is called *command substitution*.

Use the `$()` form when you need to nest command substitution. That form also makes it easier to figure out what is going on, because parentheses have a left and right form but two backticks are identical. The choice is yours, and backticks are still common.

You will often use command substitution with loops (covered later under "Loops"). However, you can also use it to simplify the `myorder` function slightly. Since `shopt -p nocasematch` prints the command that you need to set the `nocasematch` option to its current value, you only need to save that output and then execute it at the end of the `case` statement. By doing so, you restore the `nocasematch` option, whether or not you changed it. Your revised function might now look like Listing 21. Try it for yourself.

Listing 21. Using command substitution instead of return value tests

```
ian@attic-u15:~$ type myorder
myorder is a function
myorder ()
{
    local restorecase=$(shopt -p nocasematch) rc=0;
    shopt -s nocasematch;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
        ;;
        "tea")
            echo "Hot tea on its way"
        ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
        ;;
        *)
            echo "Sorry, we don't serve that here";
            rc=1
        ;;
    esac;
    $restorecase;
    return $rc
}
ian@attic-u15:~$ shopt -p nocasematch
shopt -u nocasematch
ian@attic-u15:~$ myorder DECAF
Hot coffee coming right up
ian@attic-u15:~$ myorder TeA
Hot tea on its way
ian@attic-u15:~$ shopt -p nocasematch
shopt -u nocasematch
```

Debugging

If you have typed function definitions and made typing errors that left you wondering what was wrong, you might also be wondering how to debug functions. Fortunately, you can set the `-x` option to trace commands and their arguments as the shell executes them. Listing 22 shows how this option works for the `myorder` function from Listing 21.

Listing 22. Tracing execution

```
ian@attic-u15:~$ set -x
ian@attic-u15:~$ myorder tea
+ myorder tea
++ shopt -p nocasematch
+ local 'restorecase=shopt -u nocasematch' rc=0
+ shopt -s nocasematch
+ case "$*" in
+ echo 'Hot tea on its way'
Hot tea on its way
+ shopt -u nocasematch
+ return 0
ian@attic-u15:~$ set +x
+ set +x
```

You can use this technique for your aliases, functions, or scripts. If you need more information, add the `-v` option for verbose output.

Loops

Bash and other shells have three looping constructs that are somewhat similar to those in the C language. Each executes a list of commands zero or more times. The list of commands is surrounded by the words `do` and `done`, each preceded by a semicolon.

`for`
: `for` loops come in two flavors. The most common form in shell scripting iterates over a set of values, executing the command list once for each value. The set can be empty, in which case the command list is not executed. The other form is more like a traditional C `for` loop, using three arithmetic expressions to control a starting condition, step function, and end condition.

`while`
: `while` loops evaluate a condition each time the loop starts and execute the command list if the condition is true. If the condition is not initially true, the commands are never executed.

`until`
: `until` loops execute the command list and evaluate a condition each time the loop ends. If the condition is true, the loop is executed again. Even if the condition is not initially true, the commands execute at least once.

The condition tested can be a list of commands. In this case, the return value of the *last* command executed is the one used. Listing 23 illustrates the loop commands.

Listing 23. Simple `for`, `while`, and `until` loops

```
ian@attic-u15:~$ for x in abd 2 "my stuff"; do echo $x; done
abd
2
my stuff
ian@attic-u15:~$ for (( x=2; x<5; x++ )); do echo $x; done
2
3
4
ian@attic-u15:~$ let x=3; while [ $x -ge 0 ] ; do echo $x ;let x--;done
3
2
1
0
ian@attic-u15:~$ let x=3; until echo -e "x=\c"; (( x-- == 0 )) ; do echo $x ; done
x=2
x=1
x=0
```

These examples are somewhat artificial, but they illustrate the concepts. You will most often want to iterate over the parameters to a function or shell script, or a list created by command substitution.

In "*Learn Linux, 101*: Customize and use the shell environment," you discovered that the shell can refer to the list of passed parameters as `$*` or `$@` and that whether you quote these expressions or not affects how they are interpreted. Table 3 reviews the differences.

Table 3. Shell parameters for functions

Parameter	Purpose
`*`	The positional parameters starting from parameter 1. If the expansion is done within double quotation marks, the expansion is a single word with the first character of the interfield separator (IFS) special variable separating the parameters or no intervening space if IFS is null. The default IFS value is a blank, tab, and newline. If IFS is unset, the separator used is a blank, as for the default IFS.
`@`	The positional parameters starting from parameter 1. If the expansion is done within double quotation marks, each parameter becomes a single word, so that `"$@"` is equivalent to `"$1"`, `"$2"`, ... If your parameters are likely to contain embedded blanks, use this form.

Listing 24 shows a function that prints out the number of parameters and then prints the parameters according to the four alternatives.

Listing 24. A function to print parameter information

```
ian@attic-u15:~$ type testfunc
testfunc is a function
testfunc ()
{
    echo "$# parameters";
    echo Using '$*';
    for p in $*;
    do
        echo "[$p]";
    done;
    echo Using '"$*"';
    for p in "$*";
    do
        echo "[$p]";
    done;
    echo Using '$@';
    for p in $@;
    do
        echo "[$p]";
    done;
    echo Using '"$@"';
    for p in "$@";
    do
        echo "[$p]";
    done
}
```

Listing 25 shows the function in action, with an additional character added to the front of the `IFS` variable for the function execution.

Listing 25. Printing parameter information with `testfunc`

```
ian@attic-u15:~$ IFS="|${IFS}" testfunc abc "a bc" "1 2
> 3"
3 parameters
Using $*
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$*"
[abc|a bc|1 2
3]
Using $@
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$@"
[abc]
[a bc]
[1 2
3]
```

Study the differences carefully, particularly the quoted forms and the parameters that include white space such as blanks or newline characters.

The `break` and `continue` commands

Use the `break` command to exit from a loop immediately. If you have nested loops, you can specify the number of levels to break out of. For example, if you have an `until` loop inside a `for` loop inside another `for` loop and all inside a `while` loop, `break 3` immediately terminates the `until` loop and the two `for` loops, and returns control to the next instruction in the `while` loop.

Use the `continue` statement to bypass the remaining statements in the command list and go immediately to the next iteration of the loop. Listing 26 illustrates the use of `break` and `continue`.

Listing 26. Using `break` and `continue`

```
ian@attic-u15:~$ for word in red blue green yellow violet; do
> if [ "$word" = blue ]; then continue; fi
> if [ "$word" = yellow ]; then break; fi
> echo "$word"
> done
red
green
```

Revisiting `ldirs`

Remember the work that you did in "*Learn Linux, 101*: Customize and use the shell environment" to get the `ldirs` function to extract the filename from a long listing and also figure out if it was a directory or not? The final function that you developed was not too bad, but suppose you had all the information that you now have. Would you have created the same function? Perhaps not. You know how to test whether a name is a directory or not by using `[ -d $name ]`, and you know about the `for` compound. Listing 27 shows another way you might have coded the `ldirs` function.

Listing 27. Another approach to `ldirs`

```
ian@attic-u15:~$ type ldirs
ldirs is a function
ldirs ()
{
    if [ $# -gt 0 ]; then
        for file in "$@";
        do
            [ -d "$file" ] && echo "$file";
        done;
    else
        for file in *;
        do
            [ -d "$file" ] && echo "$file";
        done;
    fi;
    return 0
}
ian@attic-u15:~$ cd developerworks/
ian@attic-u15:~/developerworks$ ldirs
my first article
readme
schema
templates
tools
web
xsl
ian@attic-u15:~/developerworks$ ldirs *s* tools/*
my first article
schema
templates
tools
xsl
tools/java
ian@attic-u15:~/developerworks$ ldirs *www*
```

The new `ldirs` function quietly returns if no directories match your criteria. This might or might not be what you want. At least you now have another tool in your toolbox.

Creating scripts

Recall that the `myorder` function can handle only one drink at a time. You could now combine that single-drink function with a `for` compound to iterate through the parameters and handle multiple drinks. This is as simple as placing your function in a file and adding the `for` instruction. Listing 28 illustrates the new myorder.sh script.

Listing 28. Ordering multiple drinks with myorder.sh

```
ian@attic-u15:~$ cat myorder.sh
function myorder ()
{
    local restorecase=$(shopt -p nocasematch) rc=0;
    shopt -s nocasematch;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
        ;;
        "tea")
            echo "Hot tea on its way"
        ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
        ;;
        *)
            echo "Sorry, we don't serve that here";
            rc=1
        ;;
    esac;
    $restorecase;
    return $rc
}

for file in "$@"; do myorder "$file"; done

ian@attic-u15:~$ . myorder.sh coffee tea "milk shake"
Hot coffee coming right up
Hot tea on its way
Sorry, we don't serve that here
```

Note that by using the `.` command, the script is sourced to run in the current shell environment rather than its own shell. To run a script, you must source it, or you must use the `chmod +x` command to mark the script file executable, as illustrated in Listing 29.

Listing 29. Making a script executable

```
ian@attic-u15:~$ chmod +x myorder.sh
ian@attic-u15:~$ ./myorder.sh coffee tea "milk shake"
Hot coffee coming right up
Hot tea on its way
Sorry, we don't serve that here
```

You still must give the full or relative path to the script unless you place it in a directory that is on your `PATH`.

The `seq`, `read`, and `exec` commands

Bash and other shells have three useful commands that you will often see in scripts: `seq`, `read`, and `exec`.

The seq command

The `seq` command generates a sequence of numbers with a specified step increment. You specify up to three parameters: an ending value alone; a start and an end; or a start, an increment, and an end. The increment and start default to 1 if not specified. The increment can be negative. Use the `-s` option to specify a separator other than the default `\n`, and use the `-w` option to get equal-width numbers if you need them. You can also use the `-f` option for `printf`-style formatting. See the `seq` man pages for more details. Listing 30 shows some examples.

Listing 30. Generating sequences with seq

```
ian@attic-u15:~$ seq 3
1
2
3
ian@attic-u15:~$ seq -s " - " 7 10
7 - 8 - 9 - 10
ian@attic-u15:~$ seq -w 2 7 19
02
09
16
ian@attic-u15:~$ seq -s ' ' 2 -3 -8
2 -1 -4 -7
ian@attic-u15:~$ seq 3 2
```

Now for a more interesting example that uses several of the concepts that I've covered so far. You probably learned about prime numbers in school and maybe heard of ways to generate them including the sieve of Eratosthenes and trial division. I'll use trial division in this example. The idea is that you test a number for primality by dividing it by smaller numbers. Obviously, you only need to check if it's divisible by a smaller prime number, and the largest such prime number that you need to test can't be bigger than the square root of the number that you are testing.

Listing 31 shows my primes.sh script. The script uses testing with `[ ]`, arithmetic with `(( ))`, decisions with `if`, looping using `for`, and the `break` command to break out of a loop. I use command substitution (with backticks) to assign the output of the `seq` command as the list of value for the `for` command to process.

Listing 31. Using seq and other tools to generate primes

```
ian@attic-u15:~$ cat primes.sh
#!/bin/bash
# Find all the positive primes up to $1
declare -i lastnum=0
# primelist will contain all prime values up to
# the square root of $1
primelist="2"
# Only try to do something if we have a parameter
if [ $# -gt 0 ]; then
  (( lastnum+= $1 ))
  echo "Positive primes up to $lastnum"
  if [ $lastnum -ge 2 ]; then
      echo "2"
      # Now only look at odd numbers greater than 2
      for n in `seq 3 2  $lastnum`
      do
   # Flag this one as prime till proven otherwise
   p=0
   for t in $primelist
   do
       (( remainder = n%t ))
       if [ $remainder -eq 0 ]; then
    p=1
    # Skip to next now we know not a prime
    break
       fi
   done
   if [ $p -eq 0 ]; then
       # Found a prime
       echo $n
       if (( lastnum > (n * n) )) ; then
    primelist="$primelist $n"
       fi
    fi
      done
  fi
fi
```

Paste the code for the script into your own Linux system and try it. Listing 32 shows some sample output. Think about how you might modify this script to find primes between two different numbers, say between 10,000 and 10,500. Could you, or should you, add additional error checks or input cleansing?

Listing 32. Primes up to 30

```
ian@attic-u15:~$ ./primes.sh 30
Positive primes up to 30
2
3
5
7
11
13
17
19
23
29
```

The `read` command

The `seq` command is great if you want to iterate over a set of numbers, but what happens if you need to iterate over input from the terminal or from a file? The answer is the `read` command, which reads a line from stdin, breaks it into tokens, and assigns the tokens to one or more variables. Listing 33 shows how to read a line into three array variables and then print the results by using `for` and `seq`. After the second variable is read, the remainder of the input line is placed in the third variable, `v[3]`. If you want the whole line in one variable, use a single variable with `read`.

Listing 33. Using the `read` command

```
ian@attic-u15:~$ read v[1] v[2] v[3]The quick brown fox jumps over the lazy dog
ian@attic-u15:~$ for n in `seq 1 3`; do echo ${v[n]} ;done
The
quick
brown fox jumps over the lazy dog
```

The `read` command has several options that you use to set the line delimiter, write out a prompt before reading input, read up to a specific number of characters, and so on. Use `help read` for a brief summary or `info bash read`. On some systems, such as Ubuntu or Debian, you might need to install the `bash-doc` package to get the bash manuals in `info` format.

Now that you know how to read a single line from stdin, you can combine this with a looping structure — typically `while` to iterate over all the lines from stdin. You could try this as another approach to the `ldirs` function from Listing 27. Listing 34 shows an attempt.

Listing 34. Another approach to `ldirs`

```
ian@attic-u15:~$ type ldirs
ldirs is a function
ldirs ()
{
    if [ $# -gt 0 ]; then
        /bin/ls "$@" | while read l; do
            [ -d "$l" ] && echo "$l";
        done;
    else
        /bin/ls | while read l; do
            [ -d "$l" ] && echo "$l";
        done;
    fi;
    return 0
}
ian@attic-u15:~$ cd developerworks/
ian@attic-u15:~/developerworks$ ldirs
my first article
readme
schema
templates
tools
web
xsl
ian@attic-u15:~/developerworks$ ldirs *s* tools/*
ian@attic-u15:~/developerworks$ # Oops! No output
```

Study the output from the `ls` command, and you will find that it does not display full paths. So the revised function works fine with no parameters but might fail with parameters. If you go back to the function from the previous tutorial "*Learn Linux, 101*: Customize and use the shell environment," you will find that it suffers from the same issue, a fact that I didn't bring up at the time. The `ldirs` function in Listing 27 works better with parameters, because the input comes directly from shell globbing and wildcard substitution rather than from the formatted output of the `ls` command.

You now know how to use `read` with a `while` loop, and you have seen three different approaches to the `ldirs` function.

The `exec` command

The `exec` command has two purposes. The first is to transfer control entirely to a new program, replacing the shell that you are currently running, but without creating a new process. You might do this if you want to use the power of the bash shell to set up a complex environment for a command. After you use `exec` to replace your shell with the desired command, your user cannot get back to a shell prompt, even if the command fails. You might use `exec` where you want users to have limited and controlled access to your system — for example, in a kiosk environment or library catalog terminal.

In the second usage of `exec`, you do not specify a command. Use `exec` to input from and output to different file handles. Why would you want to do this? Suppose you want to use a `while` loop to read a file and count the total lines and the blank lines. In Listing 34, the process is done using a pipeline, where the output of the `ls` command is piped into the `while` loop. When bash runs a pipeline, it runs it in a subshell and any changes to the environment are invisible to the calling environment. Listing 35 illustrates the problem.

Listing 35. Environment variables can't be set in pipelines

```
ian@attic-u15:~$ x=3
ian@attic-u15:~$ echo "abc" | while read n; do echo $n;x=4;done
abc
ian@attic-u15:~$ echo $x
3
```

If you could redirect the input from the specified file rather than use stdin, you wouldn't need to pipe the output of `cat` through the `while` loop. Using `exec` to redirect file descriptors is your friend. Listing 36 shows a simple script to count total lines and blank lines from a specified file. Think about how you could modify the script to process multiple files.

Listing 36. Counting lines in a file

```
ian@attic-u15:~$ cat ./countlines.sh
#!/bin/bash
# Simple script to count lines in a file and also blank lines

if [ $# -gt 0 ]; then
  if [ -f "$1" -a -r "$1" ] ; then
      lines=0
      blanklines=0
      exec 3< "$1" # Redirect input to file descriptor 3
      while read line <&3 # Read from fd 3
      do {
   [ -z "$line" ] &&  (( blanklines ++  ))
   (( lines ++  ))
      }
      done
      exec 3>&- # Restore input to stdin (fd 0)
      echo "$1 has $lines lines of which $blanklines are blank"
  fi
fi
exit 0
ian@attic-u15:~$ ./countlines.sh  .bashrc
.bashrc has 120 lines of which 23 are blank
```

Specifying a shell

Now that you have some brand new shell scripts to play with, you might ask whether they work in all shells. Listing 37 shows what happens if you run the myorder.sh shell script on a Ubuntu system using first the bash shell, then the dash shell.

Listing 37. Shell differences

```
ian@attic-u15:~$ ./myorder.sh tea soda
Hot tea on its way
Your ice-cold soda will be ready in a moment
ian@attic-u15:~$ dash
$ ./myorder.sh tea soda
./myorder.sh: 1: ./myorder.sh: Syntax error: "(" unexpected
```

That's not good!

Recall from "*Learn Linux, 101*: Customize and use the shell environment" that the word `function` is optional in a bash function definition but isn't part of the POSIX shell specification. Well, dash is a smaller and lighter shell than bash, and it doesn't support that optional feature. You can't guarantee which shell your potential users might prefer, so always ensure that your script is portable to all shell environments — which can be difficult — or use the so-called shebang (`#!`) to instruct the shell to run your script in a particular shell. The shebang line must be the first line of your script, and the rest of the line contains the path to the shell that your program must run under. So you would use `#!/bin/bash` for the myorder.sh script, as shown in Listing 38.

Listing 38. Using shebang

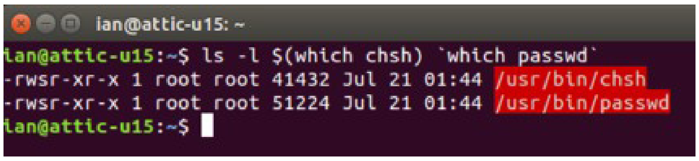
```
ian@attic-u15:~$ head -n3 myorder.sh
#!/bin/bash
function myorder ()
{
ian@attic-u15:~$ dash
$ ./myorder.sh Tea Coffee
Hot tea on its way
Hot coffee coming right up
```

You can use the `cat` command to display /etc/shells, which is the list of shells on your system. Some systems do list shells that are not installed, and some listed shells (possibly /dev/null) might be there to ensure that FTP users cannot accidentally escape from their limited environment. If you need to change your default shell, you can do so with the `chsh` command, which updates the entry for your userid in /etc/passwd.

Suid rights and script locations

In the earlier tutorial "*Learn Linux, 101*: Manage file permissions and ownership," you learned how to change a file's owner and group and how to set the suid and sgid permissions. An executable file with either of these permissions set will run in a shell with effective permissions of the file's owner (for suid) or group (for suid). Thus, the program will be able to do anything that the owner or group could do, according to which permission bit is set. There are good reasons why some programs need to do this. For example, the `passwd` program needs to update /etc/shadow, and the `chsh` command, which you use to change your default shell, needs to update /etc/passwd. If you use an alias for `ls`, listing these programs is likely to result in a red, highlighted listing to warn you, as shown in Figure 1. Both of these programs have the suid bit or bits set and thus operate as if the owner (root in this case) were running them.

Figure 1. Programs with suid permission



```
ian@attic-u15:~$ ls -l $(which chsh) `which passwd`
-rwsr-xr-x 1 root root 41432 Jul 21 01:44 /usr/bin/chsh
-rwsr-xr-x 1 root root 51224 Jul 21 01:44 /usr/bin/passwd
ian@attic-u15:~$
```

Listing 39 shows that an ordinary user can run these and update files owned by root.

Listing 39. Using suid programs

```
jenni@attic-u15:~$ passwd
Changing password for jenni.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
jenni@attic-u15:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
jenni@attic-u15:~$ chsh
Password:
Changing the login shell for jenni
Enter the new value, or press ENTER for the default
Login Shell [/bin/bash]: /bin/dash
jenni@attic-u15:~$ find /etc -mmin -4 -ls 2>/dev/null
4325377   12 drwxr-xr-x 139 root     root        12288 Dec  1 22:47 /etc
4334839    4 -rw-r--r--   1 root     root         2304 Dec  1 22:47 /etc/passwd
jenni@attic-u15:~$ grep jenni /etc/passwd
jenni:x:1001:1001:Jenni Aloi,,,:/home/jenni:/bin/dash
```

You can set suid and sgid permissions for shell scripts, but most modern shells ignore these bits for scripts. As you have seen, the shell has a powerful scripting language, with even more features than are covered in this tutorial — such as the ability to interpret and execute arbitrary expressions. These features make the shell an unsafe environment to allow such wide permission. So, if you do set suid or sgid permission for a shell script, don't expect it to be honored when the script runs.

Earlier (see Listing 29), you changed the permissions of myorder.sh to mark it executable. But to run the script, you still had to qualify the name by prepending `./`, unless you sourced it in the current shell. If you want to run a shell script by name only, it must be on your search path, as represented by the `PATH` variable. Normally, you do not want the current directory on your path, because that creates a potential security exposure. After you test your script and find it satisfactory, place it in your home directory or a directory such as ~/bin if it is a personal script, or place it in /usr/local/bin if it is to be available for others on the system. If you simply used `chmod +x` to mark it executable, it is executable by everyone (owner, group, and world), which is generally what you want. If you need to restrict the script so that only members of a certain group can run it, refer back to "*Learn Linux, 101*: Manage file permissions and ownership."

You might notice that shell programs, such as bash and dash, are usually located in /bin rather than in /usr/bin. According to the Filesystem Hierarchy Standard, /usr/bin can be on a filesystem shared among systems, and so it might not be available at initialization time. Therefore, certain functions, such as shells, should be in /bin so they are available even if /usr/bin is not yet mounted. User-created scripts do not usually need to be in /bin (or /sbin), because the programs in these directories should give you enough tools to get your system up and running to the point where you can mount the /usr filesystem.

Mailing notifications to root

Suppose your script is running an administrative task on your system in the dead of night while you're sound asleep. What happens when something goes wrong? Fortunately, it's easy to mail error information or log files to yourself or to another administrator or to root. Simply pipe the message to the `mail` command, and use the `-s` option to add a subject line, as shown in Listing 40.

Listing 40. Mailing an error message to a user

```
ian@attic-u15:~$ echo "Midnight error message" | mail -s "Admin error" ian
ian@attic-u15:~$ mail
"/var/mail/ian": 1 message 1 new
>N   1 Ian Shields        Tue Dec  1 23:08  13/423   Admin error
? 1
Return-Path: <ian@attic-u15>
X-Original-To: ian@attic-u15
Delivered-To: ian@attic-u15
Received: by attic-u15 (Postfix, from userid 1000)
id 6755C42740; Tue,  1 Dec 2015 23:08:57 -0500 (EST)
Subject: Admin error
To: <ian@attic-u15>
X-Mailer: mail (GNU Mailutils 2.99.98)
Message-Id: <20151202040857.6755C42740@attic-u15>
Date: Tue,  1 Dec 2015 23:08:57 -0500 (EST)
From: ian@attic-u15 (Ian Shields)

Midnight error message
? d
? q
Held 0 messages in /var/mail/ian
```

If you need to mail a log file, use the `<` redirection function to redirect it as input to the `mail` command. If you need to send several files, you can use `cat` to combine them and pipe the output to `mail`. In Listing 40, mail is sent to user ian, who happened to also be the one running the command, but admin scripts are more likely to direct mail to root or another administrator. As usual, consult the man pages for `mail` to learn about other options that you can specify.

This concludes your introduction to customizing and writing bash scripts.

Resources

- developerWorks Premium provides an all-access pass to powerful tools, curated technical library from Safari Books Online, conference discounts and proceedings, SoftLayer and Bluemix credits, and more.
- Use the developerWorks roadmap for LPIC-1 to find the developerWorks tutorials to help you study for LPIC-1 certification based on the LPI Version 4.0 April 2015 objectives.
- At the Linux Professional Institute website, find detailed objectives, task lists, and sample questions for the certifications. In particular, see:
  - The LPIC-1: Linux Server Professional Certification program details
  - LPIC-1 exam 101 objectives
  - LPIC-1 exam 102 objectives

  Always refer to the Linux Professional Institute website for the latest objectives.
- Shell Command Language defines the shell command language as specified by The Open Group and IEEE.
- Check out the Bash Reference Manual.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
- Follow developerWorks on Twitter.

About the author

Ian Shields

Ian Shields is a freelance Linux writer. He retired from IBM at the Research Triangle Park, NC. Ian joined IBM in Canberra, Australia, as a systems engineer in 1973, and has worked in Montreal, Canada, and RTP, NC in both systems engineering and software development. He has been using, developing on, and writing about Linux since the late 1990s. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. He enjoys orienteering and likes to travel.