

# Handling User Input

## IN THIS CHAPTER

Passing parameters

Tracking parameters

Being shifty

Working with options

Standardizing options

Getting user input

So far you've seen how to write scripts that interact with data, variables, and files on the Linux 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.

## Passing Parameters

---

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:

```
$ ./addem 10 30
```

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.

### Reading parameters

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's a simple example of using one command line parameter in a shell script:

```
$ cat test1.sh
#!/bin/bash
# using one command line parameter
#
factorial=1
for (( number = 1; number <= $1 ; number++ ))
do
    factorial=$(( $factorial * $number ))
done
echo The factorial of $1 is $factorial
$
$ ./test1.sh 5
The factorial of 5 is 120
$
```

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:

```
$ cat test2.sh
#!/bin/bash
# testing two command line parameters
#
total=$(( $1 * $2 ))
echo The first parameter is $1.
echo The second parameter is $2.
echo The total value is $total.
$
$ ./test2.sh 2 5
The first parameter is 2.
The second parameter is 5.
The total value is 10.
$
```

The shell 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:

```
$ cat test3.sh
#!/bin/bash
# testing string parameters
#
echo Hello $1, glad to meet you.
$
$ ./test3.sh Rich
Hello Rich, glad to meet you.
$
```

The shell passes the string value entered into the command line to the script. However, you'll have a problem if you try to do this with a text string that contains spaces:

```
$ ./test3.sh Rich Blum
Hello Rich, glad to meet you.
$
```

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 'Rich Blum'
Hello Rich Blum, glad to meet you.
$
$ ./test3.sh "Rich Blum"
Hello Rich Blum, glad to meet you.
$
```

#### NOTE

The quotation marks used when you pass text strings as parameters are not part of the data. They just delineate the beginning and the end of the data.

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:

```
$ cat test4.sh
#!/bin/bash
# handling lots of parameters
#
total=$(( ${10} * ${11} ])
echo The tenth parameter is ${10}
echo The eleventh parameter is ${11}
echo The total is $total
```

```
$
$ ./test4.sh 1 2 3 4 5 6 7 8 9 10 11 12
The tenth parameter is 10
The eleventh parameter is 11
The total is 110
$
```

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

### Reading the script name

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 multiple functions.

```
$ cat test5.sh
#!/bin/bash
# Testing the $0 parameter
#
echo The zero parameter is set to: $0
#
$
$ bash test5.sh
The zero parameter is set to: 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
The zero parameter is set to: ./test5.sh
$
```

There is another potential problem. When the actual string passed is the full script path, and not just the script's name, the `$0` variable gets set to the full script path and name:

```
$ bash /home/Christine/test5.sh
The zero parameter is set to: /home/Christine/test5.sh
$
```

If you want to write a script that performs different functions based on just the script's name, 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's a handy little command available that does just that. The `basename` command returns just the script's name without the path:

```
$ cat test5b.sh
#!/bin/bash
```

```
# Using basename with the $0 parameter
#
name=$(basename $0)
echo
echo The script name is: $name
#
$ bash /home/Christine/test5b.sh

The script name is: test5b.sh
$
$ ./test5b.sh

The script name is: 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:

```
$ cat test6.sh
#!/bin/bash
# Testing a Multi-function script
#
name=$(basename $0)
#
if [ $name = "addem" ]
then
    total=$(( $1 + $2 ))
#
elif [ $name = "multem" ]
then
    total=$(( $1 * $2 ))
fi
#
echo
echo The calculated value is $total
#
$
$ cp test6.sh addem
$ chmod u+x addem
$
$ ln -s test6.sh multem
$
$ ls -l *em
-rwxrw-r--. 1 Christine Christine 224 Jun 30 23:50 addem
lrwxrwxrwx. 1 Christine Christine   8 Jun 30 23:50 multem -> test6.sh
$
$ ./addem 2 5

The calculated value is 7
```

```
$  
$ ./multem 2 5  
  
The calculated value is 10  
$
```

The example creates two separate filenames from the `test6.sh` script, 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.

### Testing parameters

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

```
$ ./addem 2  
./addem: line 8: 2 + : syntax error: operand expected (error  
token is " ")  
The calculated value is  
$
```

When the script assumes there is data in a parameter variable, and no data is present, 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:

```
$ cat test7.sh  
#!/bin/bash  
# testing parameters before use  
#  
if [ -n "$1" ]  
then  
    echo Hello $1, glad to meet you.  
else  
    echo "Sorry, you did not identify yourself. "  
fi  
$  
$ ./test7.sh Rich  
Hello Rich, glad to meet you.  
$  
$ ./test7.sh  
Sorry, you did not identify yourself.  
$
```

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

## Using Special Parameter Variables

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

### Counting parameters

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 when the script was run. You can use this special variable anywhere in the script, just like a normal variable:

```
$ cat test8.sh
#!/bin/bash
# getting the number of parameters
#
echo There were $# parameters supplied.
$
$ ./test8.sh
There were 0 parameters supplied.
$
$ ./test8.sh 1 2 3 4 5
There were 5 parameters supplied.
$
$ ./test8.sh 1 2 3 4 5 6 7 8 9 10
There were 10 parameters supplied.
$
$ ./test8.sh "Rich Blum"
There were 1 parameters supplied.
$
```

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

```
$ cat test9.sh
#!/bin/bash
# Testing parameters
#
if [ $# -ne 2 ]
then
    echo
```

```
        echo Usage: test9.sh a b
        echo
    else
        total=$(( $1 + $2 ))
        echo
        echo The total is $total
        echo
    fi
#
$
$ bash test9.sh

Usage: test9.sh a b

$ bash test9.sh 10

Usage: test9.sh a b

$ bash test9.sh 10 15

The total is 25

$
```

The `if-then` statement uses the `-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.

If you think this through, you might think that because the `$#` variable contains the value of the number of parameters, using the variable `${$#}` would represent the last command line parameter variable. Try that and see what happens:

```
$ cat badtest1.sh
#!/bin/bash
# testing grabbing last parameter
#
echo The last parameter was ${$#}
$
$ ./badtest1.sh 10
The last parameter was 15354
$
```

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:



```

$ cat test10.sh
#!/bin/bash
# Grabbing the last parameter
#
params=$#
echo
echo The last parameter is $params
echo The last parameter is ${!#}
echo
#
$
$ bash test10.sh 1 2 3 4 5

The last parameter is 5
The last parameter is 5

$
$ bash test10.sh

The last parameter is 0
The last parameter is test10.sh

$

```

Perfect. This script also assigned the `$#` variable value to the variable `params` and then used that variable within the special command line parameter variable format as well. Both versions worked. It's also important to notice that, when there weren't any parameters on the command line, the `$#` value was zero, which is what appears in the `params` variable, but the `${!#}` variable returns the script name used on the command line.

## Grabbing all the data

In some situations you want to grab all the parameters provided on the command line. Instead of 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 line as 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:

```
$ cat test11.sh
#!/bin/bash
# testing $* and $@
#
echo
echo "Using the \"$*\" method: \"$*"
echo
echo "Using the \"$@\" method: \"$@"
$
$ ./test11.sh rich barbara katie jessica

Using the $* method: rich barbara katie jessica

Using the $@ method: rich barbara katie jessica
$
```

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

The following example demonstrates where the differences are:

```
$ cat test12.sh
#!/bin/bash
# testing $* and $@
#
echo
count=1
#
for param in "$*"
do
    echo "\"$*\" Parameter #$count = $param"
    count=$(( $count + 1 ))
done
#
echo
count=1
#
for param in "$@"
do
    echo "\"$@\" Parameter #$count = $param"
    count=$(( $count + 1 ))
done
$
$ ./test12.sh rich barbara katie jessica

$* Parameter #1 = rich barbara katie jessica

$@ 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.

## Being Shifty

Another tool 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` command literally shifts the command line parameters in their relative positions.

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`, the program name, remains unchanged).

This is another great way to iterate through command line parameters, especially if you don'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's a short demonstration of how this works:

```
$ cat test13.sh
#!/bin/bash
# demonstrating the shift command
echo
count=1
while [ -n "$1" ]
do
    echo "Parameter #$count = $1"
    count=$((count + 1))
    shift
done
$
$ ./test13.sh rich barbara katie jessica

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

The script 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

Be careful when working with the `shift` command. When a parameter is shifted out, its value is lost and can't be recovered.

Alternatively, you can perform a multiple location shift by providing a parameter to the `shift` command. Just provide the number of places you want to shift:

```
$ cat test14.sh
#!/bin/bash
# demonstrating a multi-position shift
#
echo
echo "The original parameters: $*"
shift 2
echo "Here's the new first parameter: $1"
$
$ ./test14.sh 1 2 3 4 5

The original parameters: 1 2 3 4 5
Here's the new first parameter: 3
$
```

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

## Working with Options

---

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.

### Finding your options

On the surface, there's nothing all that special about command line options. They appear on 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.

### Processing simple options

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.

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

```
$ cat test15.sh
#!/bin/bash
# extracting command line options as parameters
#
echo
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
$
$ ./test15.sh -a -b -c -d

Found the -a option
Found the -b option
Found the -c option
-d is not an option
$
```

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

-d is not an option
Found the -c option
Found the -a option
$
```

The `case` statement processes 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.

### Separating options from parameters

Often you'll run into situations where you'll want to use both options and parameters for a shell script. The standard way to do this in Linux is to separate the two with a special character code that tells the script when the options are finished and when the normal parameters start.

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.

To check for the double dash, simply add another entry in the `case` statement:

```
$ cat test16.sh
#!/bin/bash
# extracting options and parameters
echo
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
#
count=1
for param in $@
do
    echo "Parameter #$count: $param"
    count=$(( $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.

For the first test, try running the script using a normal set of options and parameters:

```
$ ./test16.sh -c -a -b test1 test2 test3

Found the -c option
Found the -a option
Found the -b option
test1 is not an option
```

```
test2 is not an option
test3 is not an option
$
```

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 double dash to separate the options from the parameters on the command line:

```
$ ./test16.sh -c -a -b -- test1 test2 test3

Found the -c option
Found the -a option
Found the -b option
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.

### Processing options with values

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

```
$ ./testing.sh -a test1 -b -c -d test2
```

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:

```
$ cat test17.sh
#!/bin/bash
# extracting command line options and values
echo
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option";;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift ;;
        -c) echo "Found the -c option";;
        --) shift
            break ;;
        *) echo "$1 is not an option";;
    esac
    shift
done
```

```
#
count=1
for param in "$@"
do
    echo "Parameter #$count: $param"
    count=$(( $count + 1 ))
done
$
$ ./test17.sh -a -b test1 -d

Found the -a option
Found the -b option, with parameter value test1
-d is not an option
$
```

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` variable. 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 -b test1 -a -d
Found the -b option, with parameter value test1
Found the -a option
-d is not an option
$
```

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
-ac is not an option
$
```

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.

## Using the `getopt` command

The `getopt` command is a great tool to have handy when processing command line options and parameters. It reorganizes the command line parameters to make parsing them in your script easier.



### Looking at the command format

The `getopt` command can 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:

```
getopt optstring parameters
```

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, list each command line option letter you're going to use in your script in the *optstring*. Then place a colon after each option letter that requires a parameter value. The `getopt` command parses the supplied parameters based on the *optstring* you define.

#### 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 commands confused. Be careful!

Here's a simple example of how `getopt` works:

```
$ getopt ab:cd -a -b test1 -cd test2 test3
-a -b test1 -c -d -- test2 test3
$
```

The *optstring* defines four valid option letters, a, b, c, and d. A colon (:) is placed behind the letter b in order to require option b to have a parameter value. When the `getopt` command runs, it examines the provided parameter list (`-a -b test1 -cd test2 test3`) and parses it based on the supplied *optstring*. Notice that it automatically separated the `-cd` options into two separate options and inserted the double dash to separate the additional parameters on the line.

If you specify a parameter option not in the *optstring*, by default the `getopt` command produces an error message:

```
$ getopt ab:cd -a -b test1 -cde test2 test3
getopt: invalid option -- e
-a -b test1 -c -d -- test2 test3
$
```

If you prefer to just ignore the error messages, use `getopt` with the `-q` option:

```
$ getopt -q ab:cd -a -b test1 -cde test2 test3
-a -b 'test1' -c -d -- 'test2' 'test3'
$
```

Note that the `getopt` command options must be listed before the *optstring*. Now you should be ready to use this command in your scripts to process command line options.

### Using getopt in your scripts

You can use the `getopt` command 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 formatted version produced by the `getopt` command. The way to do that is to use the `set` command.

You saw the `set` command back in Chapter 6. The `set` command works with the different variables in the shell.

One of the `set` command options is the double dash (`--`). The double dash instructs `set` to replace the command line parameter variables with the values on the `set` command's command line.

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

```
set -- $(getopt -q ab:cd "$@")
```

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

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

```
$ cat test18.sh
#!/bin/bash
# Extract command line options & values with getopt
#
set -- $(getopt -q ab:cd "$@")
#
echo
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break ;;
        *) echo "$1 is not an option";;
    esac
    shift
done
```

```

done
#
count=1
for param in "$@"
do
    echo "Parameter #${count}: $param"
    count=$((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.

Now when you run the script with complex options, things work much better:

```

$ ./test18.sh -ac

Found the -a option
Found the -c option
$

```

And of course, all the original features work just fine as well:

```

$ ./test18.sh -a -b test1 -cd test2 test3 test4

Found the -a option
Found the -b option, with parameter value 'test1'
Found the -c option
Parameter #1: 'test2'
Parameter #2: 'test3'
Parameter #3: 'test4'
$

```

Now things are looking pretty fancy. However, there's still one small bug that lurks in the `getopt` command. Check out this example:

```

$ ./test18.sh -a -b test1 -cd "test2 test3" test4

Found the -a option
Found the -b option, with parameter value 'test1'
Found the -c option
Parameter #1: 'test2
Parameter #2: test3'
Parameter #3: 'test4'
$

```

The `getopt` command isn'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.

### Advancing to getopt

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 found in 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. This makes it great for using in loops to parse all the parameters on the command line.

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

```
getopts optstring variable
```

The *optstring* value is 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` command uses two environment variables. The `OPTARG` environment variable contains the 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's look at a simple example that uses the `getopts` command:

```
$ cat test19.sh
#!/bin/bash
# simple demonstration of the getopt command
#
echo
while getopts :ab:c opt
do
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG";;
        c) echo "Found the -c option" ;;
        *) echo "Unknown option: $opt";;
    esac
done
$
```

```
$ ./test19.sh -ab test1 -c

Found the -a option
Found the -b option, with value test1
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` command offers several nice features. For starters, you can include spaces in your parameter values:

```
$ ./test19.sh -b "test1 test2" -a

Found the -b option, with value test1 test2
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

Found the -a option
Found the -b option, with value test1
$
```

The `getopts` command correctly parsed the `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

Unknown option: ?
$
$ ./test19.sh -acde

Found the -a option
Found the -c option
Unknown option: ?
Unknown option: ?
$
```

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

The `getopts` command knows when to stop processing options and leave the parameters for you to process. 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:

```
$ cat test20.sh
#!/bin/bash
# Processing options & parameters with getopts
#
echo
while getopts :ab:cd opt
do
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG" ;;
        c) echo "Found the -c option" ;;
        d) echo "Found the -d option" ;;
        *) echo "Unknown option: $opt" ;;
    esac
done
#
shift ${OPTIND - 1}
#
echo
count=1
for param in "$@"
do
    echo "Parameter $count: $param"
    count=$((count + 1))
done
#
$
$ ./test20.sh -a -b test1 -d test2 test3 test4

Found the -a option
Found the -b option, with value test1
Found the -d option

Parameter 1: test2
Parameter 2: test3
Parameter 3: test4
$
```

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

## Standardizing Options

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, a few letter options have achieved a somewhat standard meaning in the Linux world. If you leverage these options in your shell script, your scripts will be more user-friendly.

Table 14-1 shows some of the common meanings for command line options used in Linux.

**TABLE 14-1 Common Linux Command Line Options**

Option	Description
-a	Shows all objects
-c	Produces a count
-d	Specifies a directory
-e	Expands an object
-f	Specifies a file to read data from
-h	Displays a help message for the command
-i	Ignores text case
-l	Produces a long format version of the output
-n	Uses a non-interactive (batch) mode
-o	Specifies an output file to redirect all output to
-q	Runs in quiet mode
-r	Processes directories and files recursively
-s	Runs in silent mode
-v	Produces verbose output
-x	Excludes an object
-y	Answers yes to all questions

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

# Getting User Input

---

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.

## Reading basics

The `read` command accepts 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:

```
$ cat test21.sh
#!/bin/bash
# testing the read command
#
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program. "
#
$
$ ./test21.sh
Enter your name: Rich Blum
Hello Rich Blum, welcome to my program.
$
```

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 your scripts a more form-like appearance.

In fact, the `read` command includes the `-p` option, which allows you to specify a prompt directly in the `read` command line:

```
$ cat test22.sh
#!/bin/bash
# testing the read -p option
#
read -p "Please enter your age: " age
days=$(( $age * 365 ))
echo "That makes you over $days days old! "
#
$
$ ./test22.sh
Please enter your age: 10
That makes you over 3650 days old!
$
```



You'll notice in the first example that when a name was entered, the `read` command assigned both the first name and last name to the same variable. The `read` command 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:

```
$ cat test23.sh
#!/bin/bash
# entering multiple variables
#
read -p "Enter your name: " first last
echo "Checking data for $last, $first..."
$
$ ./test23.sh
Enter your name: Rich Blum
Checking data for Blum, Rich...
$
```

You can also specify no variables on the `read` command line. If you do that, the `read` command places any data it receives in the special environment variable `REPLY`:

```
$ cat test24.sh
#!/bin/bash
# Testing the REPLY Environment variable
#
read -p "Enter your name: "
echo
echo Hello $REPLY, welcome to my program.
#
$
$ ./test24.sh
Enter your name: Christine

Hello Christine, welcome to my program.
$
```

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

## Timing out

Be careful when using the `read` command. 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 `-t` option to specify a timer. The `-t` option specifies the number of seconds for the `read` command to wait for input. When the timer expires, the `read` command returns a non-zero exit status:

```
$ cat test25.sh
#!/bin/bash
```

```
# timing the data entry
#
if read -t 5 -p "Please enter your name: " name
then
    echo "Hello $name, welcome to my script"
else
    echo
    echo "Sorry, too slow! "
fi
$
$ ./test25.sh
Please enter your name: Rich
Hello Rich, welcome to my script
$
$ ./test25.sh
Please enter your name:
Sorry, too slow!
$
```

Because the `read` command exits with a non-zero exit status if the timer expires, it's easy to use the standard structured statements, such as an `if-then` statement or a `while` loop to track what happened. In this example, when the timer expires, the `if` statement fails, and the shell executes the commands in the `else` section.

Instead of 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:

```
$ cat test26.sh
#!/bin/bash
# getting just one character of input
#
read -n1 -p "Do you want to continue [Y/N]? " answer
case $answer in
Y | y) echo
        echo "fine, continue on...";;
N | n) echo
        echo OK, goodbye
        exit;;
esac
echo "This is the end of the script"
$
$ ./test26.sh
Do you want to continue [Y/N]? Y
fine, continue on...
This is the end of the script
$
$ ./test26.sh
```

```
Do you want to continue [Y/N]? n
OK, goodbye
$
```

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

## Reading with no display

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 `-s` option prevents the data entered in the `read` command from being displayed on the monitor; actually, the data is displayed, but the `read` command sets the text color to the same as the background color. Here's an example of using the `-s` option in a script:

```
$ cat test27.sh
#!/bin/bash
# hiding input data from the monitor
#
read -s -p "Enter your password: " pass
echo
echo "Is your password really $pass? "
$
$ ./test27.sh
Enter your password:
Is your password really T3st1ng?
$
```

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

## Reading from a file

Finally, you can also use the `read` command to read data stored in a file on the Linux system. Each call to the `read` command reads a single line of text from the file. When no more lines are left in the file, the `read` command exits with a non-zero exit status.

The tricky part is getting the data from the file to the `read` command. The most common method is to pipe the result of the `cat` command of the file directly to a `while` command that contains the `read` command. Here's an example:

```
$ cat test28.sh
#!/bin/bash
```

```
# reading data from a file
#
count=1
cat test | while read line
do
    echo "Line $count: $line"
    count=$((count + 1))
done
echo "Finished processing the file"
$
$ cat test
The quick brown dog jumps over the lazy fox.
This is a test, this is only a test.
O Romeo, Romeo! Wherefore art thou Romeo?
$
$ ./test28.sh
Line 1: The quick brown dog jumps over the lazy fox.
Line 2: This is a test, this is only a test.
Line 3: O Romeo, Romeo! Wherefore art thou Romeo?
Finished processing the file
$
```

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

## Summary

---

This chapter showed three methods for retrieving data from the script user. 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 parameters as separate words. These variables come in handy when you're trying to process 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.

The bash shell provides three ways to handle command line options.

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 appears on the command line.

Another way 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 parameters in 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` command allows your scripts to query users for information and wait. The `read` command places any data entered by the script user into one or more variables, which you can use within the script.

Several options 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, and requesting a specific number of input characters.

In the next chapter, we look further into how bash shell scripts output data. So far, you've seen how to display data on the monitor and redirect it to a file. Next, we explore a few other options that you have available not only to direct data to specific locations but also to direct specific types of data to specific locations. This will help make your shell scripts look professional!