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

Chapter 13

Handling User Input

In This Chapter

- Command line parameters
- Special parameter variables
- 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.

Command Line 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 parameters entered in a command line. This even includes the name of the program the shell executes. The positional parameter variables are standard numbers, with *$0* being the name of the program, *$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
#!/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 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
#!/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 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 this example, the command line parameters used were both numerical values. You can also use text strings in the command line:

```
$ cat test3
#!/bin/bash
```

```
# testing string parameters


echo Hello $1, glad to meet you.
$
$ ./test3 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 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 'Rich Blum'
Hello Rich Blum, glad to meet you.
$
$ ./test3 "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
#!/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 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 Program Name

You can use the *$0* parameter to determine the name of the program that the shell started from the command line. This can come in handy if you're writing a utility that can have multiple functions. However, there's a small problem that you'll have to deal with. Look what happens in this simple example:

```
$ cat test5
#!/bin/bash
# testing the $0 parameter


echo The command entered is: $0
$
$ ./test5
The command entered is: ./test5
$
$ /home/rich/test5
The command entered is: /home/rich/test5
$
```

When the actual string passed in the *$0* variable is the entire script path, then the entire path will be used for the program, and not just the program name.

If you want to write a script that performs different functions based on the name of the script run from the command line, you'll have to do a little work. You need to be able to strip off whatever path is used to run the script from the command line.

Fortunately, there's a handy little command available for you that does just that. The basename command returns just the program name without the path. Let's modify the example script and see how this works:

```
$ cat test5b
#!/bin/bash
# using basename with the $0 parameter


name=‘basename $0’
echo The command entered is: $name
$
$ ./test5b
The command entered is: test5b
$
$ /home/rich/test5b
The command entered is: test5b
$
```

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

```
$ cat test6
#!/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 The calculated value is $total
$
$ chmod u+x test6
$ cp test6 addem
$ ln -s test6 multem
$ ls -l
-rwxr--r--    1 rich     rich   211 Oct 15 18:00 addem
lrwxrwxrwx    1 rich     rich     5 Oct 15 18:01 multem -> test6
-rwxr--r--    1 rich     rich   211 Oct 15 18:00 test6
$
$ ./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` code, one by just copying the file and the other by using a link to create the new file. In both cases, the script determines the base name of the script and performs the appropriate function based on that value.

Testing Parameters

You need to be careful when using command line parameters in your shell scripts. If the script runs 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 there isn't, most likely you'll get an error message from your script. This is a poor way to write scripts. It's always a good idea to check your parameters to make sure the data is there before using it:

```
$ cat test7
#!/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 Rich
Hello Rich, glad to meet you.
$
$ ./test7
Sorry, you did not identify yourself.
$
```

In this example, the `-n` parameter was used in the `test` command to check if there was data in the command line parameter. In the next section, you'll see there is yet another way to check for command line parameters.

Special Parameter Variables

There are a few special variables available in the bash shell, which track command line parameters. This section describes what they are, and how to use them.

Counting Parameters

As you saw in the last section, it's often a good idea to verify command line parameters before using them in your script. For scripts that use multiple command line parameters, this can get tedious.

Instead of testing each parameter, you can just 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
#!/bin/bash
# getting the number of parameters

echo There were $# parameters supplied.
$
$ ./test8
There were 0 parameters supplied.
$
$ ./test8 1 2 3 4 5
There were 5 parameters supplied.
$
$ ./test8 1 2 3 4 5 6 7 8 9 10
There were 10 parameters supplied.
$
$ ./test8 “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
#!/bin/bash
# testing parameters

if [ $# -ne 2 ]
then
  echo Usage: test9 a b
else
  total=$[ $1 + $2 ]
  echo The total is $total
fi
$
$ ./test9
Usage: test9 a b
$
$ ./test9 10
Usage: test9 a b
$
$ ./test9 10 15
The total is 25
```

```
$
$ ./test9 10 15 20
Usage: test9 a b
$
```

The `if-then` statement uses the `test` command to perform a numeric test of the number of parameters supplied on the command line. If the correct number of parameters isn't present, you can print an error message that shows 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 *$#* variable contains the value of the number of parameters, then using the variable *${$#}* would represent the last command line parameter variable. Try that out and see what happens:

```
$ cat badtest1
#!/bin/bash
# testing grabbing last parameter


echo The last parameter was ${$#}
$
$ ./badtest1 10
The last parameter was 15354
$
```

Wow, what happened here? Obviously, something wrong happened. 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
#!/bin/bash
# grabbing the last parameter


params=$#
echo The last parameter is $params
echo The last parameter is ${!#}
$
$ ./test10 1 2 3 4 5
The last parameter is 5
The last parameter is 5
$
$ ./test10
The last parameter is 0
The last parameter is ./test10
$
```

Perfect. This test 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

There are situations where you'll want to just grab all of the parameters provided on the command line and iterate through all of them. Instead of having to mess with using the *$#* variable to determine how many parameters are on the command line, then having to loop through all of them, you can use a couple of other special variables.

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

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

The *$@* variable, on the other hand, takes all of the parameters supplied on the command line as separate words in the same string. It allows you to iterate through the value, 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 take a look, so you can see the difference between the two:

```
$ cat test11
#!/bin/bash
# testing $* and $@


echo “Using the \$* method: $*”
echo “Using the \$@ method: $@”
$
$ ./test11 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 of the command line parameters provided at once.

The following example demonstrates where the differences are:

```
$ cat test12
#!/bin/bash
# testing $* and $@


count=1
for param in “$*”
do
```

```
  echo "\$* Parameter #$count = $param"
  count=$[ $count + 1 ]
done

count=1
for param in "$@"
do
  echo "\$@ Parameter #$count = $param"
  count=$[ $count + 1 ]
done
$
$ ./test12 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 *$** variable treated all of the parameters as a single parameter, while the *$@* 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 "downgrades" each parameter variable one position 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
#!/bin/bash
# demonstrating the shift command
```

```
count=1
while [ -n “$1” ]
do
  echo “Parameter #$count = $1”
  count=$[ $count + 1 ]
  shift
done
$
$ ./test13 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 of the parameters one position.

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
#!/bin/bash
# demonstrating a multi-position shift

echo “The original parameters: $*”
shift 2
echo “Here's the new first parameter: $1”
$
$ ./test14 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.

Caution

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

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 different 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 that you process command line parameters.

Processing Simple Options

In the `test13` 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 to determine when a parameter is formatted as an option:

```
$ cat test15
#!/bin/bash
# extracting command line options as parameters


while [ -n “$1” ]
do
  case “$1” in
  -a) echo “Found the -a option” ;;
  -b) echo vFound the -b option”;;
  -c) echo “Found the -c option” ;;
  *) echo “$1 is not an option”;;
  esac
  shift
done
$
$ ./test15 -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 what order the options are presented on the command line:

```
$ ./test15 -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 done 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
#!/bin/bash
# extracting options and parameters

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 -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 -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 -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
#!/bin/bash
# extracting command line options and values

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 2;;
  -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 -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. Since the parameter being processed is *$1*, you know that the additional parameter value is located in *$2* (because all of 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 two positions.

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 -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 won't work if you try to combine multiple options in one parameter:

```
$ ./test17 -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.

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 options optstring parameters
```

The *optstring* is the key to the process. It defines the valid option letters 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.

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`. It also defines that the option letter `b` requires a parameter value. When the `getopt` command runs, it examines the provided parameter list 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 an 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 the `-q` option with the command:

```
$ 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 5. The `set` command works with the different variables in the shell. Chapter 5 showed how to use the `set` command to display all of the system environment variables.

One of the options of the `set` command is the double dash, which instructs it 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 -- 'getopts -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
#!/bin/bash
# extracting command line options and values with getopt

set -- 'getopt -q ab:c "$@"'
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
$
```

You'll notice this is basically the same script as in `test17`. 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 -ac
Found the -a option
Found the -c option
$
```

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

```
$ ./test18 -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 -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. It interpreted the space as the parameter separator, instead of following the double quotation marks and combining the two values into one parameter. Fortunately, there's yet another solution that solves this problem.

The More Advanced getopts

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

Unlike `getopt`, which produces one output for all of 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 of the parameters on the command line.

The format of the `getopts` command is:

```
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.

There are two environment variables that the *getopts* command uses. 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 take a look at a simple example that uses the `getopts` command:

```
$ cat test19
#!/bin/bash
# simple demonstration of the getopts command


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 -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 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 also strips off the leading dash, so you don't need them in the `case` definitions.

There are several nice features in the `getopts` command. For starters, you can now include spaces in your parameter values:

```
$ ./test19 -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 -abtest1
Found the -a option
Found the -b option, with value test1
$
```

The `getopts` command correctly parsed the `test1` value from the `-b` option. Yet another nice feature of the `getopts`command is that it bundles any undefined option that it finds in the command line into a single output, the question mark:

```
$ ./test19 -d
Unknown option: ?
$
$ ./test19 -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 just use the *`OPTIND`* value with the `shift` command to move to the parameters:

```
$ cat test20
#!/bin/bash
```

```
# processing options and parameters with getopts

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 ]

count=1
for param in “$@”
do
  echo “Parameter $count: $param”
  count=$[ $count + 1 ]
done
$
$ ./test20 -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 of 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, there are a few letter options that have achieved somewhat of a standard meaning in the Linux world. If you leverage these options in your shell script, it will make your scripts more user-friendly.

Table 13.1 shows some of the common meanings for command line options used in Linux.

Table 13.1 Common Linux Command Line Options

Option	Description
-a	Show all objects.
-c	Produce a count.
-d	Specify a directory.
-e	Expand an object.
-f	Specify a file to read data from.
-h	Display a help message for the command.
-i	Ignore text case.
-l	Produce a long format version of the output.
-n	Use a non-interactive (batch) mode.
-o	Specify an output file to redirect all output to.
-q	Run in quiet mode.
-r	Process directories and files recursively.
-s	Run in silent mode.
-v	Produce verbose output.
-x	Exclude and object.
-y	Answer 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

While 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. There are times when 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.

Basic Reading

The `read` command accepts input from the standard input (the keyboard) or from another file descriptor (see Chapter 14). After receiving the input, the `read` command places the data into a standard variable. Here's the `read` command at its simplest:

```
$ cat test21
#!/bin/bash
# testing the read command
```

```
echo -n “Enter your name: ”
read name
echo “Hello $name, welcome to my program. ”
$
$ ./test21
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
#!/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
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 will assign 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
#!/bin/bash
# entering multiple variables

read -p “Enter your name: ” first last
echo “Checking data for $last, $first…”
$
$ ./test23
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
#!/bin/bash
# testing the REPLY environment variable


read -p “Enter a number: ”
factorial=1
for (( count=1; count <= $REPLY; count++ ))
do
  factorial=$[ $factorial * $count ]
done
echo “The factorial of $REPLY is $factorial”
$
$ ./test24
Enter a number: 5
The factorial of 5 is 120
$
```

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

Timing Out

There's a danger when using the read command. It's quite possible that your script will get stuck waiting for the script user to enter data. If the script must go on regardless of whether there was any data 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
#!/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
Please enter your name: Rich
```

```
Hello Rich, welcome to my script
$
$ ./test25
Please enter your name:
Sorry, too slow!
$
```

Since 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
#!/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
Do you want to continue [Y/N]? Y
fine, continue on…
This is the end of the script
$
$ ./test26
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. There's no need to press the Enter key.

Silent Reading

There are times when 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 will 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
#!/bin/bash
# hiding input data from the monitor


read -s -p “Enter your password: ” pass
echo
echo “Is your password really $pass? ”
$
$ ./test27
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 there are no more lines left in the file, the `read` command will exit with a non-zero exit status.

The tricky part of this is getting the data from the file to the `read` command. The most common method for doing this 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 of how to do this:

```
$ cat test28
#!/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
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 different 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.

There are three special variables that you can use 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 of the parameters as a single string, and the *$@* variable contains all of the parameters as separate words. These variables come in handy when trying to process long parameter lists.

Besides parameters, your script users can also 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!

Chapter 14

Presenting Data

In This Chapter

- Revisiting redirection
- Standard input and output
- Reporting errors
- Throwing away data
- Creating log files

So far the scripts shown in this book display information either by echoing data to the monitor or by redirecting data to a file. Chapter 10 demonstrated how to redirect the output of a command to a file. This chapter expands on that topic by showing you how you can redirect the output of your script to different locations on your Linux system.

Understanding Input and Output

So far, you've seen two methods for displaying the output from your scripts:

Chapter 18

Introducing sed and gawk

In This Chapter

- Text manipulation
- The sed editor basics

By far, one of the most common functions that people use shell scripts for is to work with text files. Between examining log files, reading configuration files, and handling data elements, shell scripts can help automate the mundane tasks of manipulating any type of data contained in text files. However, trying to manipulate the contents of text files using just shell script commands can be somewhat awkward. If you perform any type of data manipulation in your shell scripts, you'll want to become familiar with the sed and gawk tools available in Linux. These tools can greatly simplify any data-handling tasks you need to perform.

Text Manipulation

Chapter 9 showed you how to edit text files using different editor programs available in the Linux environment. These editors enable you to easily manipulate text contained in a text file by using simple commands or mouse clicks.

There are times, however, when you'll find yourself wanting to manipulate text in a text file on the fly, without having to pull out a full-fledged interactive text editor. In

these situations, it would be useful to have a simple command line editor that could easily format, insert, modify, or delete text elements automatically.

The Linux system provides two common tools for doing just that. This section describes the two most popular command line editors used in the Linux world, sed and gawk.

The sed Editor

The sed editor is called a *stream editor*, as opposed to a normal interactive text editor. In an interactive text editor, such as vim, you interactively use keyboard commands to insert, delete, or replace text in the data. A stream editor edits a stream of data based on a set of rules you supply ahead of time, before the editor processes the data.

The sed editor can manipulate data in a data stream based on commands you either enter into the command line or store in a command text file. It reads one line of data at a time from the input, matches that data with the supplied editor commands, changes data in the stream as specified in the commands, and then outputs the new data toSTDOUT. After the stream editor matches all of the commands against a line of data, it reads the next line of data and repeats the process. After the stream editor processes all of the lines of data in the stream, it terminates.

Because the commands are applied sequentially line by line, the sed editor has to make only one pass through the data stream to make the edits. This makes the sed editor much faster than an interactive editor, and allows you to quickly make changes to data in a file on-the-fly.

The format for using the sed command is:

```
sed options script file
```

The options parameters allow you to customize the behavior of the sed command, and include the options shown inTable 18.1.

Table 18.1 The sed Command Options

Option	Description
-e script	Add commands specified in the script to the commands run while processing the input.
-f file	Add the commands specified in the file to the commands run while processing the input.
-n	Don't produce output for each command, but wait for the print command.

The script parameter specifies a single command to apply against the stream data. If more than one command is required, you must use either the -e option to specify them in the command line or the -f option to specify them in a separate file. Numerous commands are available for manipulating data. We'll examine some of the basic commands used by the sed editor later in this chapter, and then look at some of the more advanced commands in Chapter 20.

Defining an Editor Command in the Command Line

By default, the sed editor applies the specified commands to the STDIN input stream. This allows you to pipe data directly to the sed editor for processing. Here's a quick example demonstrating how to do this:

```
$ echo “This is a test” | sed ‘s/test/big test/’
This is a big test
$
```

This example uses the s command in the sed editor. The s command substitutes a second text string for the first text string pattern specified between the forward slashes. In this example, the words `big test` were substituted for the word `test`.

When you run this example, it should display the results almost instantaneously. That's the power of using thesed editor. You can make multiple edits to data in about the same time it takes for some of the interactive editors just to start up.

Of course, this simple test just edited one line of data. You should get the same speedy results when editing complete files of data:

```
$ cat data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
$ sed ‘s/dog/cat/’ data1
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
$
```

The sed command executes and returns the data almost instantaneously. As it processes each line of data, the results are displayed. You'll start seeing results before the sed editor completes processing the entire file.

It's important to note that the sed editor doesn't modify the data in the text file itself. It only sends the modified text to STDOUT. If you look at the text file, it still contains the original data:

```
$ cat data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

Using Multiple Editor Commands in the Command Line

To execute more than one command from the sed command line, just use the -e option:

```
$ sed -e ‘s/brown/green/; s/dog/cat/’ data1
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
```

```
The quick green fox jumps over the lazy cat.
$
```

Both commands are applied to each line of data in the file. The commands must be separated with a semicolon, and there shouldn't be any spaces between the end of the command and the semicolon.

Instead of using a semicolon to separate the commands, you can use the secondary prompt in the bash shell. Just enter the first single quotation mark to open the script, and bash will continue to prompt you for more commands until you enter the closing quotation mark:

```
$ sed -e ‘
> s/brown/green/
> s/fox/elephant/
> s/dog/cat/’ data1
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

You must remember to finish the command on the same line that the closing single quotation mark appears. Once the bash shell detects the closing quotation mark, it will process the command. Once it starts, the sed command applies each command you specified to each line of data in the text file.

Reading Editor Commands from a File

Finally, if you have lots of sed commands you want to process, it is often easier to just store them in a separate file. Use the -f option to specify the file in the sed command:

```
$ cat script1
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1 data1
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

In this case, you don't put a semicolon after each command. The sed editor knows that each line contains a separate command. As with entering commands on the command line, the sed editor reads the commands from the specified file and applies them to each line in the data file.

We'll be looking at some other sed editor commands that will come in handy for manipulating data in the “The sedEditor Basics” section. Before that, let's take a quick look at the other Linux data editor.

```
bin         /bin/sh
...
Samantha     /bin/bash
Timothy      /bin/sh
Christine    /bin/sh
This concludes the listing
$
```

As expected, the BEGIN script created the header text, the program script processed the information from the specified data file (the /etc/passwd file), and the END script produced the footer text.

This gives you a small taste of the power available when you use simple gawk scripts. Chapter 21 describes some more basic programming principles available for your gawk scripts, along with some even more advanced programming concepts you can use in your gawk program scripts to create professional looking reports from even the most cryptic data files.

The sed Editor Basics

The key to successfully using the sed editor is to know its myriad of commands and formats, which help you customize your text editing. This section describes some of the basic commands and features you can incorporate into your script to start using the sed editor.

More Substitution Options

You've already seen how to use the s command to substitute new text for the text in a line. However, a few additional options are available for the substitute command that can help make your life easier.

Substitution Flags

There's a caveat to how the substitute command replaces matching patterns in the text string. Watch what happens in this example:

```
$ cat data5
This is a test of the test script.
This is the second test of the test script.
$
$ sed 's/test/trial/' data5
This is a trial of the test script.
This is the second trial of the test script.
$
```

The substitute command works fine in replacing text in multiple lines, but by default, it only replaces the first occurrence in each line. To get

the substitute command to work on different occurrences of the text, you must use a*substitution flag*. The substitution flag is set after the substitution command strings:

```
s/pattern/replacement/flags
```

There are four types of substitution flags available:

- A number, indicating the pattern occurrence for which new text should be substituted.
- g—Indicates that new text should be substituted for all occurrences of the existing text.
- p—Indicates that the contents of the original line should be printed.
- w *file*—Write the results of the substitution to a file.

In the first type of substitution, you can specify which occurrence of the matching pattern the sed editor should substitute new text for:

```
$ sed 's/test/trial/2' data5
This is a test of the trial script.
This is the second test of the trial script.
$
```

As a result of specifying a 2 as the substitution flag, the sed editor only replaces the pattern in the second occurrence in each line. The g substitution flag enables you to replace every occurrence of the pattern in the text:

```
$ sed 's/test/trial/g' data5
This is a trial of the trial script.
This is the second trial of the trial script.
$
```

The p substitution flag prints a line that contains a matching pattern in the substitute command. This is most often used in conjunction with the -n sed option:

```
$ cat data6
This is a test line.
This is a different line.
$
$ sed -n 's/test/trial/p' data5
This is a trial line.
$
```

The -n option suppresses output from the sed editor. However, the p substitution flag outputs any line that has been modified. Using the two in combination produces output only for lines that have been modified by the substitute command.

The w substitution flag produces the same output but stores the output in the specified file:

```
$ sed 's/test/trial/w test' data6
This is a trial line.
This is a different line.
$
$ cat test
This is a trial line.
$
```

The normal output of the sed editor appears in STDOUT, but only the lines that include the matching pattern are stored in the specified output file.

Replacement Characters

There are times when you run across characters in text strings that aren't easy to use in the substitution pattern. One popular example in the Linux world is the forward slash.

Substituting pathnames in a file can get awkward. For example, if you wanted to substitute the C shell for the bash shell in the /etc/passwd file, you'd have to do this:

```
$ sed 's/\/bin\/bash/\/bin\/csh/' /etc/passwd
```

Because the forward slash is used as the string delimiter, you must use a backslash to escape it if it appears in the pattern text. This often leads to confusion and mistakes.

To solve this problem, the sed editor allows you to select a different character for the string delimiter in the substitute command:

```
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
```

In this example, the exclamation point is used for the string delimiter, making the pathnames much easier to read and understand.

Using Addresses

By default, the commands you use in the sed editor apply to all lines of the text data. If you only want to apply a command to a specific line, or a group of lines, you must use *line addressing*.

There are two forms of line addressing in the sed editor:

- A numeric range of lines
- A text pattern that filters out a line

Both forms use the same format for specifying the address:

```
[address]command
```

You can also group more than one command together for a specific address:

```
address {
  command1
  command2
  command3
}
```

The sed editor applies each of the commands you specify only to lines that match the address specified.

This section demonstrates using both of these addressing techniques in your sed editor scripts.

Numeric Line Addressing

When using numeric line addressing, you reference lines using their line position in the text stream. The sededitor assigns the first line in the text stream as line number one and continues sequentially for each new line.

The address you specify in the command can be a single line number or a range of lines specified by a starting line number, a comma, and an ending line number. Here's an example of specifying a line number to which the sedcommand will be applied:

```
$ sed '2s/dog/cat/' data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
$
```

The sed editor modified the text only in line two per the address specified. Here's another example, this time using a range of line addresses:

```
$ sed '2,3s/dog/cat/' data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy dog
$
```

If you want to apply a command to a group of lines starting at some point within the text, but continuing to the end of the text, you can use the special address, the dollar sign:

```
$ sed '2,$s/dog/cat/' data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
The quick brown fox jumps over the lazy cat
$
```

Because you may not know how many lines of data are in the text, the dollar sign often comes in handy.

Using Text Pattern Filters

The other method of restricting which lines a command applies to is a bit more complicated. The sed editor allows you to specify a text pattern that it uses to filter lines for the command. The format for this is:

```
/pattern/command
```

You must encapsulate the *pattern* you specify in forward slashes. The sed editor applies the command only to lines that contain the text pattern that you specify.

For example, if you want to change the default shell for only the user Samantha, you'd use the sed command:

```
$
$ grep Samantha /etc/passwd
Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/bash
```

```
$
$ sed '/Samantha/s/bash/csh/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/csh
Timothy:x:1002:1005::/home/Timothy:/bin/sh
Christine:x:1003:1006::/home/Christine:/bin/sh
$
```

The command was only applied to the line with the matching text pattern. While using a fixed text pattern may be useful for filtering specific values, as in the `userid` example, it's somewhat limited in what you can do with it. The `sed` editor uses a feature called *regular expressions* in text patterns to allow you to create patterns that get pretty involved.

Regular expressions allow you to create advanced text pattern–matching formulas to match all sorts of data. These formulas combine a series of wildcard characters, special characters, and fixed text characters to produce a concise pattern that can match just about any text situation. Regular expressions are one of the scarier parts of shell script programming and Chapter 19 covers them in great detail.

Grouping Commands

If you need to perform more than one command on an individual line, group the commands together using braces. The `sed` editor will process each command listed on the address line(s):

```
$ sed '2{
> s/fox/elephant/
> s/dog/cat/
> }' data1
The quick brown fox jumps over the lazy dog.
The quick brown elephant jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

Both commands are processed against the address. And of course, you can also specify an address range before the grouped commands:

```
$ sed '3,${
> s/brown/green/
> s/lazy/active/
> }' data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick green fox jumps over the active dog.
The quick green fox jumps over the active dog.
$
```

The `sed` editor applies all of the commands to all of the lines in the address range.

Deleting Lines

The text substitution command isn't the only command available in the sed editor. If you need to delete specific lines of text in a text stream, there's the delete command.

The delete command, d, pretty much does what it says. It will delete any text lines that match the addressing scheme supplied. Be careful with the delete command, because if you forget to include an addressing scheme, all of the lines will be deleted from the stream:

```
$ cat data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
$ sed ‘d’ data1
$
```

The delete command is obviously most useful when used in conjunction with a specified address. This allows you to delete specific lines of text from the data stream, either by line number:

```
$ sed ‘3d’ data7
This is line number 1.
This is line number 2.
This is line number 4.
$
```

or by a specific range of lines:

```
$ sed ‘2,3d’ data7
This is line number 1.
This is line number 4.
$
```

or by using the special end-of-file character:

```
$ sed ‘3,$d’ data7
This is line number 1.
This is line number 2.
$
```

The pattern-matching feature of the sed editor also applies to the delete command:

```
$ sed ‘/number 1/d’ data7
This is line number 2.
This is line number 3.
This is line number 4.
$
```

The sed editor removes the line containing text that matches the pattern you specify.

Note

Remember that the sed editor doesn't touch the original file. Any lines you delete are only gone from the output of the sed editor. The original file still contains the “deleted” lines.

You can also delete a range of lines using two text patterns, but be careful if you do this. The first pattern you specify "turns on" the line deletion, and the second pattern "turns off" the line deletion. The sed editor deletes any lines between the two specified lines (including the specified lines):

```
$ sed '/1/,/3/d' data6
This is line number 4.
$
```

In addition, you need to be careful, as the delete feature will "turn on" whenever the sed editor detects the start pattern in the data stream. This may produce an unexpected result:

```
$ cat data8
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is line number 1 again.
This is text you want to keep.
This is the last line in the file.
$
$ sed '/1/,/3/d' data7
This is line number 4.
$
```

The second occurrence of a line with the number 1 in it triggered the delete command again, deleting the rest of the lines in the data stream, as the stop pattern wasn't recognized. Of course, the other obvious problem occurs if you specify a stop pattern that never appears in the text:

```
$ sed '/1/,/5/d' data8
$
```

Because the delete features "turned on" at the first pattern match, but never found the end pattern match, the entire data stream was deleted.

Inserting and Appending Text

As you would expect, like any other editor, the sed editor allows you to insert and append text lines to the data stream. The difference between the two actions can be confusing:

- The insert command (i) adds a new line before the specified line.
- The append command (a) adds a new line after the specified line.

What's confusing about these two commands is their formats. You can't use these commands on a single command line. You must specify the line to insert or append on a separate line by itself. The format for doing this is:

```
sed '[address]command\
new line'
```

The text in *new line* appears in the sed editor output in the place you specify. Remember that when you use the insert command, the text appears before the data stream text:

```
$ echo “Test Line 2” | sed ‘i\Test Line 1’
Test Line 1
Test Line 2
$
```

And when you use the append command, the text appears after the data stream text:

```
$ echo “Test Line 2” | sed ‘a\Test Line 1’
Test Line 2
Test Line 1
$
```

When you use the sed editor from the command line interface prompt, you'll get the secondary prompt to enter the new line data. You must complete the sed editor command on this line. Once you enter the ending single quotation mark, the bash shell will process the command:

```
$ echo “Test Line 2” | sed ‘i\
> Test Line 1’
Test Line 1
Test Line 2
$
```

This works well for adding text before or after the text in the data stream, but what about adding text inside the data stream?

To insert or append data inside the data stream lines, you must use addressing to tell the sed editor where you want the data to appear. You can specify only a single line address when using these commands. You can match either a numeric line number or a text pattern, but you cannot use a range of addresses. This is logical, because you can only insert or append before or after a single line, and not a range of lines.

The following is an example of inserting a new line before line 3 in the data stream:

```
$ sed ‘3i\
> This is an inserted line.’ data7
This is line number 1.
This is line number 2.
This is an inserted line.
This is line number 3.
This is line number 4.
$
```

The following is an example of appending a new line after line 3 in the data stream:

```
$ sed ‘3a\
>This is an appended line.’ data7
This is line number 1.
This is line number 2.
This is line number 3.
This is an appended line.
This is line number 4.
```

```
$
```

This uses the same process as the `insert` command; it just places the new text line after the specified line number. If you have a multiline data stream, and you want to append a new line of text to the end of a data stream, just use the dollar sign, which represents the last line of data:

```
$ sed '$a\
> This is a new line of text.' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is a new line of text.
$
```

The same idea applies if you want to add a new line at the beginning of the data stream. Just insert a new line before line number one.

To insert or append more than one line of text, you must use a backslash on each line of new text until you reach the last text line where you want to insert or append text:

```
$ sed '1i\
> This is one line of new text.\
> This is another line of new text.' data7
This is one line of new text.
This is another line of new text.
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
```

Both of the specified lines are added to the data stream.

Changing Lines

The `change` command allows you to change the contents of an entire line of text in the data stream. It works the same way as the `insert` and `append` commands, in that you must specify the new line separately from the rest of the `sed` command:

```
$ sed '3c\
> This is a changed line of text.' data7
This is line number 1.
This is line number 2.
This is a changed line of text.
This is line number 4.
$
```

In this example, the `sed` editor changes the text in line number 3. You can also use a text pattern for the address:

```
$ sed '/number 3/c\
> This is a changed line of text.' data7
```

```
This is line number 1.
This is line number 2.
This is a changed line of text.
This is line number 4.
$
```

The text pattern change command will change any line of text in the data stream that it matches.

```
$ sed '/number 1/c\
> This is a changed line of text.' data8
This is a changed line of text.
This is line number 2.
This is line number 3.
This is line number 4.
This is a changed line of text.
This is yet another line.
This is the last line in the file.
$
```

You can use an address range in the change command, but the results may not be what you expect:

```
$ sed '2,3c\
> This is a new line of text.' data7
This is line number 1.
This is a new line of text.
This is line number 4.
$
```

Instead of changing both lines with the text, the sed editor uses the single line of text to replace both lines.

The transform Command

The transform command (y) is the only sed editor command that operates on a single character. The transform command uses the format:

```
[address]y/inchars/outchars/
```

The transform command performs a one-to-one mapping of the inchars and the outchars values. The first character in inchars is converted to the first character in outchars. The second character in inchars is converted to the second character in outchars. This mapping continues throughout the length of the specified characters. If theinchars and outchars are not the same length, the sed editor will produce an error message.

A simple example of using the transform command is:

```
$ sed 'y/123/789/' data8
This is line number 7.
This is line number 8.
This is line number 9.
This is line number 4.
```

```
This is line number 7 again.
This is yet another line.
This is the last line in the file.
$
```

As you can see from the output, each instance of the characters specified in the `inchars` pattern has been replaced by the character in the same position in the `outchars` pattern.

The `transform` command is a global command; that is, it performs the transformation on any character found in the text line automatically, without regard to the occurrence:

```
$ echo “This 1 is a test of 1 try.” | sed ‘y/123/456/’
This 4 is a test of 4 try.
$
```

The `sed` editor transformed both instances of the matching character 1 in the text line. You can't limit the transformation to a specific occurrence of the character.

Printing Revisited

The “More Substitution Options” section showed you how to use the `p` flag with the substitution command to display lines that the `sed` editor changed. There are three commands that also can be used to print information from the data stream:

- The lowercase `p` command to print a text line
- The equal sign (=) command to print line numbers
- The `l` (lowercase L) command to list a line

The following sections look at each of these three printing commands in the `sed` editor.

Printing Lines

Like the `p` flag in the `substitution` command, the `p` command prints a line in the `sed` editor output. On its own, there's not much excitement:

```
$ echo “this is a test” | sed ‘p’
this is a test
this is a test
$
```

All it does is print the data text that you already know is there. The most common use for the `print` command is printing lines that contain matching text from a text pattern:

```
$ sed -n ‘/number 3/p’ data7
This is line number 3.
$
```

By using the `-n` option on the command line, you can suppress all of the other lines and only print the line that contains the matching text pattern.

You can also use this as a quick way to print a subset of lines in a data stream:

```
$ sed -n ‘2,3p’ data7
```

```
This is line number 2.
This is line number 3.
$
```

You can also use the `print` command when you need to see a line before it gets altered, such as with the `substitution` or `change` command. You can create a script that displays the line before it's changed:

```
$ sed -n '/3/{
p
s/line/test/p
}' data7
This is line number 3.
This is test number 3.
$
```

This `sed` editor command searches for lines that contain the number 3, and then executes two commands. First, the script uses the `p` command to print the original version of the line; then it uses the `s` command to substitute text, along with the `p` flag to print the resulting text. The output shows both the original line text and the new line text.

Printing Line Numbers

The `equal sign` command prints the current line number for the line within the data stream. Line numbers are determined by using the newline character in the data stream. Each time a newline character appears in the data stream, the `sed` editor assumes that it terminates a line of text:

```
$ sed '=' data1
1
The quick brown fox jumps over the lazy dog.
2
The quick brown fox jumps over the lazy dog.
3
The quick brown fox jumps over the lazy dog.
4
The quick brown fox jumps over the lazy dog.
$
```

The `sed` editor prints the line number before the actual line of text. The `equal sign` command comes in handy if you're searching for a specific text pattern in the data stream:

```
$ sed -n '/number 4/{
=
p
}' data7
4
This is line number 4.
$
```

By using the -n option you can have the sed editor display only both the line number and text for the line that contains the matching text pattern.

Listing Lines

The list command (l) allows you to print both the text and nonprintable ASCII characters in a data stream. Any nonprintable characters are shown using either their octal values, preceded by a backslash, or the standard C-style nomenclature for common nonprintable characters, such as \t for tab characters:

```
$ cat data9
This    line    contains        tabs.
$
$ sed -n 'l' data9
This\tline\tcontains\ttabs.$
$
```

The tab character locations are shown with the \t nomenclature. The dollar sign at the end of the line indicates the newline character. If you have a data stream that contains an escape character, the list command displays it using the octal code:

```
$ cat data10
This line contains an escape character
$
$ sed -n 'l' data10
This line contains an escape character \033[44m$
$
```

The data10 text file contains an escape control code (see Chapter 17) to change the display color. When you use the cat command to display the text file, you don't see the escape control code; it just changes the color of the display.

However, using the list command, you can display the escape control code used. The \033 is the octal value of the ASCII code for the escape key.

Using Files with sed

The substitution command contains flags that allow you to work with files. There are also regular sed editor commands that let you do that without having to substitute text.

Writing to a File

The w command is used to write lines to a file. The format for the w command is:

```
[address]w filename
```

The *filename* can be specified as either a relative or absolute pathname, but in either case, the person running the sed editor must have write permissions for the file. The address can be any type of addressing method used insed, such as a single line number, a text pattern, or a range of line numbers or text patterns.

The following is an example that prints only the first two lines of a data stream to a text file:

```
$ sed '1,2w test' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
$ cat test
This is line number 1.
This is line number 2.
$
```

Of course, if you don't want the lines to display on STDOUT, you can use the `-n` option for the `sed` command.

This is a great tool to use if you need to create a data file from a master file on the basis of common text values, such as those in a mailing list:

```
$ cat data11
Blum, Katie      Chicago, IL
Mullen, Riley   West Lafayette, IN
Snell, Haley    Ft. Wayne, IN
Woenker, Matthew        Springfield, IL
Wisecarver, Emma        Grant Park, IL
$
$ sed -n '/IN/w INcustomers' data11
$
$ cat INcustomers
Mullen, Riley   West Lafayette, IN
Snell, Haley    Ft. Wayne, IN
$
```

The `sed` editor writes to a destination file only the data lines that contain the text pattern.

Reading Data from a File

You've already seen how to insert data into and append text to a data stream from the `sed` command line. The `read`command (`r`) allows you to insert data contained in a separate file.

The format of the `read` command is:

```
[address]r filename
```

The `filename` parameter specifies either an absolute or relative pathname for the file that contains the data. You can't use a range of addresses for the `read` command. You can only specify a single line number or text pattern address. The `sed` editor inserts the text from the file after the address.

```
$ cat data12
This is an added line.
This is the second added line.
$
```

```
$ sed '3r data12' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is an added line.
This is the second added line.
This is line number 4.
$
```

The sed editor inserts into the data stream all of the text lines in the data file. The same technique works when using a text pattern address:

```
$ sed '/number 2/r data12' data7
This is line number 1.
This is line number 2.
This is an added line.
This is the second added line.
This is line number 3.
This is line number 4.
$
```

If you want to add text to the end of a data stream, just use the dollar sign address symbol:

```
$ sed '$r data12' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is an added line.
This is the second added line.
$
```

A cool application of the read command is to use it in conjunction with a delete command to replace a placeholder in a file with data from another file. For example, suppose that you had a form letter stored in a text file that looked like this:

```
$ cat letter
Would the following people:
LIST
please report to the office.
$
```

The form letter uses the generic placeholder LIST in place of a list of people. To insert the list of people after the placeholder, all you need to do is use the read command. However, this still leaves the placeholder text in the output. To remove that, just use the delete command. The result looks like this:

```
$ sed '/LIST/{
> r data11
> d
> }' letter
Would the following people:
```

```
Blum, Katie      Chicago, IL
Mullen, Riley   West Lafayette, IN
Snell, Haley    Ft. Wayne, IN
Woenker, Matthew        Springfield, IL
Wisecarver, Emma        Grant Park, IL
please report to the office.
$
```

Now the placeholder text is replaced with the list of names from the data file.

Summary

While shell scripts can do a lot of work on their own, it's often difficult to manipulate data with just a shell script. Linux provides two handy utilities to help out with handling text data. The sed editor is a stream editor that quickly processes data on the fly as it reads it. You must provide the sed editor with a list of editing commands, which it applies to the data.

The gawk program is a utility from the GNU organization that mimics and expands on the functionality of the Unixawk program. The gawk program contains a built-in programming language that you can use to write scripts to handle and process data. You can use the gawk program to extract data elements from large data files and output them in just about any format you desire. This makes processing large log files a snap, as well as creating custom reports from data files.

A crucial element of using both the sed and gawk programs is knowing how to use regular expressions. Regular expressions are key to creating customized filters for extracting and manipulating data in text files. The next chapter dives into the often misunderstood world of regular expressions, showing you how to build regular expressions for manipulating all types of data.

Chapter 19

Regular Expressions

In This Chapter

- Defining regular expressions
- Looking at the basics
- Extending our patterns
- Creating expressions

The key to successfully working with the sed editor and the gawk program in your shell script is your comfort using regular expressions. This is not always an easy thing to do, as trying to filter specific data from a large batch of data can (and often does) get complicated. This chapter describes how to create regular expressions in both the sed editor and the gawk program that can filter out just the data you need.

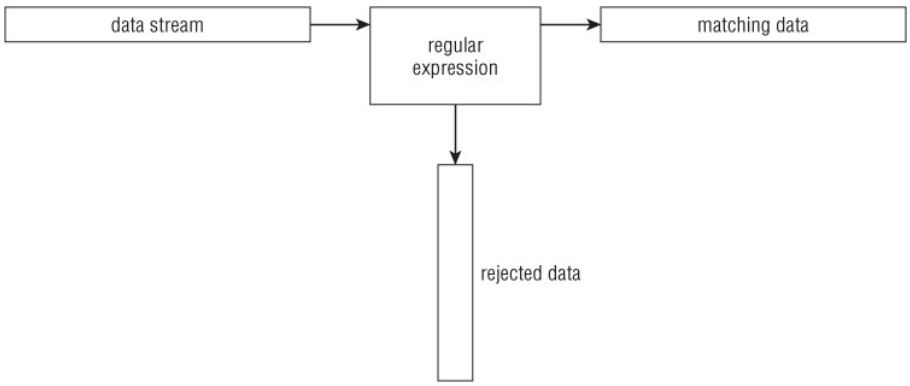
What Are Regular Expressions?

The first step to understanding regular expressions is to define just exactly what they are. This section explains what a regular expression is and describes how Linux uses regular expressions.

A Definition

A regular expression is a pattern template you define that a Linux utility uses to filter text. A Linux utility (such as the sed editor or the gawk program) matches the regular expression pattern against data as that data flows into the utility. If the data matches the pattern, it's accepted for processing. If the data doesn't match the pattern, it's rejected. This is illustrated in Figure 19.1.

Figure 19.1 Matching data against a regular expression pattern



The regular expression pattern makes use of wildcard characters to represent one or more characters in the data stream. There are plenty of instances in Linux where you can specify a wildcard character to represent data that you don't know about. You've already seen an example of using wildcard characters with the Linux `ls` command for listing files and directories (see Chapter 3).

The asterisk wildcard character allows you to list only files that match a certain criteria. For example:

```
$ ls -al da*
-rw-r--r--    1 rich     rich           45 Nov 26 12:42 data
-rw-r--r--    1 rich     rich           25 Dec  4 12:40 data.tst
-rw-r--r--    1 rich     rich          180 Nov 26 12:42 data1
-rw-r--r--    1 rich     rich           45 Nov 26 12:44 data2
-rw-r--r--    1 rich     rich           73 Nov 27 12:31 data3
-rw-r--r--    1 rich     rich           79 Nov 28 14:01 data4
-rw-r--r--    1 rich     rich          187 Dec  4 09:45 datatest
$
```

The `da*` parameter instructs the `ls` command to list only the files whose name starts with *da*. There can be any number of characters after the `da` in the filename (including none). The `ls` command reads the information regarding all of the files in the directory but displays only the ones that match the wildcard character.

Regular expression wildcard patterns work in a similar way. The regular expression pattern contains text and/or special characters that define a template for the sed editor and the gawk program to follow when matching data. There are different special characters you can use in a regular expression to define a specific pattern for filtering data.

Types of Regular Expressions

The biggest problem with using regular expressions is that there isn't just one set of them. Several different applications use different types of regular expressions in the Linux environment. These include such diverse applications as programming languages (Java, Perl, and Python), Linux utilities (such as the sed editor, the gawk program, and the grep utility), and mainstream applications (such as the MySQL and PostgreSQL database servers).

A regular expression is implemented using a *regular expression engine*. A regular expression engine is the underlying software that interprets regular expression patterns and uses those patterns to match text.

In the Linux world, there are two popular regular expression engines:

- The POSIX Basic Regular Expression (BRE) engine
- The POSIX Extended Regular Expression (ERE) engine

Most Linux utilities at a minimum conform to the POSIX BRE engine specifications, recognizing all of the pattern symbols it defines. Unfortunately, some utilities (such as the sed editor) only conform to a subset of the BRE engine specifications. This is due to speed constraints, as the sed editor attempts to process text in the data stream as quickly as possible.

The POSIX ERE engine is often found in programming languages that rely on regular expressions for text filtering. It provides advanced pattern symbols as well as special symbols for common patterns, such as matching digits, words, and alphanumeric characters. The gawk program uses the ERE engine to process its regular expression patterns.

Because there are so many different ways to implement regular expressions, it's hard to present a single, concise description of all the possible regular expressions. The following sections discuss the most commonly found regular expressions and demonstrate how to use them in the sed editor and gawk program.

Defining BRE Patterns

The most basic BRE pattern is matching text characters in a data stream. This section demonstrates how you can define text in the regular expression pattern and what to expect from the results.

Plain Text

Chapter 18 demonstrated how to use standard text strings in the sed editor and the gawk program to filter data. Here's an example to refresh your memory:

```
$ echo “This is a test” | sed -n ‘/test/p’
This is a test
$ echo “This is a test” | sed -n ‘/trial/p’
$
$ echo “This is a test” | gawk ‘/test/{print $0}’
This is a test
$ echo “This is a test” | gawk ‘/trial/{print $0}’
$
```

The first pattern defines a single word, *test*. The sed editor and gawk program scripts each use their own version of the `print` command to print any lines that match the regular expression pattern. Because the `echo` statement contains the word “test” in the text string, the data stream text matches the defined regular expression pattern, and the sed editor displays the line.

The second pattern again defines just a single word, this time the word “trial.” Because the `echo` statement text string doesn't contain that word, the regular expression pattern doesn't match, so neither the sed editor nor the gawk program prints the line.

You probably already noticed that the regular expression doesn't care where in the data stream the pattern occurs. It also doesn't matter how many times the pattern occurs. Once the regular expression can match the pattern anywhere in the text string, it passes the string along to the Linux utility that's using it.

The key is matching the regular expression pattern to the data stream text. It's important to remember that regular expressions are extremely picky about matching patterns. The first rule to remember is that regular expression patterns are case sensitive. This means they'll only match patterns with the proper case of characters:

```
$ echo “This is a test” | sed -n ‘/this/p’
$
$ echo “This is a test” | sed -n ‘/This/p’
This is a test
$
```

The first attempt failed to match because the word “this” doesn't appear in all lowercase in the text string, while the second attempt, which uses the uppercase letter in the pattern, worked just fine.

You don't have to limit yourself to whole words in the regular expression. If the defined text appears anywhere in the data stream, the regular expression will match the following:

```
$ echo “The books are expensive” | sed -n ‘/book/p’
The books are expensive
$
```

Even though the text in the data stream is `books`, the data in the stream contains the regular expression book, so the regular expression pattern matches the data. Of course, if you try the opposite, the regular expression will fail:

```
$ echo “The book is expensive” | sed -n ‘/books/p’
```

```
$
```

The complete regular expression text didn't appear in the data stream, so the match failed and the sed editor didn't display the text.

You also don't have to limit yourself to single text words in the regular expression. You can include spaces and numbers in your text string as well:

```
$ echo “This is line number 1” | sed -n ‘/ber 1/p’
This is line number 1
$
```

Spaces are treated just like any other character in the regular expression:

```
$ echo “This is line number1” | sed -n ‘/ber 1/p’
$
```

If you define a space in the regular expression, it must appear in the data stream. You can even create a regular expression pattern that matches multiple contiguous spaces:

```
$ cat data1
This is a normal line of text.
This is  a line with too many spaces.
$ sed -n ‘/  /p’ data1
This is  a line with too many spaces.
$
```

The line with two spaces between words matches the regular expression pattern. This is a great way to catch spacing problems in text files!

Special Characters

As you use text strings in your regular expression patterns, there's something you need to be aware of. There are a few exceptions when defining text characters in a regular expression. Regular expression patterns assign a special meaning to a few characters. If you try to use these characters in your text pattern, you won't get the results you were expecting.

The special characters recognized by regular expressions are:

```
.*[]^${}\+?|()
```

As the chapter progresses, you'll find out just what these special characters do in a regular expression. For now, however, just remember that you can't use these characters by themselves in your text pattern.

If you want to use one of the special characters as a text character, you need to *escape* it. When you escape the special characters, you add a special character in front of it to indicate to the regular expression engine that it should interpret the next character as a normal text character. The special character that does this is the backslash character (\).

For example, if you want to search for a dollar sign in your text, just precede it with a backslash character:

```
$ cat data2
The cost is $4.00
$ sed -n ‘/\$/p’ data2
```

```
The cost is $4.00
$
```

Because the backslash is a special character, if you need to use it in a regular expression pattern you'll need to escape it as well, producing a double backslash:

```
$ echo “\ is a special character” | sed -n ‘/\\/p’
\ is a special character
$
```

Finally, although the forward slash isn't a regular expression special character, if you use it in your regular expression pattern in the sed editor or the gawk program, you'll get an error:

```
$ echo “3 / 2” | sed -n ‘///p’
sed: -e expression #1, char 2: No previous regular expression
$
```

To use a forward slash you'll need to escape that as well:

```
$ echo “3 / 2” | sed -n ‘/\//p’
3 / 2
$
```

Now the sed editor can properly interpret the regular expression pattern, and all is well.

Anchor Characters

As shown in the “Plain Text” section, by default, when you specify a regular expression pattern, if the pattern appears anywhere in the data stream, it will match. There are two special characters you can use to anchor a pattern to either the beginning or the end of lines in the data stream.

Starting at the Beginning

The caret character (^) defines a pattern that starts at the beginning of a line of text in the data stream. If the pattern is located any place other than the start of the line of text, the regular expression pattern fails.

To use the caret character, you must place it before the pattern specified in the regular expression:

```
$ echo “The book store” | sed -n ‘/^book/p’
$
$ echo “Books are great” | sed -n ‘/^Book/p’
Books are great
$
```

The caret anchor character checks for the pattern at the beginning of each new line of data, as determined by the newline character:

```
$ cat data3
This is a test line.
this is another test line.
A line that tests this feature.
```

```
Yet more testing of this
$ sed -n '/^this/p' data3
this is another test line.
$
```

As long as the pattern appears at the start of a new line, the caret anchor will catch it.

If you position the caret character in any place other than at the beginning of the pattern, it will act like a normal character and not as a special character:

```
$ echo "This ^ is a test" | sed -n '/s ^/p'
This ^ is a test
$
```

Because the caret character is listed last in the regular expression pattern, the sed editor uses it as a normal character to match text.

Note

If you need to specify a regular expression pattern using only the caret character, you don't need to escape it with a backslash. However, if you specify the caret character first, followed by additional text in the pattern, you'll need to use the escape character before the caret character.

Looking for the Ending

The opposite of looking for a pattern at the start of a line is looking for it at the end of a line. The dollar sign ($) special character defines the end anchor. Add this special character after a text pattern to indicate that the line of data must end with the text pattern:

```
$ echo "This is a good book" | sed -n '/book$/p'
This is a good book
$ echo "This book is good" | sed -n '/book$/p'
$
```

The problem with an ending text pattern is that you must be careful what you're looking for:

```
$ echo "There are a lot of good books" | sed -n '/book$/p'
$
```

Making the word "book" plural at the end of the line means that it no longer matches the regular expression pattern, even though book is in the data stream. The text pattern must be the last thing on the line for the pattern to match.

Combining Anchors

There are a couple of common situations where you can combine both the start and end anchor on the same line. In the first situation, suppose that you want to look for a line of data containing only a specific text pattern:

```
$ cat data4
this is a test of using both anchors
I said this is a test
this is a test
```

```
I'm sure this is a test.
$ sed -n '/^this is a test$/p' data4
this is a test
$
```

The sed editor ignores the lines that include other text besides the specified text.

The second situation may seem a little odd at first but is extremely useful. By combining both anchors in a pattern with no text, you can filter blank lines from the data stream. Consider this example:

```
$ cat data5
This is one test line.


This is another test line.
$ sed '/^$/d' data5
This is one test line.
This is another test line.
$
```

The regular expression pattern that is defined looks for lines that have nothing between the start and end of the line. Because blank lines contain no text between the two newline characters, they match the regular expression pattern. The sed editor uses the d delete command to delete lines that match the regular expression pattern, thus removing all blank lines from the text. This is an effective way to remove blank lines from documents.

The Dot Character

The dot special character is used to match any single character except a newline character. The dot character must match a character, however; if there's no character in the place of the dot, then the pattern will fail.

Let's take a look at a few examples of using the dot character in a regular expression pattern:

```
$ cat data6
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.
$ sed -n '/.at/p' data6
The cat is sleeping.
That is a very nice hat.
This test is at line four.
$
```

You should be able to figure out why the first line failed and why the second and third lines passed. The fourth line is a little tricky. Notice that we matched the `at`, but there's no character in front of it to match the dot character. Ah, but there is! In regular expressions, spaces count as characters, so the space in front of the `at`matches the

pattern. The fifth line proves this, by putting the `at` in the front of the line, which fails to match the pattern.

Character Classes

The dot special character is great for matching a character position against any character, but what if you want to limit what characters to match? This is called a *character class* in regular expressions.

You can define a class of characters that would match a position in a text pattern. If one of the characters from the character class is in the data stream, it matches the pattern.

To define a character class, you use square brackets. The brackets should contain any character that you want to include in the class. You then use the entire class within a pattern just like any other wildcard character. This takes a little getting used to at first, but once you catch on it can generate some pretty amazing results.

The following is an example of creating a character class:

```
$ sed -n '/[ch]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

Using the same data file as in the dot special character example, we came up with a different result. This time we managed to filter out the line that just contained the word `at`. The only words that match this pattern are `cat`and `hat`. Also notice that the line that started with `at` didn't match as well. There must be a character in the character class that matches the appropriate position.

Character classes come in handy if you're not sure which case a character is in:

```
$ echo "Yes" | sed -n '/[Yy]es/p'
Yes
$ echo "yes" | sed -n '/[Yy]es/p'
yes
$
```

You can use more than one character class in a single expression:

```
$ echo "Yes" | sed -n '/[Yy][Ee][Ss]/p'
Yes
$ echo "yEs" | sed -n '/[Yy][Ee][Ss]/p'
yEs
$ echo "yeS" | sed -n '/[Yy][Ee][Ss]/p'
yeS
$
```

The regular expression used three character classes to cover both lower and upper cases for all three character positions.

Character classes don't have to contain just letters; you can use numbers in them as well:

```
$ cat data7
```

```
This line doesn't contain a number.
This line has 1 number on it.
This line a number 2 on it.
This line has a number 4 on it.
$ sed -n '/[0123]/p' data7
This line has 1 number on it.
This line a number 2 on it.
$
```

The regular expression pattern matches any lines that contain the numbers 0, 1, 2, or 3. Any other numbers are ignored, as are lines without numbers in them.

You can combine character classes to check for properly formatted numbers, such as phone numbers and zip codes. However, when you're trying to match a specific format, you must be careful. Here's an example of a zip code match gone wrong:

```
$ cat data8
60633
46201
223001
4353
22203
$ sed -n '
>/[0123456789][0123456789][0123456789][0123456789][0123456789]/p
>' data8
60633
46201
223001
22203
$
```

This might not have produced the result you were thinking of. It did a fine job of filtering out the number that was too short to be a zip code, as the last character class didn't have a character to match against. However, it still passed the six-digit number, even though we only defined five character classes.

Remember that the regular expression pattern can be found anywhere in the text of the data stream. There can always be additional characters besides the matching pattern characters. If you want to ensure that you only match against five numbers, you need to delineate them somehow, either with spaces, or as in this example, by showing that they're at the start and end of the line:

```
$ sed -n '
> /^[0123456789][0123456789][0123456789][0123456789][0123456789]$/p
> ' data8
60633
46201
22203
$
```

Now that's much better! Later in this chapter we look at how to simplify this even further.

One extremely popular use for character classes is parsing words that might be misspelled, such as data entered from a user form. You can easily create regular expressions that can accept common misspellings in data:

```
$ cat data9
I need to have some maintenence done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$ sed -n ‘
/maint[ea]n[ae]nce/p
/sep[ea]r[ea]te/p
’data9
I need to have some maintenence done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$
```

The two sed print commands in this example utilize regular expression character classes to help catch the misspelled words, maintenance and separate, in the text. The same regular expression pattern also matches the properly spelled occurrence of “maintenance.”

Negating Character Classes

In regular expression patterns, you can also reverse the effect of a character class. Instead of looking for a character contained in the class, you can look for any character that's not in the class. To do that, just place a caret character at the beginning of the character class range:

```
$ sed -n ‘/[^ch]at/p’ data6
This test is at line two.
$
```

By negating the character class, the regular expression pattern matches any character that's neither a *c* nor an*h*, along with the text pattern. Because the space character fits this category, it passed the pattern match. However, even with the negation, the character class must still match a character, so the line with the `at` in the start of the line still doesn't match the pattern.

Using Ranges

You may have noticed when I showed the zip code example earlier that it was somewhat awkward having to list all of the possible digits in each character class. Fortunately, you can use a shortcut so you don't have to do that.

You can use a range of characters within a character class by using the dash symbol. Just specify the first character in the range, a dash, and then the last character in the range. The regular expression includes any character that's within the specified character range, according to the character set used by the Linux system (see Chapter 2).

Now you can simplify the zip code example by specifying a range of digits:

```
$ sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p' data8
60633
46201
45902
$
```

That saved a lot of typing! Each character class will match any digit from 0 to 9. The pattern will fail if a letter is present anywhere in the data:

```
$ echo "a8392" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
$ echo "1839a" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
$ echo "18a92" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
```

The same technique also works with letters:

```
$ sed -n '/[c-h]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

The new pattern `[c-h]at` matches words where the first letter is between the letter *c* and the letter *h*. In this case, the line with only the word `at` failed to match the pattern.

You can also specify multiple, noncontinuous ranges in a single character class:

```
$ sed -n '/[a-ch-m]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

The character class allows the ranges *a* through *c*, and *h* through *m* to appear before the `at` text. This range would reject any letters between *d* and *g*:

```
$ echo "I'm getting too fat." | sed -n '/[a-ch-m]at/p'
$
```

This pattern rejected the `fat` text, as it wasn't in the specified range.

Special Character Classes

In addition to defining your own character classes, the BRE contains special character classes you can use to match against specific types of characters. Table 19.1 describes the BRE special characters you can use.

Table 19.1 BRE Special Character Classes

Class	Description
`[[:alpha:]]`	Match any alphabetical character, either upper or lower case.
`[[:alnum:]]`	Match any alphanumeric character 0–9, A–Z, or a–z.
`[[:blank:]]`	Match a space or Tab character.

`[[:digit:]]`	Match a numerical digit from 0 through 9.
`[[:lower:]]`	Match any lowercase alphabetical character a–z.
`[[:print:]]`	Match any printable character.
`[[:punct:]]`	Match a punctuation character.
`[[:space:]]`	Match any whitespace character: space, Tab, NL, FF, VT, CR.
`[[:upper:]]`	Match any uppercase alphabetical character A–Z.

You use the special character classes just as you would a normal character class in your regular expression patterns:

```
$ echo “abc” | sed -n ‘/[[:digit:]]/p’
$
$ echo “abc” | sed -n ‘/[[:alpha:]]/p’
abc
$ echo “abc123” | sed -n ‘/[[:digit:]]/p’
abc123
$ echo “This is, a test” | sed -n ‘/[[:punct:]]/p’
This is, a test
$ echo “This is a test” | sed -n ‘/[[:punct:]]/p’
$
```

Using the special character classes is an easy way to define ranges. Instead of having to use a range [0–9], you can just use `[[:digit:]]`.

The Asterisk

Placing an asterisk after a character signifies that the character must appear zero or more times in the text to match the pattern:

```
$ echo “ik” | sed -n ‘/ie*k/p’
ik
$ echo “iek” | sed -n ‘/ie*k/p’
iek
$ echo “ieek” | sed -n ‘/ie*k/p’
ieek
$ echo “ieeek” | sed -n ‘/ie*k/p’
ieeek
$ echo “ieeeek” | sed -n ‘/ie*k/p’
ieeeek
$
```

This pattern symbol is commonly used for handling words that have a common misspelling or variations in language spellings. For example, if you need to write a script that may be used in either American or British English, you could write:

```
$ echo “I'm getting a color TV” | sed -n ‘/colou*r/p’
I'm getting a color TV
$ echo “I'm getting a colour TV” | sed -n ‘/colou*r/p’
I'm getting a colour TV
$
```

The u* in the pattern indicates that the letter *u* may or may not appear in the text to match the pattern. Similarly, if you know of a word that is commonly misspelled, you can accommodate it by using the asterisk:

```
$ echo “I ate a potatoe with my lunch.” | sed -n ‘/potatoe*/p’
I ate a potatoe with my lunch.
$ echo “I ate a potato with my lunch.” | sed -n ‘/potatoe*/p’
I ate a potato with my lunch.
$
```

Placing an asterisk next to the possible extra letter allows you to accept the misspelled word.

Another handy feature is combining the dot special character with the asterisk special character.This combination provides a pattern to match any number of any characters. It's often used between two text strings that may or may not appear next to each other in the data stream:

```
$ echo “this is a regular pattern expression” | sed -n ‘
> /regular.*expression/p’
this is a regular pattern expression
$
```

Using this pattern, you can easily search for multiple words that may appear anywhere in a line of text in the data stream.

The asterisk can also be applied to a character class. This allows you to specify a group or range of characters that can appear more than once in the text:

```
$ echo “bt” | sed -n ‘/b[ae]*t/p’
bt
$ echo “bat” | sed -n ‘/b[ae]*t/p’
bat
$ echo “bet” | sed -n ‘/b[ae]*t/p’
bet
$ echo “btt” | sed -n ‘/b[ae]*t/p’
btt
$
$ echo “baat” | sed -n ‘/b[ae]*t/p’
baat
$ echo “baaeeet” | sed -n ‘/b[ae]*t/p’
baaeeet
$ echo “baeeaeeat” | sed -n ‘/b[ae]*t/p’
baeeaeeat
$ echo “baakeeet” | sed -n ‘/b[ae]*t/p’
$
```

As long as the *a* and *e* characters appear in any combination between the *b* and *t* characters (including not appearing at all), the pattern matches. If any other character outside of the defined character class appears, the pattern match fails.

Extended Regular Expressions

The POSIX ERE patterns include a few additional symbols that are used by some Linux applications and utilities. The gawk program recognizes the ERE patterns, but the sed editor doesn't.

Caution

It's important to remember that there is a difference between the regular expression engines in the sed editor and the gawk program. The gawk program can use most of the extended regular expression pattern symbols, and it can provide some additional filtering capabilities that the sed editor doesn't have. However, because of this, it is often slower in processing data streams.

This section describes the more commonly found ERE pattern symbols that you can use in your gawk program scripts.

The Question Mark

The question mark is similar to the asterisk, but with a slight twist. The question mark indicates that the preceding character can appear zero or one time, but that's all. It doesn't match repeating occurrences of the character:

```
$ echo “bt” | gawk ‘/be?t/{print $0}’
bt
$ echo “bet” | gawk ‘/be?t/{print $0}’
bet
$ echo “beet” | gawk ‘/be?t/{print $0}’
$
$ echo “beeet” | gawk ‘/be?t/{print $0}’
$
```

If the *e* character doesn't appear in the text, or as long as it appears only once in the text, the pattern matches.

As with the asterisk, you can use the question mark symbol along with a character class:

```
$ echo “bt” | gawk ‘/b[ae]?t/{print $0}’
bt
$ echo “bat” | gawk ‘/b[ae]?t/{print $0}’
bat
$ echo “bot” | gawk ‘/b[ae]?t/{print $0}’
$
$ echo “bet” | gawk ‘/b[ae]?t/{print $0}’
bet
$ echo “baet” | gawk ‘/b[ae]?t/{print $0}’
$
$ echo “beat” | gawk ‘/b[ae]?t/{print $0}’
$
$ echo “beet” | gawk ‘/b[ae]?t/{print $0}’
$
```

If zero or one character from the character class appears, the pattern match passes. However, if both characters appear, or if one of the characters appears twice, the pattern match fails.

The Plus Sign

The plus sign is another pattern symbol that's similar to the asterisk, but with a different twist than the question mark. The plus sign indicates that the preceding character can appear one or more times, but must be present at least once. The pattern doesn't match if the character is not present:

```
$ echo “beeet” | gawk ‘/be+t/{print $0}’
beeet
$ echo “beet” | gawk ‘/be+t/{print $0}’
beet
$ echo “bet” | gawk ‘/be+t/{print $0}’
bet
$ echo “bt” | gawk ‘/be+t/{print $0}’
$
```

If the *e* character is not present, the pattern match fails. The plus sign also works with character classes, the same way as the asterisk and question mark do:

```
$ echo “bt” | gawk ‘/b[ae]+t/{print $0}’
$
$ echo “bat” | gawk ‘/b[ae]+t/{print $0}’
bat
$ echo “bet” | gawk ‘/b[ae]+t/{print $0}’
bet
$ echo “beat” | gawk ‘/b[ae]+t/{print $0}’
beat
$ echo “beet” | gawk ‘/b[ae]+t/{print $0}’
beet
$ echo “beeat” | gawk ‘/b[ae]+t/{print $0}’
beeat
$
```

This time if either character defined in the character class appears, the text matches the specified pattern.

Using Braces

Curly braces are available in ERE to allow you to specify a limit on a repeatable regular expression. This is often referred to as an *interval*. You can express the interval in two formats:

- `m` - The regular expression appears exactly *m* times.
- `m,n` - The regular expression appears at least *m* times, but no more than *n* times.

This feature allows you to fine-tune exactly how many times you allow a character (or character class) to appear in a pattern.

Caution

By default, the gawk program doesn't recognize regular expression intervals. You must specify the `--re-interval` command line option for the gawk program to recognize regular expression intervals.

Here's an example of using a simple interval of one value:

```
$ echo “bt” | gawk --re-interval ‘/be{1}t/{print $0}’
$
$ echo “bet” | gawk --re-interval ‘/be{1}t/{print $0}’
bet
$ echo “beet” | gawk --re-interval ‘/be{1}t/{print $0}’
$
```

By specifying an interval of one, you restrict the number of times the character can be present for the string to match the pattern. If the character appears more times, the pattern match fails.

There are lots of times when specifying the lower and upper limit comes in handy:

```
$ echo “bt” | gawk --re-interval ‘/be{1,2}t/{print $0}’
$
$ echo “bet” | gawk --re-interval ‘/be{1,2}t/{print $0}’
bet
$ echo “beet” | gawk --re-interval ‘/be{1,2}t/{print $0}’
beet
$ echo “beeet” | gawk --re-interval ‘/be{1,2}t/{print $0}’
$
```

In this example, the *e* character can appear once or twice for the pattern match to pass; otherwise, the pattern match fails.

The interval pattern match also applies to character classes:

```
$ echo “bt” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
$
$ echo “bat” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
bat
$ echo “bet” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
bet
$ echo “beat” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
beat
$ echo “beet” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
beet
$ echo “beeat” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
$
$ echo “baeet” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
$
$ echo “baeaet” | gawk --re-interval ‘/b[ae]{1,2}t/{print $0}’
$
```

This regular expression pattern will match if there are exactly one or two instances of the letter *a* or *e* in the text pattern, but it will fail if there are any more in any combination.

The Pipe Symbol

The pipe symbol allows to you to specify two or more patterns that the regular expression engine uses in a logical0R formula when examining the data stream. If any of the patterns match the data stream text, the text passes. If none of the patterns match, the data stream text fails.

The format for using the pipe symbol is:

```
expr1|expr2|...
```

Here's an example of this:

```
$ echo “The cat is asleep” | gawk ‘/cat|dog/{print $0}’
The cat is asleep
$ echo “The dog is asleep” | gawk ‘/cat|dog/{print $0}’
The dog is asleep
$ echo “The sheep is asleep” | gawk ‘/cat|dog/{print $0}’
$
```

This example looks for the regular expression `cat` or `dog` in the data stream. You can't place any spaces within the regular expressions and the pipe symbol, or they'll be added to the regular expression pattern.

The regular expressions on either side of the pipe symbol can use any regular expression pattern, including character classes, to define the text:

```
$ echo “He has a hat.” | gawk ‘/[ch]at|dog/{print $0}’
He has a hat.
$
```

This example would match `cat`, `hat`, or `dog` in the data stream text.

Grouping Expressions

Regular expression patterns can also be grouped by using parentheses. When you group a regular expression pattern, the group is treated like a standard character. You can apply a special character to the group just as you would to a regular character. For example:

```
$ echo “Sat” | gawk ‘/Sat(urday)?/{print $0}’
Sat
$ echo “Saturday” | gawk ‘/Sat(urday)?/{print $0}’
Saturday
$
```

The grouping of the “urday” ending along with the question mark allows the pattern to match either the full day name Saturday or the abbreviated name Sat.

It's common to use grouping along with the pipe symbol to create groups of possible pattern matches:

```
$ echo “cat” | gawk ‘/(c|b)a(b|t)/{print $0}’
cat
$ echo “cab” | gawk ‘/(c|b)a(b|t)/{print $0}’
cab
$ echo “bat” | gawk ‘/(c|b)a(b|t)/{print $0}’
bat
$ echo “bab” | gawk ‘/(c|b)a(b|t)/{print $0}’
bab
$ echo “tab” | gawk ‘/(c|b)a(b|t)/{print $0}’
$
$ echo “tac” | gawk ‘/(c|b)a(b|t)/{print $0}’
$
```

The pattern `(c|b)a(b|t)` matches any combination of the letters in the first group along with any combination of the letters in the second group.

Regular Expressions in Action

Now that you've seen the rules and a few simple demonstrations of using regular expression patterns, it's time to put that knowledge into action. The following sections demonstrate some common regular expression examples within shell scripts.

Counting Directory Files

To start things out, let's look at a shell script that counts the executable files that are present in the directories defined in your PATH environment variable. To do that, you'll need to parse out the PATH variable into separate directory names. Chapter 5 showed you how to display the PATH environment variable:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/games:/usr/java/
j2sdk1.4.1_01/bin
$
```

Your PATH environment variable will differ, depending on where the applications are located on your Linux system. The key is to recognize that each directory in the PATH is separated by a colon. To get a listing of directories that you can use in a script, you'll have to replace each colon with a space. You now recognize that the sed editor can do just that using a simple regular expression:

```
$ echo $PATH | sed ‘s/:/ /g’
/usr/local/bin /bin /usr/bin /usr/X11R6/bin /usr/games /usr/java/
j2sdk1.4.1_01/bin
$
```

Once you've got the directories separated out, you can use them in a standard `for` statement (see Chapter 12) to iterate through each directory:

```
mypath=‘echo $PATH | sed ‘s/:/ /g’‘
for directory in $mypath
```

```
do
...
done
```

Once you have each directory, you can use the `ls` command to list each file in each directory, and use another `for`statement to iterate through each file, incrementing a counter for each file.

The final version of the script looks like this:

```
$ cat countfiles
#!/bin/bash
# count number of files in your PATH
mypath=‘echo $PATH | sed ‘s/:/ /g’‘
count=0
for directory in $mypath
do
  check=‘ls $directory‘
  for item in $check
  do
     count=$[ $count + 1 ]
  done
  echo “$directory - $count”
  count=0
done
$ ./countfiles
/usr/local/bin - 79
/bin - 86
/usr/bin - 1502
/usr/X11R6/bin - 175
/usr/games - 2
/usr/java/j2sdk1.4.1_01/bin - 27
$
```

Now we're starting to see some of the power behind regular expressions!

Validating a Phone Number

The previous example showed how to incorporate the simple regular expression along with sed to replace characters in a data stream to process data. Often regular expressions are used to validate data to ensure that data is in the correct format for a script.

A common data validation application checks phone numbers. Often, data entry forms request phone numbers, and often customers fail to enter a properly formatted phone number. In the United States, there are several common ways to display a phone number:

```
(123)456-7890
(123) 456-7890
123-456-7890
123.456.7890
```

This leaves four possibilities for how customers can enter their phone number in a form. The regular expression must be robust enough to be able to handle either situation.

When building a regular expression, it's best to start on the left-hand side, and build your pattern to match the possible characters you'll run into. In this example, there may or may not be a left parenthesis in the phone number. This can be matched by using the pattern:

```
^\(?
```

The caret is used to indicate the beginning of the data. Because the left parenthesis is a special character, you must escape it to use it as a normal character. The question mark indicates that the left parenthesis may or may not appear in the data to match.

Next comes the three-digit area code. In the United States, area codes start with the number 2 (no area codes start with the digits 0 or 1), and can go to 9. To match the area code, you'd use the following pattern:

```
[2-9][0-9]{2}
```

This requires that the first character be a digit between 2 and 9, followed by any two digits. After the area code, the ending right parenthesis may or may not be there:

```
\)?
```

After the area code, there can be a space, no space, a dash, or a dot. You can group those using a character group along with the pipe symbol:

```
(| |-|\.)
```

The very first pipe symbol appears immediately after the left parenthesis to match the no space condition. You must use the escape character for the dot; otherwise, it will be interpreted to match any character.

Next comes the three-digit phone exchange number. Nothing special required here:

```
[0-9]{3}
```

After the phone exchange number, you must match a space, a dash, or a dot (this time you don't have to worry about matching no space because there must be at least a space between the phone exchange number and the rest of the number):

```
( |-|\.)
```

Then to finish things off, you must match the four-digit local phone extension at the end of the string:

```
[0-9]{4}$
```

Putting the entire pattern together results in this:

```
^\(?[2-9][0-9]{2}\)?(| |-|\.)[0-9]{3}( |-|\.)[0-9]{4}$
```

You can use this regular expression pattern in the gawk program to filter out bad phone numbers. All you need to do now is create a simple script using the regular expression in a gawk program, and then filter your phone list through the script. Remember that when you use regular expression intervals in the gawk program, you

must use the `--re-interval` command line option or you won't get the correct results.

Here's the script:

```
$ cat isphone
#!/bin/bash
# script to filter out bad phone numbers
gawk --re-interval '/^\(?[2-9][0-9]{2}\)?(| |-|\.)
[0-9]{3}( |-|\.)[0-9]{4}/{print $0}'
$
```

While you can't tell from this listing, the gawk command is on a single line in the shell script. You can then redirect phone numbers to the script for processing:

```
$ echo "317-555-1234" | ./isphone
317-555-1234
$ echo "000-555-1234" | ./isphone
$
```

Or you can redirect an entire file of phone numbers to filter out the invalid ones:

```
$ cat phonelist
000-000-0000
123-456-7890
212-555-1234
(317)555-1234
(202) 555-9876
33523
1234567890
234.123.4567
$ cat phonelist | ./isphone
212-555-1234
(317)555-1234
(202) 555-9876
234.123.4567
$
```

Only the valid phone numbers that match the regular expression pattern appear.

Parsing an E-mail Address

In this day and age, e-mail addresses have become a crucial form of communication. Trying to validate e-mail addresses has become quite a challenge for script builders because there are a myriad of ways to create an e-mail address. The basic form of an e-mail address is:

username@hostname

The *username* value can use any alphanumeric character, along with several special characters:

- Dot
- Dash

- Plus sign
- Underscore

These characters can appear in any combination in a valid e-mail userid. The *hostname* portion of the e-mail address consists of one or more domain names and a server name. The server and domain names must also follow strict naming rules, allowing only alphanumeric characters, along with the special characters:

- Dot
- Underscore

The server and domain names are each separated by a dot, with the server name specified first, any subdomain names specified next, and finally, the top-level domain name without a trailing dot.

At one time there were a fairly limited number of top-level domains, and regular expression pattern builders attempted to add them all in patterns for validation. Unfortunately, as the Internet grew so did the possible top-level domains. This technique is no longer a viable solution.

Let's start building the regular expression pattern from the left side. We know that there can be multiple valid characters in the username. This should be fairly easy:

```
^([a-zA-Z0-9_\-\.\+]+)@
```

This grouping specifies the allowable characters in the username, and the plus sign to indicate that there must be at least one character present. The next character is obviously going to be the @ symbol, no surprises there.

The hostname pattern uses the same technique to match the server name and the subdomain names:

```
([a-zA-Z0-9_\-\.]+)
```

This pattern matches the text:

```
server
server.subdomain
server.subdomain.subdomain
```

There are special rules for the top-level domain. Top-level domains are only alphabetic characters, and they must be no fewer than two characters (used in country codes) and no more than five characters in length. The following is the regular expression pattern for the top-level domain:

```
\.([a-zA-Z]{2,5})$
```

Putting the entire pattern together results in the following:

```
^([a-zA-Z0-9_\-\.\+]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$
```

This pattern will filter out poorly formatted e-mail addresses from a data list. Now you can create your script to implement the regular expression:

```
$ echo “rich@here.now” | ./isemail
rich@here.now
$ echo “rich@here.now.” | ./isemail
$
$ echo “rich@here.n” | ./isemail
$
```

```
$ echo “rich@here-now” | ./isemail
$
$ echo “rich.blum@here.now” | ./isemail
rich.blum@here.now
$ echo “rich_blum@here.now” | ./isemail
rich_blum@here.now
$ echo “rich/blum@here.now” | ./isemail
$
$ echo “rich#blum@here.now” | ./isemail
$
$ echo “rich*blum@here.now” | ./isemail
$
```

Summary

If you manipulate data files in shell scripts, you'll need to become familiar with regular expressions. Regular expressions are implemented in Linux utilities, programming languages, and applications using regular expression engines. A host of different regular expression engines are available in the Linux world. The two most popular are the POSIX Basic Regular Expression (BRE) engine and the POSIX Extended Regular Expression (ERE) engine. The sed editor conforms mainly to the BRE engine, while the gawk program utilizes most features found in the ERE engine.

A regular expression defines a pattern template that's used to filter text in a data stream. The pattern consists of a combination of standard text characters and special characters. The special characters are used by the regular expression engine to match a series of one or more characters, similarly to how wildcard characters work in other applications.

By combining characters and special characters, you can define a pattern to match most any type of data. You can then use the sed editor or gawk program to filter specific data from a larger data stream, or for validating data received from data entry applications.

The next chapter digs deeper into using the sed editor to perform advanced text manipulation. Lots of advanced features are available in the sed editor that make it useful for handling large data streams and filtering out just what you need.