

Introducing sed and gawk

IN THIS CHAPTER

Learning about the sed Editor

Getting introduced to the gawk Editor

Exploring 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 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.

Manipulating Text

Chapter 10 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`.

Getting to know 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. The `sed` editor does these things:

1. Reads one data line at a time from the input
2. Matches that data with the supplied editor commands
3. Changes data in the stream as specified in the commands
4. Outputs the new data to `STDOUT`

After the stream editor matches all the commands against a line of data, it reads the next line of data and repeats the process. After the stream editor processes all the lines of data in the stream, it terminates.

Because the commands are applied sequentially line by line, the `sed` editor makes 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.

Here's the format for using the `sed` command:

```
sed options script file
```

The *options* parameters allow you to customize the behavior of the `sed` command and include the options shown in Table 19-1.

TABLE 19-1 The `sed` Command Options

Option	Description
<code>-e script</code>	Adds commands specified in the script to the commands run while processing the input
<code>-f file</code>	Adds the commands specified in the file to the commands run while processing the input
<code>-n</code>	Doesn't produce output for each command, but waits for the <code>print</code> 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 examine some of the basic commands used by the `sed` editor in this chapter and then look at some of the more advanced commands in Chapter 21.

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 the `sed` 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.txt
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.txt
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.txt
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.txt
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 `sed` program script (`sed` editor command list), and bash continues 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.txt
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 where the closing single quotation mark appears. After the bash shell detects the closing quotation mark, it processes the command. After 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.sed
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1.sed data1.txt
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.

TIP

It can be easy to confuse your `sed` editor script files with your bash shell script files. To eliminate confusion, use a `.sed` file extension on your `sed` script files.

We'll look at some other `sed` editor commands that come in handy for manipulating data in the "Commanding at the `sed` Editor Basics" section. Before that, let's quickly look at the other Linux data editor.

Getting to know the gawk program

Although the `sed` editor is a handy tool for modifying text files on the fly, it has its limitations. Often, you need a more advanced tool for manipulating data in a file, one that provides a more programming-like environment allowing you to modify and reorganize data in a file. This is where `gawk` comes in.

NOTE

The `gawk` program is not installed by default on all distributions. If your Linux distribution does not have the `gawk` program, install the `gawk` package using Chapter 9 as a guide.

The `gawk` program is the GNU version of the original `awk` program in Unix. The `gawk` program takes stream editing one step further than the `sed` editor by providing a programming language instead of just editor commands. Within the `gawk` programming language, you can do the following:

- Define variables to store data.
- Use arithmetic and string operators to operate on data.
- Use structured programming concepts, such as `if-then` statements and loops, to add logic to your data processing.
- Generate formatted reports by extracting data elements within the data file and repositioning them in another order or format.

The `gawk` program's report-generating capabilities are often used for extracting data elements from large bulky text files and formatting them into a readable report. The perfect example of this is formatting log files. Trying to pore through lines of errors in a log file can be difficult. The `gawk` program allows you to filter just the data elements you want to view from the log file, and then you can format them in a manner that makes reading the important data easier.

Visiting the `gawk` command format

Here's the basic format of the `gawk` program:

```
gawk options program file
```

Table 19-2 shows the options available with the `gawk` program.

TABLE 19-2 The `gawk` Options

Option	Description
<code>-F fs</code>	Specifies a file separator for delineating data fields in a line
<code>-f file</code>	Specifies a file name to read the program from
<code>-v var=value</code>	Defines a variable and default value used in the <code>gawk</code> program
<code>-mf N</code>	Specifies the maximum number of fields to process in the data file
<code>-mr N</code>	Specifies the maximum record size in the data file
<code>-W keyword</code>	Specifies the compatibility mode or warning level for <code>gawk</code>

The command line options provide an easy way to customize features in the `gawk` program. We'll look more closely at these as we explore `gawk`.

The power of `gawk` is in the program script. You can write scripts to read the data within a text line and then manipulate and display the data to create any type of output report.

Reading the program script from the command line

A `gawk` program script is defined by opening and closing braces. You must place script commands between the two braces (`{}`). If you incorrectly use a parenthesis instead of a brace to enclose your `gawk` script, you get error messages, similar to the following:

```
$ gawk '{print "Hello World!"}'  
gawk: (print "Hello World!")  
gawk: ^ syntax error
```

Because the `gawk` command line assumes that the script is a single text string, you must also enclose your script in single quotation marks. Here's an example of a simple `gawk` program script specified on the command line:

```
$ gawk '{print "Hello World!"}'
```

The program script defines a single command, the `print` command. The `print` command does what it says: It prints text to `STDOUT`. If you try running this command, you'll be somewhat disappointed, because nothing happens right away. Because no filename was defined in the command line, the `gawk` program retrieves data from `STDIN`. When you run the program, it just waits for text to come in via `STDIN`.

If you type a line of text and press the Enter key, gawk runs the text through the program script. Just like the sed editor, the gawk program executes the program script on each line of text available in the data stream. Because the program script is set to display a fixed text string, no matter what text you enter in the data stream, you get the same text output:

```
$ gawk '{print "Hello World!"}'
This is a test
Hello World!
hello
Hello World!
This is another test
Hello World!
```

To terminate the gawk program, you must signal that the data stream has ended. The bash shell provides a key combination to generate an End-of-File (EOF) character. The Ctrl+D key combination generates an EOF character in bash. Using that key combination terminates the gawk program and returns you to a command line interface prompt.

Using data field variables

One of the primary features of gawk is its ability to manipulate data in the text file. It does this by automatically assigning a variable to each data element in a line. By default, gawk assigns the following variables to each data field it detects in the line of text:

- \$0 represents the entire line of text.
- \$1 represents the first data field in the line of text.
- \$2 represents the second data field in the line of text.
- \$n represents the *n*th data field in the line of text.

Each data field is determined in a text line by a *field separation character*. When gawk reads a line of text, it delineates each data field using the defined field separation character. The default field separation character in gawk is any whitespace character (such as the tab or space characters).

Here's an example gawk program that reads a text file and displays only the first data field value:

```
$ cat data2.txt
One line of test text.
Two lines of test text.
Three lines of test text.
$
$ gawk '{print $1}' data2.txt
One
Two
Three
$
```

This program uses the `$1` field variable to display only the first data field for each line of text.

If you're reading a file that uses a different field separation character, you can specify it by using the `-F` option:

```
$ gawk -F: '{print $1}' /etc/passwd
root
bin
daemon
adm
lp
sync
shutdown
halt
mail
[...]
```

This short program displays the first data field in the password file on the system. Because the `/etc/passwd` file uses a colon to separate the data fields, if you want to separate each data element, you must specify it as the field separation character in the `gawk` options.

Using multiple commands in the program script

A programming language wouldn't be very useful if you could only execute one command. The `gawk` programming language allows you to combine commands into a normal program. To use multiple commands in the program script specified on the command line, just place a semicolon between each command:

```
$ echo "My name is Rich" | gawk '{$4="Christine"; print $0}'
My name is Christine
$
```

The first command assigns a value to the `$4` field variable. The second command then prints the entire data field. Notice from the output that the `gawk` program replaced the fourth data field in the original text with the new value.

You can also use the secondary prompt to enter your program script commands one line at a time:

```
$ gawk '{
> $4="Christine"
> print $0}'
My name is Rich
My name is Christine
$
```

After you open the single quotation mark, the bash shell provides the secondary prompt to prompt you for more data. You can add your commands one at a time on each line until you

enter the closing single quotation mark. Because no filename was defined in the command line, the gawk program retrieves data from STDIN. When you run the program, it waits for text to come in via STDIN. To exit the program, just press the Ctrl+D key combination to signal the end of the data.

Reading the program from a file

As with the sed editor, the gawk editor allows you to store your programs in a file and refer to them in the command line:

```
$ cat script2.gawk
{print $1 "'s home directory is " $6}
$
$ gawk -F: -f script2.gawk /etc/passwd
root's home directory is /root
bin's home directory is /bin
daemon's home directory is /sbin
adm's home directory is /var/adm
lp's home directory is /var/spool/lpd
[...]
Christine's home directory is /home/Christine
Samantha's home directory is /home/Samantha
Timothy's home directory is /home/Timothy
$
```

The script2.gawk program script uses the print command again to print the /etc/passwd file's home directory data field (field variable \$6) and the userid data field (field variable \$1).

You can specify multiple commands in the program file. To do so, just place each command on a separate line. You don't need to use semicolons:

```
$ cat script3.gawk
{
text = "'s home directory is "
print $1 text $6
}
$
$ gawk -F: -f script3.gawk /etc/passwd
root's home directory is /root
bin's home directory is /bin
daemon's home directory is /sbin
adm's home directory is /var/adm
lp's home directory is /var/spool/lpd
[...]
Christine's home directory is /home/Christine
Samantha's home directory is /home/Samantha
Timothy's home directory is /home/Timothy
$
```

The `script3.gawk` program script defines a variable to hold a text string used in the `print` command. Notice that `gawk` programs don't use a dollar sign when referencing a variable's value, as a shell script does.

Running scripts before processing data

The `gawk` program also allows you to specify when the program script is run. By default, `gawk` reads a line of text from the input and then executes the program script on the data in the line of text. Sometimes, you may need to run a script before processing data, such as to create a header section for a report. The `BEGIN` keyword is used to accomplish this. It forces `gawk` to execute the program script specified after the `BEGIN` keyword, before `gawk` reads the data:

```
$ gawk 'BEGIN {print "Hello World!"}'  
Hello World!  
$
```

This time the `print` command displays the text before reading any data. However, after it displays the text, it quickly exits, without waiting for any data.

The reason for this is that the `BEGIN` keyword only applies the specified script before it processes any data. If you want to process data with a normal program script, you must define the program using another script section:

```
$ cat data3.txt  
Line 1  
Line 2  
Line 3  
$  
$ gawk 'BEGIN {print "The data3 File Contents:"}  
> {print $0}' data3.txt  
The data3 File Contents:  
Line 1  
Line 2  
Line 3  
$
```

Now after `gawk` executes the `BEGIN` script, it uses the second script to process any file data. Be careful when doing this; both of the scripts are still considered one text string on the `gawk` command line. You need to place your single quotation marks accordingly.

Running scripts after processing data

Like the `BEGIN` keyword, the `END` keyword allows you to specify a program script that `gawk` executes after reading the data:

```
$ gawk 'BEGIN {print "The data3 File Contents:"}  
> {print $0}'
```

```
> END {print "End of File"}' data3.txt
The data3 File Contents:
Line 1
Line 2
Line 3
End of File
$
```

When the gawk program is finished printing the file contents, it executes the commands in the END script. This is a great technique to use to add footer data to reports after all the normal data has been processed.

You can put all these elements together into a nice little program script file to create a full report from a simple data file:

```
$ cat script4.gawk
BEGIN {
print "The latest list of users and shells"
print " UserID \t Shell"
print "----- \t -----"
FS=":"
}

{
print $1 "      \t " $7
}

END {
print "This concludes the listing"
}
$
```

This script uses the BEGIN script to create a header section for the report. It also defines a special variable called FS. This is yet another way to define the field separation character. This way, you don't have to depend on the script's user to define the field separation character in the command line options.

Here's a somewhat truncated output from running this gawk program script:

```
$ gawk -f script4.gawk /etc/passwd
The latest list of users and shells
  UserID      Shell
-----
root          /bin/bash
bin           /sbin/nologin
daemon        /sbin/nologin
[...]
Christine     /bin/bash
```

```
mysql          /bin/bash
Samantha       /bin/bash
Timothy        /bin/bash
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. The `\t` within the `print` command produces some nicely formatted tabbed output.

This gives you a small taste of the power available when you use simple `gawk` scripts. Chapter 22 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.

Commanding at the `sed` Editor Basics

The key to successfully using the `sed` editor is to know its myriad of commands and formats, which help you to 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.

Introducing 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.

Substituting flags

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

```
$ cat data4.txt
This is a test of the test script.
This is the second test of the test script.
$
$ sed 's/test/trial/' data4.txt
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 replaces only 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
```

Four types of substitution flags are available:

- A number, indicating the pattern occurrence for which new text should be substituted
- `g`, indicating that new text should be substituted for all occurrences of the existing text
- `p`, indicating that the contents of the original line should be printed
- `w file`, which means to 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' data4.txt
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 replaces the pattern only 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' data4.txt
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 data5.txt
This is a test line.
This is a different line.
$
$ sed -n 's/test/trial/p' data5.txt
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.txt' data5.txt
This is a trial line.
This is a different line.
$
$ cat test.txt
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.

Replacing characters

Sometimes, 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 want to apply a command only 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.

Addressing the numeric line

When using numeric line addressing, you reference lines using their line position in the text stream. The `sed` editor 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 `sed` command will be applied:

```
$ sed '2s/dog/cat/' data1.txt
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.txt
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.txt
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. This is the format:

```
/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 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:502:502::/home/Samantha:/bin/bash
$
$ sed '/Samantha/s/bash/csh/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
[...]
Christine:x:501:501:Christine B:/home/Christine:/bin/bash
Samantha:x:502:502::/home/Samantha:/bin/csh
Timothy:x:503:503::/home/Timothy:/bin/bash
$
```

The command was applied only to the line with the matching text pattern. Although 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 20 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 processes each command listed on the address line(s):

```
$ sed '2{
> s/fox/elephant/
> s/dog/cat/
```



```
> }' data1.txt
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 specify an address range before the grouped commands:

```
$ sed '3,$ {
> s/brown/green/
> s/lazy/active/
> }' data1.txt
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 the commands to all 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, you can use the delete command.

The delete command, d, pretty much does what it says. It deletes 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 the lines are deleted from the stream:

```
$ cat data1.txt
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.txt
$
```

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:

```
$ cat data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
```

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

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

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

In addition, you must be careful because the delete feature “turns on” whenever the `sed` editor detects the start pattern in the data stream. This may produce an unexpected result:

```
$ cat data7.txt
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.txt
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, because 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' data7.txt
$
```

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 is 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 the line to insert on a separate line by itself. Here's the format for doing this:

```
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 get the secondary prompt to enter the new line of data. You must complete the `sed` editor command on this line. After you enter the ending single quotation mark, the bash shell processes 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.

Here's an example of inserting a new line before line 3 in the data stream:

```
$ sed '3i\
> This is an inserted line.' data6.txt
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.
$
```

Here's an example of appending a new line after line 3 in the data stream:

```
$ sed '3a\
> This is an appended line.' data6.txt
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.' data6.txt  
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.' data6.txt  
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.' data6.txt  
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.' data6.txt  
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 changes any line of text in the data stream that it matches.

```
$ cat data8.txt  
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 yet another line.  
This is the last line in the file.  
$  
$ sed '/number 1/c\  
> This is a changed line of text.' data8.txt  
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.' data6.txt  
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.

Transforming characters

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 the *inchars* and *outchars* are not the same length, the sed editor produces an error message.

Here's a simple example of using the transform command:

```
$ sed 'y/123/789/' data8.txt
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 "Introducing more substitution options" section showed you how to use the *p* flag with the substitution command to display lines that the sed editor changed. In addition, three commands that can be used to print information from the data stream:

- The *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 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, this command doesn't offer 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:

```
$ cat data6.txt
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
$ sed -n '/number 3/p' data6.txt
This is line number 3.
$
```

By using the `-n` option on the command line, you can suppress all the other lines and print only 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' data6.txt
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
> }' data6.txt
This is line number 3.
This is test number 3.
$
```

This `sed` editor command searches for lines that contain the number 3 and 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:

```
$ cat data1.txt
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 '=' data1.txt
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
> }' data6.txt
4
This is line number 4.
$
```

By using the `-n` option, you can have the `sed` editor display 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 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.txt
This    line    contains          tabs.
```

```
$  
$ sed -n '1' data9.txt  
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 if necessary:

```
$ cat data10.txt  
This line contains an escape character.  
$  
$ sed -n '1' data10.txt  
This line contains an escape character. \a$  
$
```

The `data10.txt` file contains an escape control code, which generates a bell sound. When you use the `cat` command to display the text file, you don't see the escape control code; you just hear the sound (if your speakers are turned on). However, using the `list` command, you can display the escape control code used.

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. Here's the format for the `w` command:

```
[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 in `sed`, such as a single line number, a text pattern, or a range of line numbers or text patterns.

Here's an example that prints only the first two lines of a data stream to a text file:

```
$ sed '1,2w test.txt' data6.txt  
This is line number 1.  
This is line number 2.  
This is line number 3.  
This is line number 4.  
$  
$ cat test.txt  
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.txt
Blum, R      Browncoat
McGuinness, A Alliance
Bresnahan, C Browncoat
Harken, C   Alliance
$
$ sed -n '/Browncoat/w Browncoats.txt' data11.txt
$
$ cat Browncoats.txt
Blum, R      Browncoat
Bresnahan, C Browncoat
$
```

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.

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

```
[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.txt
This is an added line.
This is the second added line.
$
$ sed '3r data12.txt' data6.txt
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 the text lines in the data file. The same technique works when using a text pattern address:

```
$ sed '/number 2/r data12.txt' data6.txt
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.txt' data6.txt
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 stored in a text file that looked like this:

```
$ cat notice.std
Would the following people:
LIST
please report to the ship's captain.
$
```

The form letter uses the generic placeholder `LIST` in place of a list of people. To insert the list of people after the placeholder, you just 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.txt
> d
> }' notice.std
Would the following people:
Blum, R      Browncoat
McGuinness, A Alliance
Bresnahan, C Browncoat
```

```
Harken, C      Alliance  
please report to the ship's captain.  
$
```

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

Summary

Shell scripts can do lots of work on their own, but it's often difficult to manipulate data with just a shell script. Linux provides two handy utilities to help 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 Unix `awk` 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.