

Advanced sed

IN THIS CHAPTER

Using multiline commands

Understanding the hold space

Negating a command

Changing the flow

Replacing via a pattern

Using sed in scripts

Creating sed utilities

Chapter 19 showed you how to use the basics of the `sed` editor to manipulate text in data streams. The basic `sed` editor commands are capable of handling most of your everyday text-editing requirements. This chapter looks at the more advanced features that the `sed` editor has to offer. These are features that you might not use as often. But when you need them, it's nice to know that they're available and how to use them.

Looking at Multiline Commands

When using the basic `sed` editor commands, you may have noticed a limitation. All the `sed` editor commands perform functions on a single line of data. As the `sed` editor reads a data stream, it divides the data into lines based on the presence of newline characters. The `sed` editor handles each data line one at a time, processing the defined script commands on the data line, and then moving on to the next line and repeating the processing.

Sometimes, you need to perform actions on data that spans more than one line. This is especially true if you're trying to find or replace a phrase.

For example, if you're looking for the phrase `Linux System Administrators Group` in your data, it's quite possible that the phrase's words can be split onto two lines. If you processed the text using a normal `sed` editor command, it would be impossible to detect the split phrase.

Fortunately, the designers behind the `sed` editor thought of that situation and devised a solution. The `sed` editor includes three special commands that you can use to process multiline text:

- `N` adds the next line in the data stream to create a multiline group for processing.
- `D` deletes a single line in a multiline group.
- `P` prints a single line in a multiline group.

The following sections examine these multiline commands more closely and demonstrate how you can use them in your scripts.

Navigating the next command

Before you can examine the multiline `next` command, you first need to look at how the single-line version of the `next` command works. After you know what that command does, it's much easier to understand how the multiline version of the `next` command operates.

Using the single-line `next` command

The lowercase `n` command tells the `sed` editor to move to the next line of text in the data stream, without going back to the beginning of the commands. Remember that normally the `sed` editor processes all the defined commands on a line before moving to the next line of text in the data stream. The single-line `next` command alters this flow.

This may sound somewhat complicated, and sometimes it is. In this example, you have a data file that contains five lines, two of which are empty. The goal is to remove the blank line after the header line but leave the blank line before the last line intact. If you write a `sed` script to just remove blank lines, you remove both blank lines:

```
$ cat data1.txt
This is the header line.

This is a data line.

This is the last line.
$
$ sed '/^$/d' data1.txt
This is the header line.
This is a data line.
This is the last line.
$
```

Because the line you want to remove is blank, you don't have any text you can search for to uniquely identify the line. The solution is to use the `n` command. In this next example, the

script looks for a unique line that contains the word `header`. After the script identifies that line, the `n` command moves the `sed` editor to the next line of text, which is the empty line.

```
$ sed '/header/{n ; d}' data1.txt
This is the header line.
This is a data line.

This is the last line.
$
```

At that point, the `sed` editor continues processing the command list, which uses the `d` command to delete the empty line. When the `sed` editor reaches the end of the command script, it reads the next line of text from the data stream and starts processing commands from the top of the command script. The `sed` editor does not find another line with the word `header`; thus, no further lines are deleted.

Combining lines of text

Now that you've seen the single-line `next` command, you can look at the multiline version. The single-line `next` command moves the next line of text from the data stream into the processing space (called the *pattern space*) of the `sed` editor. The multiline version of the `next` command (which uses a capital `N`) adds the next line of text to the text already in the pattern space.

This has the effect of combining two lines of text from the data stream into the same pattern space. The lines of text are still separated by a newline character, but the `sed` editor can now treat both lines of text as one line.

Here's a demonstration of how the `N` command operates:

```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed '/first/{ N ; s/\n/ / }' data2.txt
This is the header line.
This is the first data line. This is the second data line.
This is the last line.
$
```

The `sed` editor script searches for the line of text that contains the word "first" in it. When it finds the line, it uses the `N` command to combine the next line with that line. It then uses the substitution command (`s`) to replace the newline character with a space. The result is that the two lines in the text file appear as one line in the `sed` editor output.

This has a practical application if you're searching for a text phrase that may be split between two lines in the data file. Here's an example:

```
$ cat data3.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All System Administrators should attend.
Thank you for your attendance.
$
$ sed 'N ; s/System Administrator/Desktop User/' data3.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All Desktop Users should attend.
Thank you for your attendance.
$
```

The substitution command is looking for the specific two-word phrase System Administrator in the text file. In the single line where the phrase appears, everything is fine; the substitution command can replace the text. But in the situation where the phrase is split between two lines, the substitution command doesn't recognize the matching pattern.

The N command helps solve this problem:

```
$ sed 'N ; s/System.Administrator/Desktop User/' data3.txt
On Tuesday, the Linux Desktop User's group meeting will be held.
All Desktop Users should attend.
Thank you for your attendance.
$
```

By using the N command to combine the next line with the line where the first word is found, you can detect when a line split occurs in the phrase.

Notice that the substitution command uses a wildcard pattern (.) between the word System and the word Administrator to match both the space and the newline situation. However, when it matched the newline character, it removed it from the string, causing the two lines to merge into one line. This may not be exactly what you want.

To solve this problem, you can use two substitution commands in the sed editor script, one to match the multiline occurrence and one to match the single-line occurrence:

```
$ sed 'N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data3.txt
On Tuesday, the Linux Desktop
```

```
User's group meeting will be held.
All Desktop Users should attend.
Thank you for your attendance.
$
```

The first substitution command specifically looks for the newline character between the two search words and includes it in the replacement string. This allows you to add the newline character in the same place in the new text.

There's still one subtle problem with this script, however. The script always reads the next line of text into the pattern space before executing the `sed` editor commands. When it reaches the last line of text, there isn't a next line of text to read, so the `N` command causes the `sed` editor to stop. If the matching text is on the last line in the data stream, the commands don't catch the matching data:

```
$ cat data4.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All System Administrators should attend.
$
$ sed 'N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All System Administrators should attend.
$
```

Because the `System Administrator` text appears in the last line in the data stream, the `N` command misses it, as there isn't another line to read into the pattern space to combine. You can easily resolve this problem by moving your single-line commands before the `N` command and having only the multiline commands appear after the `N` command, like this:

```
$ sed '
> s/System Administrator/Desktop User/
> N
> s/System\nAdministrator/Desktop\nUser/
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All Desktop Users should attend.
$
```

Now, the substitution command that looks for the phrase in a single line works just fine on the last line in the data stream, and the multiline substitution command covers the occurrence in the middle of the data stream.

Navigating the multiline delete command

Chapter 19 introduced the single-line delete command (`d`). The `sed` editor uses it to delete the current line in the pattern space. If you're working with the `N` command, however, you must be careful when using the single-line delete command:

```
$ sed 'N ; /System\nAdministrator/d' data4.txt
All System Administrators should attend.
$
```

The delete command looked for the words `System` and `Administrator` in separate lines and deleted both of the lines in the pattern space. This may or may not have been what you intended.

The `sed` editor provides the multiline delete command (`D`), which deletes only the first line in the pattern space. It removes all characters up to and including the newline character:

```
$ sed 'N ; /System\nAdministrator/D' data4.txt
Administrator's group meeting will be held.
All System Administrators should attend.
$
```

The second line of text, added to the pattern space by the `N` command, remains intact. This comes in handy if you need to remove a line of text that appears before a line that you find a data string in.

Here's an example of removing a blank line that appears before the first line in a data stream:

```
$ cat data5.txt

This is the header line.
This is a data line.

This is the last line.
$
$ sed '/^$/N ; /header/D' data5.txt
This is the header line.
This is a data line.

This is the last line.
$
```

This `sed` editor script looks for blank lines and then uses the `N` command to add the next line of text into the pattern space. If the new pattern space contents contain the word

header, the `D` command removes the first line in the pattern space. Without the combination of the `N` and `D` commands, it would be impossible to remove the first blank line without removing all other blank lines.

Navigating the multiline print command

By now, you're probably catching on to the difference between the single-line and multiline versions of the commands. The multiline print command (`P`) follows along using the same technique. It prints only the first line in a multiline pattern space. This includes all characters up to the newline character in the pattern space. It is used in much the same way as the single-line `p` command to display text when you use the `-n` option to suppress output from the script.

```
$ sed -n 'N ; /System\nAdministrator/P' data3.txt
On Tuesday, the Linux System
$
```

When the multiline match occurs, the `P` command prints only the first line in the pattern space. The power of the multiline `P` command comes into play when you combine it with the `N` and `D` multiline commands.

The `D` command has a unique feature in that it forces the `sed` editor to return to the beginning of the script and repeat the commands on the same pattern space (it doesn't read a new line of text from the data stream). By including the `N` command in the command script, you can effectively single-step through the pattern space, matching multiple lines together.

Next, by using the `P` command, you can print the first line, and then using the `D` command, you can delete the first line and loop back to the beginning of the script. When you are back at the script's beginning, the `N` command reads in the next line of text and starts the process all over again. This loop continues until you reach the end of the data stream.

Holding Space

The *pattern space* is an active buffer area that holds the text examined by the `sed` editor while it processes commands. However, it isn't the only space available in the `sed` editor for storing text.

The `sed` editor utilizes another buffer area called the *hold space*. You can use the hold space to temporarily hold lines of text while working on other lines in the pattern space. The five commands associated with operating with the hold space are shown in Table 21-1.

TABLE 21-1 The sed Editor Hold Space Commands

Command	Description
h	Copies pattern space to hold space
H	Appends pattern space to hold space
g	Copies hold space to pattern space
G	Appends hold space to pattern space
x	Exchanges contents of pattern and hold spaces

These commands let you copy text from the pattern space to the hold space. This frees up the pattern space to load another string for processing.

Usually, after using the `h` or `H` commands to move a string to the hold space, eventually you want to use the `g`, `G`, or `x` commands to move the stored string back into the pattern space (otherwise, you wouldn't have cared about saving them in the first place).

With two buffer areas, trying to determine what line of text is in which buffer area can sometimes get confusing. Here's a short example that demonstrates how to use the `h` and `g` commands to move data back and forth between the `sed` editor buffer spaces:

```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed -n '/first/ {h ; p ; n ; p ; g ; p }' data2.txt
This is the first data line.
This is the second data line.
This is the first data line.
$
```

Look at the preceding code example step by step:

1. The `sed` script uses a regular expression in the address to filter the line containing the word `first`.
2. When the line containing the word `first` appears, the initial command in `{}`, the `h` command, places the line in the hold space.
3. The next command, the `p` command, prints the contents of the pattern space, which is still the first data line.
4. The `n` command retrieves the next line in the data stream (This is the second data line) and places it in the pattern space.
5. The `p` command prints the contents of the pattern space, which is now the second data line.

6. The `g` command places the contents of the hold space (This is the first data line) back into the pattern space, replacing the current text.
7. The `p` command prints the current contents of the pattern space, which is now back to the first data line.

By shuffling the text lines around using the hold space, you can force the first data line to appear after the second data line in the output. If you just drop the first `p` command, you can output the two lines in reverse order:

```
$ sed -n '/first/ {h ; n ; p ; g ; p }' data2.txt
This is the second data line.
This is the first data line.
$
```

This is the start of something useful. You can use this technique to create a `sed` script that reverses an entire file of text lines! To do that, however, you need to see the negating feature of the `sed` editor, which is what the next section is all about.

Negating a Command

Chapter 19 showed that the `sed` editor applies commands either to every text line in the data stream or to lines specifically indicated by either a single address or an address range. You can also configure a command to not apply to a specific address or address range in the data stream.

The exclamation mark command (!) is used to negate a command. This means in situations where the command would normally have been activated, it isn't. Here's an example demonstrating this feature:

```
$ sed -n '/header/!p' data2.txt
This is the first data line.
This is the second data line.
This is the last line.
$
```

The normal `p` command would have printed only the line in the `data2` file that contained the word `header`. By adding the exclamation mark, the opposite happens — all lines in the file are printed except the one that contained the word `header`.

Using the exclamation mark comes in handy in several applications. Recall that earlier in the chapter, the “Navigating the next command” section showed a situation where a `sed` editor command wouldn't operate on the last line of text in the data stream because there wasn't a line after it. You can use the exclamation point to fix that problem:

```
$ sed 'N;
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
```

```
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All System Administrators should attend.
$
$ sed '$!N;
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All Desktop Users should attend.
$
```

This example shows the exclamation mark used with the `N` command, along with the dollar sign (\$) special address. The dollar sign represents the last line of text in the data stream, so when the `sed` editor reaches the last line, it doesn't execute the `N` command. However, for all other lines, it does execute the command.

Using this technique, you can reverse the order of text lines in a data stream. To reverse the order of the lines as they appear in the text stream (display the last line first and the first line last), you need to do some fancy footwork using the hold space.

The pattern you need to work with goes like this:

1. Place a line in the pattern space.
2. Place the line from the pattern space to the hold space.
3. Put the next line of text in the pattern space.
4. Append the hold space to the pattern space.
5. Place everything in the pattern space into the hold space.
6. Repeat Steps 3 through 5 until you've put all the lines in reverse order in the hold space.
7. Retrieve the lines, and print them.

Figure 21-1 diagrams what this looks like in more detail.

When using this technique, you do not want to print lines as they are processed. This means using the `-n` command line option for `sed`. The next thing to determine is how to append the hold space text to the pattern space text. This is done by using the `G` command. The only problem is that you don't want to append the hold space to the first line of text processed. This is easily solved by using the exclamation mark command:

```
1!G
```

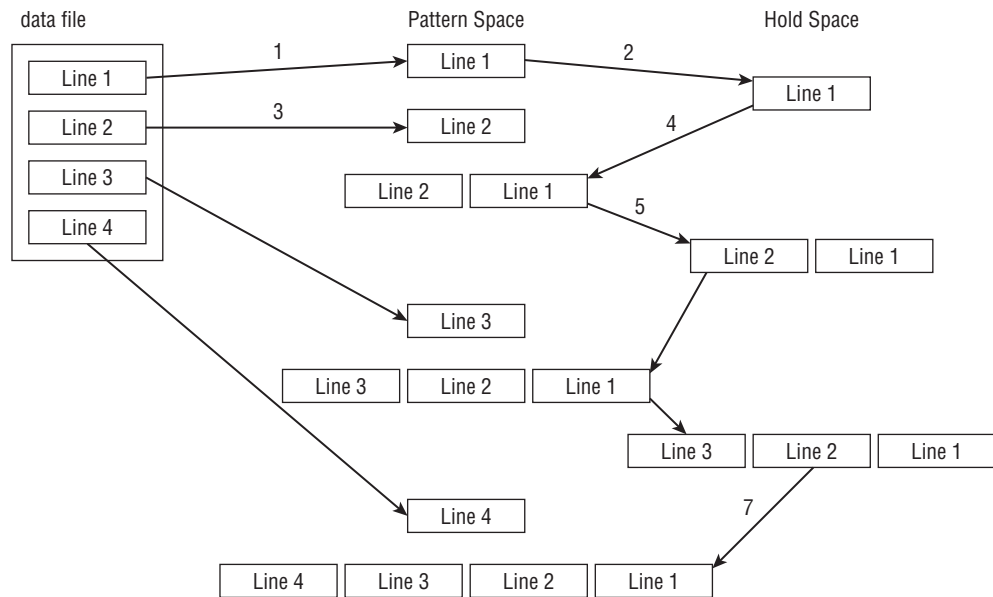
The next step is to place the new pattern space (the text line with the appended reverse lines) into the hold space. This is simple enough; just use the `h` command.

When you've got the entire data stream in the pattern space in reverse order, you just need to print the results. You know you have the entire data stream in the pattern space when you've reached the last line in the data stream. To print the results, just use the following command:

```
$p
```

FIGURE 21-1

Reversing the order of a text file using the hold space



Those are the pieces you need to create your line-reversing sed editor script. Now try it out in a test run:

```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed -n '{1!G ; h ; $p }' data2.txt
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
$
```

The `sed` editor script performed as expected. The output from the script reverses the original lines in the text file. This demonstrates the power of using the hold space in your `sed` scripts. It provides an easy way to manipulate the order of lines in the script output.

NOTE

In case you're wondering, a bash shell command can perform the function of reversing a text file. The `tac` command displays a text file in reverse order. You probably noticed the clever name of the command because it performs the reverse function of the `cat` command.

Changing the Flow

Normally, the `sed` editor processes commands starting at the top and proceeding toward the end of the script (the exception is the `D` command, which forces the `sed` editor to return to the top of the script without reading a new line of text). The `sed` editor provides a method for altering the flow of the command script, producing a result similar to that of a structured programming environment.

Branching

In the previous section, you saw how the exclamation mark command is used to negate the effect of a command on a line of text. The `sed` editor provides a way to negate an entire section of commands, based on an address, an address pattern, or an address range. This allows you to perform a group of commands only on a specific subset within the data stream.

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

```
[address]b [label]
```

The `address` parameter determines which line or lines of data trigger the branch command. The `label` parameter defines the location to branch to. If the `label` parameter is not present, the branch command proceeds to the end of the script.

```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed '{2,3b ; s/This is/Is this/ ; s/line./test?/}' data2.txt
Is this the header test?
This is the first data line.
This is the second data line.
Is this the last test?
$
```

The `branch` command skips the two substitution commands for the second and third lines in the data stream.

Instead of going to the end of the script, you can define a label for the `branch` command to jump to. Labels start with a colon and can be up to seven characters in length:

```
:label2
```

To specify the label, just add it after the `b` command. Using labels allows you to skip commands that match the branch address but still process other commands in the script:

```
$ sed '{/first/b jump1 ; s/This is the/No jump on/
> :jump1
> s/This is the/Jump here on/}' data2.txt
No jump on header line
Jump here on first data line
No jump on second data line
No jump on last line
$
```

The `branch` command specifies that the program should jump to the script line labeled `jump1` if the matching text “first” appears in the line. If the branch command pattern doesn’t match, the `sed` editor continues processing commands in the script, including the command after the branch label. (Thus, all three substitution commands are processed on lines that don’t match the branch pattern.)

If a line matches the branch pattern, the `sed` editor branches to the branch label line. Thus, only the last substitution command is executed.

The example shows branching to a label further down in the `sed` script. You can also branch to a label that appears earlier in the script, thus creating a looping effect:

```
$ echo "This, is, a, test, to, remove, commas." | sed -n '{
> :start
> s/,//lp
> b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
^C
$
```

Each script iteration removes the first occurrence of a comma from the text string and prints the string. There’s one catch to this script: It never ends. This situation creates an endless loop, searching for commas until you manually stop it by sending a signal with the `Ctrl+C` key combination.

To prevent this problem, you should specify an address pattern for the branch command to look for. If the pattern isn't present, the branching should stop:

```
$ echo "This, is, a, test, to, remove, commas." | sed -n '{
> :start
> s/,//lp
> /,/b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
$
```

Now the branch command branches only if there's a comma in the line. After the last comma has been removed, the branch command doesn't execute, allowing the script to properly finish.

Testing

Similar to the branch command, the test command (t) is also used to modify the flow of the sed editor script. Instead of jumping to a label based on an address, the test command jumps to a label based on the outcome of a substitution command.

If the substitution command successfully matches and substitutes a pattern, the test command branches to the specified label. If the substitution command doesn't match the specified pattern, the test command doesn't branch.

The test command uses the same format as the branch command:

```
[address]t [label]
```

Like the branch command, if you don't specify a label, sed branches to the end of the script if the test succeeds.

The test command provides a cheap way to perform a basic if-then statement on the text in the data stream. For example, if you don't need to make a substitution if another substitution was made, the test command can help:

```
$ sed '{
> s/first/matched/
> t
> s/This is the/No match on/
> }' data2.txt
No match on header line
```

```

This is the matched data line
No match on second data line
No match on last line
$

```

The first substitution command looks for the pattern `text first`. If it matches the pattern in the line, it replaces the text, and the `test` command jumps over the second substitution command. If the first substitution command doesn't match the pattern, the second substitution command is processed.

Using the `test` command, you can clean up the loop you tried using the branch command:

```

$ echo "This, is, a, test, to, remove, commas. " | sed -n '{
> :start
> s/,//lp
> t start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
$

```

When there are no more substitutions to make, the `test` command doesn't branch and continues with the rest of the script.

Replacing via a Pattern

You've seen how to use patterns in the `sed` commands to replace text in the data stream. However, when using wildcard characters it's not easy to know exactly what text will match the pattern.

For example, say that you want to place double quotation marks around a word you match in a line. That's simple enough if you're just looking for one word in the pattern to match:

```

$ echo "The cat sleeps in his hat." | sed 's/cat/"cat"/'
The "cat" sleeps in his hat.
$

```

But what if you use a wildcard character (`.`) in the pattern to match more than one word?

```

$ echo "The cat sleeps in his hat." | sed 's/.at"/.at"/g'
The ".at" sleeps in his ".at".
$

```

The substitution string used the dot wildcard character to match any occurrence of a letter followed by “at”. Unfortunately, the replacement string doesn’t match the wildcard character value of the matching word.

Using the ampersand

The `sed` editor has a solution for you. The ampersand symbol (`&`) is used to represent the matching pattern in the substitution command. Whatever text matches the pattern defined, you can use the ampersand symbol to recall it in the replacement pattern. This lets you manipulate whatever word matches the pattern defined:

```
$ echo "The cat sleeps in his hat." | sed 's/.at/"&"/g'
The "cat" sleeps in his "hat".
$
```

When the pattern matches the word `cat`, “cat” appears in the substituted word. When it matches the word `hat`, “hat” appears in the substituted word.

Replacing individual words

The ampersand symbol retrieves the entire string that matches the pattern you specify in the substitution command. Sometimes, you’ll only want to retrieve a subset of the string. You can do that, too, but it’s a little tricky.

The `sed` editor uses parentheses to define a substring component within the substitution pattern. You can then reference each substring component using a special character in the replacement pattern. The replacement character consists of a backslash and a number. The number indicates the substring component’s position. The `sed` editor assigns the first component the character `\1`, the second component the character `\2`, and so on.

CAUTION

When you use parentheses in the substitution command, you must use the escape character to identify them as grouping characters and not normal parentheses. This is the reverse of when you escape other special characters.

Look at an example of using this feature in a `sed` editor script:

```
$ echo "The System Administrator manual" | sed '
> s/\(System\) Administrator/\1 User/'
The System User manual
$
```

This substitution command uses one set of parentheses around the word `System` identifying it as a substring component. It then uses the `\1` in the replacement pattern to recall the first identified component. This isn’t too exciting, but it can really be useful when working with wildcard patterns.

If you need to replace a phrase with just a single word, that's a substring of the phrase, but that substring just happens to be using a wildcard character; using substring components is a lifesaver:

```
$ echo "That furry cat is pretty" | sed 's/furry \(.at\)\/\1/'
That cat is pretty
$
$ echo "That furry hat is pretty" | sed 's/furry \(.at\)\/\1/'
That hat is pretty
$
```

In this situation, you can't use the ampersand symbol, because it would replace the entire matching pattern. The substring component provides the answer, allowing you to select just which part of the pattern to use as the replacement pattern.

This feature can be especially helpful when you need to insert text between two or more substring components. Here's a script that uses substring components to insert a comma in long numbers:

```
$ echo "1234567" | sed '{
> :start
> s/\(. *[0-9]\)\([0-9]\{3\}\)/\1,\2/
> t start
> }'
1,234,567
$
```

The script divides the matching pattern into two components:

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

This pattern looks for two substrings. The first substring is any number of characters, ending in a digit. The second substring is a series of three digits (see Chapter 20 for information about how to use braces in a regular expression). If this pattern is found in the text, the replacement text puts a comma between the two components, each identified by its component position. The script uses the `test` command to iterate through the number until all commas have been placed.

Placing sed Commands in Scripts

Now that you've seen the various parts of the `sed` editor, it's time to put them together and use them in your shell scripts. This section demonstrates some of the features that you should know about when using the `sed` editor in your bash shell scripts.

Using wrappers

You may have noticed that trying to implement a `sed` editor script can be cumbersome, especially if the script is long. Instead of having to retype the entire script each time you want to use it, you can place the `sed` editor command in a shell script *wrapper*. The wrapper acts as a go-between for the `sed` editor script and the command line.

Once inside the shell script, you can use normal shell variables and parameters with your `sed` editor scripts. Here's an example of using the command line parameter variable as the input to a `sed` script:

```
$ cat reverse.sh
#!/bin/bash
# Shell wrapper for sed editor script.
#           to reverse text file lines.
#
sed -n '{ 1!G ; h ; $p }' $1
#
$
```

The shell script called `reverse` uses the `sed` editor script to reverse text lines in a data stream. It uses the `$1` shell parameter to retrieve the first parameter from the command line, which should be the name of the file to reverse:

```
$ ./reverse.sh data2.txt
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
$
```

Now you can easily use the `sed` editor script on any file, without having to constantly retype the entire command line.

Redirecting sed output

By default, the `sed` editor outputs the results of the script to `STDOUT`. You can employ all the standard methods of redirecting the output of the `sed` editor in your shell scripts.

You can use dollar sign/parenthesis, `$()`, to redirect the output of your `sed` editor command to a variable for use later in the script. The following is an example of using the `sed` script to add commas to the result of a numeric computation:

```
$ cat fact.sh
#!/bin/bash
# Add commas to number in factorial answer
#
factorial=1
```

```

counter=1
number=$1
#
while [ $counter -le $number ]
do
    factorial=$(( $factorial * $counter )
    counter=$(( $counter + 1 )
done
#
result=$(echo $factorial | sed '{
:start
s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/
t start
}')
#
echo "The result is $result"
#
$
$ ./fact.sh 20
The result is 2,432,902,008,176,640,000
$

```

After you use the normal factorial calculation script, the result of that script is used as the input to the `sed` editor script, which adds commas. This value is then used in the `echo` statement to produce the result.

Creating sed Utilities

As you've seen in the short examples presented so far in this chapter, you can do lots of cool data-formatting things with the `sed` editor. This section shows a few handy well-known `sed` editor scripts for performing common data-handling functions.

Spacing with double lines

To start things off, look at a simple `sed` script to insert a blank line between lines in a text file:

```

$ sed 'G' data2.txt
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.

$

```

That was pretty simple! The key to this trick is the default value of the hold space. Remember that the `G` command simply appends the contents of the hold space to the current pattern space contents. When you start the `sed` editor, the hold space contains an empty line. By appending that to an existing line, you create a blank line after the existing line.

You may have noticed that this script also adds a blank line to the last line in the data stream, producing a blank line at the end of the file. If you want to get rid of this, you can use the negate symbol and the last line symbol to ensure that the script doesn't add the blank line to the last line of the data stream:

```
$ sed '$!G' data2.txt
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.
$
```

Now that looks a little better. As long as the line isn't the last line, the `G` command appends the contents of the hold space. When the `sed` editor gets to the last line, it skips the `G` command.

Spacing files that may have blanks

To take double spacing one step further, what if the text file already has a few blank lines, but you want to double space all the lines? If you use the previous script, you'll get some areas that have too many blank lines, because each existing blank line gets doubled:

```
$ cat data6.txt
This is line one.
This is line two.

This is line three.
This is line four.
$
$ sed '$!G' data6.txt
This is line one.

This is line two.

This is line three.

This is line four.
$
```

Now you have three blank lines where the original blank line was located. The solution to this problem is to first delete any blank lines from the data stream and then use the `G` command to insert new blank lines after all the lines. To delete existing blank lines, you just need to use the `d` command with a pattern that matches a blank line:

```
/^$/d
```

This pattern uses the start line tag (the caret) and the end line tag (the dollar sign). Adding this pattern to the script produces the desired results:

```
$ sed '/^$/d ; $!G' data6.txt
This is line one.

This is line two.

This is line three.

This is line four.
$
```

Perfect! It works just as expected.

Numbering lines in a file

Chapter 19 showed you how to use the equal sign to display the line numbers of lines in the data stream:

```
$ sed '=' data2.txt
1
This is the header line.
2
This is the first data line.
3
This is the second data line.
4
This is the last line.
$
```

This can be a little awkward to read, because the line number is on a line above the actual line in the data stream. A better solution is to place the line number on the same line as the text.

Now that you've seen how to combine lines using the `N` command, it shouldn't be too hard to utilize that information in the `sed` editor script. The trick to this utility, however, is that you can't combine the two commands in the same script.

After you have the output for the equal sign command, you can pipe the output to another `sed` editor script that uses the `N` command to combine the two lines. You also need to use the substitution command to replace the newline character with either a space or a tab character. Here's what the final solution looks like:

```
$ sed '=' data2.txt | sed 'N; s/\n/ /'
1 This is the header line.
2 This is the first data line.
3 This is the second data line.
4 This is the last line.
$
```

Now that looks much better. This is a great little utility to have around when working on programs where you need to see the line numbers used in error messages.

There are bash shell commands that can also add line numbers. However, they add some additional (and potentially unwanted spacing):

```
$ nl data2.txt
 1 This is the header line.
 2 This is the first data line.
 3 This is the second data line.
 4 This is the last line.
$
$ cat -n data2.txt
 1 This is the header line.
 2 This is the first data line.
 3 This is the second data line.
 4 This is the last line.
$
```

The `sed` editor script handles the output without any additional spacing.

Printing last lines

So far, you've seen how to use the `p` command to print all the lines in a data stream or just lines that match a specific pattern. What if you just need to work with the last few lines of a long listing, such as a log file?

The dollar sign represents the last line of a data stream, so it's easy to display just the last line:

```
$ sed -n '$p' data2.txt
This is the last line.
$
```

Now how can you use the dollar sign symbol to display a set number of lines at the end of the data stream? The answer is to create a *rolling window*.

A rolling window is a common way to examine blocks of text lines in the pattern space by combining them using the `N` command. The `N` command appends the next line of text to the text already in the pattern space. After you have a block of 10 text lines in the pattern space, you can check to see if you're at the end of the data stream using the dollar sign. If you're not at the end, continue adding more lines to the pattern space, while removing the original lines (remember the `D` command, which deletes the first line in the pattern space).

By looping through the `N` and `D` commands, you add new lines to the block of lines in the pattern space while removing old lines. The `branch` command is the perfect fit for the loop. To end the loop, just identify the last line and use the `q` command to quit.

Here's what the final `sed` editor script looks like:

```
$ cat data7.txt
This is line 1.
This is line 2.
This is line 3.
This is line 4.
This is line 5.
This is line 6.
This is line 7.
This is line 8.
This is line 9.
This is line 10.
This is line 11.
This is line 12.
This is line 13.
This is line 14.
This is line 15.
$
$ sed '{
> :start
> $q ; N ; 11,$D
> b start
> }' data7.txt
This is line 6.
This is line 7.
This is line 8.
This is line 9.
This is line 10.
This is line 11.
This is line 12.
This is line 13.
This is line 14.
This is line 15.
$
```

The script first checks whether the line is the last line in the data stream. If it is, the `quit` command stops the loop. The `N` command appends the next line to the current line in the pattern space. The `11, $D` command deletes the first line in the pattern space if the current line is after line 10. This creates the sliding window effect in the pattern space. Thus, the `sed` program script displays only the last 10 lines of the `data7.txt` file.

Deleting lines

Another useful utility for the `sed` editor is to remove unwanted blank lines in a data stream. It's easy to remove all the blank lines from a data stream, but it takes a little ingenuity to selectively remove blank lines. This section shows you a couple of quick `sed` editor scripts that you can use to help remove unwanted blank lines from your data.

Deleting consecutive blank lines

It can be a nuisance when extra blank lines crop up in data files. Often you have a data file that contains blank lines, but sometimes a data line is missing and produces too many blank lines (as you saw in the double-spacing example earlier).

The easiest way to remove consecutive blank lines is to check the data stream using a range address. Chapter 19 showed you how to use ranges in addresses, including how to incorporate patterns in the address range. The `sed` editor executes the command for all lines that match within the specified address range.

The key to removing consecutive blank lines is to create an address range that includes a non-blank line and a blank line. If the `sed` editor comes across this range, it shouldn't delete the line. However, for lines that don't match that range (two or more blank lines in a row), it should delete the lines.

Here's the script to do this:

```
/./,/^$/!d
```

The range is `/./` to `/^$/`. The start address in the range matches any line that contains at least one character. The end address in the range matches a blank line. Lines within this range aren't deleted.

Here's the script in action:

```
$ cat data8.txt
This is line one.

This is line two.

This is line three.

This is line four.
```



```
$
$ sed '/./,/^$/!d' data8.txt
This is line one.

This is line two.

This is line three.

This is line four.
$
```

No matter how many blank lines appear between lines of data in the file, the output places only one blank line between the lines.

Deleting leading blank lines

It is also a nuisance when data files contain multiple blank lines at the start of the file. Often when you are trying to import data from a text file into a database, the blank lines create null entries, throwing off any calculations using the data.

Removing blank lines from the top of a data stream is not a difficult task. Here's the script that accomplishes that function:

```
/./,$!d
```

The script uses an address range to determine what lines are deleted. The range starts with a line that contains a character and continues to the end of the data stream. Any line within this range is not deleted from the output. This means that any lines before the first line that contain a character are deleted.

Look at this simple script in action:

```
$ cat data9.txt

This is line one.

This is line two.
$
$ sed '/./,$!d' data9.txt
This is line one.

This is line two.
$
```

The test file contains two blank lines before the data lines. The script successfully removes both of the leading blank lines, while keeping the blank line within the data intact.

Deleting trailing blank lines

Unfortunately, deleting trailing blank lines is not as simple as deleting leading blank lines. Just like printing the end of a data stream, deleting blank lines at the end of a data stream requires a little ingenuity and looping.

Before we start the discussion, let's see what the script looks like:

```
sed '{
:start
/^\\n*$/{$d; N; b start }
}'
```

This may look a little odd to you at first. Notice that there are braces within the normal script braces. This allows you to group commands together within the overall command script. The group of commands applies to the specified address pattern. The address pattern matches any line that contains only a newline character. When one is found, if it's the last line, the delete command deletes it. If it's not the last line, the N command appends the next line to it, and the branch command loops to the beginning to start over.

Here's the script in action:

```
$ cat data10.txt
This is the first line.
This is the second line.

$ sed '{
> :start
> /^\\n*$/{$d ; N ; b start }
> }' data10.txt
This is the first line.
This is the second line.
$
```

The script successfully removed the blank lines from the end of the text file.

Removing HTML tags

These days, it's not uncommon to download text from a website to save or use as data in an application. Sometimes, however, when you download text from the website, you also get the HTML tags used to format the data. This can be a problem when all you want to see is the data.

A standard HTML web page contains several different types of HTML tags, identifying formatting features required to properly display the page information. Here's a sample of what an HTML file looks like:

```
$ cat data11.txt
<html>
<head>
<title>This is the page title</title>
</head>
<body>
<p>
This is the <b>first</b> line in the Web page.
This should provide some <i>useful</i>
information to use in our sed script.
</body>
</html>
$
```

HTML tags are identified by the less-than and greater-than symbols. Most HTML tags come in pairs. One tag starts the formatting process (for example, `` for bolding), and another tag stops the formatting process (for example, `` to turn off bolding).

Removing HTML tags creates a problem, however, if you're not careful. At first glance, you'd think that the way to remove HTML tags would be to just look for a text string that starts with a less-than symbol (`<`), ends with a greater-than symbol (`>`), and has data in between the symbols:

```
s/<.*>/g
```

Unfortunately, this command has some unintended consequences:

```
$ sed 's/<.*>/g' data11.txt
```

```
This is the  line in the Web page.
This should provide some
information to use in our sed script.
```

```
$
```

Notice that the title text is missing, along with the text that was bolded and italicized. The `sed` editor literally interpreted the script to mean any text between the less-than and greater-than sign, including other less-than and greater-than signs! Each time the text was enclosed in HTML tags (such as `first`), the `sed` script removed the entire text.

The solution to this problem is to have the `sed` editor ignore any embedded greater-than signs between the original tags. To do that, you can create a character class that negates the greater-than sign. This changes the script to:

```
s/<[^>]*>//g
```

This script now works properly, displaying the data you need to see from the web page HTML code:

```
$ sed 's/<[^>]*>//g' data11.txt
```

```
This is the page title
```

```
This is the first line in the Web page.  
This should provide some useful  
information to use in our sed script.
```

```
$
```

That's a little better. To clean things up some, you can add a `delete` command to get rid of those pesky blank lines:

```
$ sed 's/<[^>]*>//g ; /^$/d' data11.txt  
This is the page title  
This is the first line in the Web page.  
This should provide some useful  
information to use in our sed script.  
$
```

Now that's much more compact; there's only the data you need to see.

Summary

The `sed` editor provides some advanced features that allow you to work with text patterns across multiple lines. This chapter showed you how to use the `next` command to retrieve the next line in a data stream and place it in the pattern space. Once in the pattern space, you can perform complex substitution commands to replace phrases that span more than one line of text.

The multiline `delete` command allows you to remove the first line when the pattern space contains two or more lines. This is a convenient way to iterate through multiple lines in the data stream. Similarly, the multiline `print` command allows you to print just the first line when the pattern space contains two or more lines of text. The combination of the multiline commands allows you to iterate through the data stream and create a multiline substitution system.

Next, we covered the hold space. The hold space allows you to set aside a line of text while processing more lines of text. You can recall the contents of the hold space at any time and either replace the text in the pattern space or append the contents of the hold space to the text in the pattern space. Using the hold space allows you to sort through data streams, reversing the order of text lines as they appear in the data.

Next we reviewed the various `sed` editor flow control commands. The `branch` command provides a way for you to alter the normal flow of `sed` editor commands in the script, creating loops or skipping commands under certain conditions. The `test` command provides an `if-then` type of statement for your `sed` editor command scripts. The `test` command branches only if a prior substitution command succeeds in replacing text in a line.

The chapter concluded with a discussion of how to use `sed` scripts in your shell scripts. A common technique for large `sed` scripts is to place the script in a shell wrapper. You can use command line parameter variables within the `sed` script to pass shell command line values. This creates an easy way to utilize your `sed` editor scripts directly from the command line, or even from other shell scripts.

The next chapter digs deeper into the `gawk` world. The `gawk` program supports many features of higher-level programming languages. You can create some pretty involved data manipulation and reporting programs just by using `gawk`. The chapter describes the various programming features and demonstrates how to use them to generate your own fancy reports from simple data.