

SED EXPLAINED

DATA STRUCTURES AND OPERATORS

Part—2

Continuing from last month's article on the subject, we now proceed to Sed data structures and operators.

Sed has a few powerful data structures and operators, which enable you to perform complex text-manipulation operations. Let's look at some of them.

Pattern space (p and P)

Pattern space is a memory area where sed copies a matched line or text first, before performing operations on it. Text-manipulation operations are (always) performed (only) on the contents of this pattern space. By default, the pattern space consists of a single line, copied as the line is read in. However, you can add multiple lines into the pattern space using line-reader operators. The 'p' operator is used to print the entire pattern space. P is an operator like p, but it prints the content of the pattern space up to the first \n character that is found. To make this clear: let's have 'line1\nline2' in the pattern space; then p prints both lines, but P prints only the first line.

Line readers n and N

The n operator is used to explicitly read a line. When used in a sed script, it reads a new line from the input, and replaces the current line in the pattern space. Let's try an example:

```
$ seq 10 | sed -n 'n;p'
```

2
4
6
8
10

When sed reads the first line, 'n' explicitly replaces the contents of the pattern space with the second line, before it is printed with p. Similarly, when sed reads the third line, 'n' explicitly reads in the fourth line, which is printed.

The second line-reader operator, N, appends the newly read line to the pattern space. Hence, the pattern space will contain *current_line\nread_line*. This is very useful to perform operations on multiple lines. For example:

```
$ seq 10 | sed -n 'N; s/\(.*\)\n\(.*\)/\2\n\1/; p'
```

2
1
4
3
6
5
8
7
10
9

Here, on each execution for a line, 'N' is executed. Hence, in the first execution, the pattern space will be 1\n2; in the second execution, it will be 3\n4 and so on. The substitution operation is used to change the order of lines. The first and second lines are swapped with a \n in between. The first \(.*) matches the first line, while the second \(.*) after a \n matches the second line. In the replacement part, \2\n\1 is used to change the order.

Line deletion (d and D)

The `'d'` operator is used to delete all data in the pattern space. For example:

```
$ seq 4 | sed '3 d'
1
2
4
```

The `D` operator also performs a deletion operation, but it removes the data in the pattern space until the first `\n` character is found. For instance, let's suppose we have the data `'line1\nline2\nline3'` in the pattern space; when `D` is applied, the content of pattern space becomes `'line2\nline3'`.

Any statement that comes after a `d` or `D` in the `sed` script will not be executed; it skips the following commands, and starts the next iteration.

Hold buffer (h and H) and exchange operation (x)

The hold buffer is a special storage location, like the pattern space. You can copy the contents of the pattern space to this storage area. When `'h'` is used, it replaces the current content of the hold buffer with a copy of the pattern space. When `'H'` is used, a copy of the pattern space will be appended to the contents of the hold buffer.

When you need to move the contents of the hold buffer back to the pattern space to work on it, use the exchange (`'x'`) operator, which swaps the contents of the hold buffer and the pattern space. If you swap again, the pattern space will contain the original data. Let's look at a simple example:

File lines.txt:

```
line 1
line 2
line 3
line 4
line 5
```

Now, you need to store the first line in the hold buffer, and not print it out. When `sed` reaches the end-of-file, it should print the first line instead of the last line:

```
$ sed -n '1{h; n} ;$x;p' lines.txt
line 2
line 3
line 4
line 1
```

In the above script, `1{h; n}` says that when the line number is 1, execute the group `{h;n}`. The group uses the copy-to-hold-buffer operator `'h'` and the read-next-line operator `'n'`. `$x` says that when `sed` reaches the last line (`$`), exchange the

hold buffer and the pattern space contents, so that the pattern space will contain the (earlier-stored) first line. The operator `'p'` is used to print the pattern space for every execution.

Get operator (g and G)

In the case of the hold buffer, when you needed to copy its contents to the pattern space, you used the exchange (`'x'`) operator to swap the contents of the hold buffer and the pattern space. Additionally, the get operator (`'g'`) can be used to copy the hold buffer contents to the pattern space, deleting the contents of the pattern space while doing so. The `'G'` operator appends the contents of the hold buffer to the pattern space, instead of deleting the pattern space's contents and replacing it with the hold buffer's content. Let's use the file lines.txt to illustrate the example:

```
$ cat lines.txt | sed -n '$p; h;n;G;p;'
line 2
line 1
line 4
line 3
line 5
```

In the above script, swap two consecutive lines, using `G` and `h`.

Operator	Effect
<code>\$p</code>	Prints the pattern space when the last line is reached.
<code>H</code>	Copies the pattern space to the hold buffer.
<code>N</code>	Reads the next line into the pattern space.
<code>G</code>	Appends the hold buffer contents to the pattern space.
<code>P</code>	Prints the current pattern space.

The above sequence is executed iteratively until end-of-file is reached.

Flow control with labels; and testing with 't'

You would never expect `sed` to support looping of statements. Yes, it has a branching command, `'b'`, which can jump to a label specified as `:labelname`. Branching can be performed by using `'b labelname'`. If the label is not specified, it will branch to the end of the `sed` script. There is an additional operator called test (`'t'`), which is used for conditional jumps. Otherwise, **identical to 'b', 't' is different in that the branching will occur only if the last statement executed successfully??????**. For branching with `b`, you may need to explicitly quit from script execution with the `'q'` operator, since otherwise, it never ends the execution loop. Let's go through an example to test whether a string is in the format `A^nB^n`:

```
$ echo AAABBB | sed ':again ; s/A(.*\)B/\1/; t again ; s/^$/
TRUE/; t ; s././FALSE/'
TRUE
$ echo AAABBBB | sed ':again ; s/A(.*\)B/\1/; t again ; s/^$/
TRUE/; t ; s././FALSE/'
FALSE
```

Let's go through the above sed script, part by part.

<code>s/A(.*\)B/\1/</code>	Removes the outer A and B. On the first operation, AAABBB will become AABB.
<code>:again</code>	A label that is used as the target of the t jump operator.
<code>t again</code>	If the substitution operation <code>s/A(.*\)B/\1/</code> succeeds, it jumps to the label again, and executes from there.
<code>s/^\$/TRUE/; t ;</code>	If the current pattern space is empty, set the pattern space to TRUE. The t ; says that if the substitution operation succeeds, jump to the end of the script.
<code>s././FALSE/'</code>	If the previous 't' test fails, replace the pattern space with FALSE.

A few cool scripts

We have gone through the essential components and features of *sed*. Now let us go through a few nice examples of sed scripts.

Emulating the tac command

The *tac* command is used to print input lines in the reverse order. You can perform the reversal of lines by pushing the lines in a stack data-structure, and when it reaches the end, start popping and printing lines, one by one. Using *sed*, we can manipulate stack operations with *h* and *G*.

```
sed -n '1!G; h ; ${ x; p }' filename
```

<code>1!G</code>	Execute <i>G</i> when it is not the first line.
<code>h</code>	Copy the contents of the pattern buffer to the hold buffer.
<code>\${ x; p }</code>	When the last line is reached, exchange the hold buffer contents to the pattern buffer, and print them out.

When *G* is applied, it appends the hold buffer contents to the pattern space. It is again pushed to the hold buffer, and the next time, again appended with the pattern space. Thus, a stack push operation is implemented.

Checking for palindrome

```
$ echo malayalam | sed ':again ; s/\(.\)\(.*\)\1/\2/; t again ;
```

```
s/^\.?$/TRUE/; t ; s././FALSE/'
TRUE
```

This palindrome-check script is a slight modification of the example given in the flow controls with the labels section. Try to dissect it yourself.

Reversing words in a line

```
$ echo this is a line | sed '/\n!/G; s/\(w*\) \(.*\n)/&2
\1/; //D; s/\n//;'
line a is this
```


In this script, we again use a type of stack push-and-pop operations. Let us dissect the script. Initially, if the contents of the pattern space do not contain `\n`, it appends the contents of the hold buffer to the pattern space. The hold buffer initially contains the `\n` character. Then the current pattern space is replaced with the current pattern space appended with the first word, and the rest of the text in the reversed order. Subsequently `//` matches the current pattern space if not empty, and deletes up to the first occurrence of `\n`, and the next iteration begins. At the end, an additional `\n` is replaced, and printed to get reversed words in the line.

Emulating the tail command

The *tail* command (by default) prints the last 10 lines of a text file. Let us look at how to keep 10 lines in the memory of *sed* as it processes lines:

```
$ seq 100 | sed '$q; :start; N ; 11,$D ; b start'
```

In this script, *N* is used to append newly read lines to the pattern space. It is iterated in a loop using a branch with the label *start*. From lines 11 to the last line, it deletes a line at the start of the pattern space, up to the first `\n`, so that the total lines in the pattern space will always be 10. Hence, when *sed* reaches the last line, it quits (*\$q*) and prints the pattern space.

We have covered most of the basic use of *sed*, with the text processing capabilities of the stream editor. For more information, there is a *sed* one-liner collection, written by the community at <http://sed.sourceforge.net/sed1line.txt>. Happy hacking till we meet again! 

By: Sarath Lakshman

The author is a hacktivist of Free and Open Source Software from Kerala. He loves working on the GNU/Linux environment, and contributes to the Pardus Linux distro project. He has recently authored the Linux Shell Scripting Cookbook, which gives insights on shell scripting through 119 recipes. He can be reached via his website, <http://www.sarathlakshman.com>.