LEARN Sed

stream editor

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

This tutorial takes you through all about Stream EDitor (SED), one of the most prominent text-processing utilities on GNU/Linux. Similar to many other GNU/Linux utilities, it is stream-oriented and uses simple programming language. It is capable of solving complex text processing tasks with few lines of code. This easy, yet powerful utility makes GNU/Linux more interesting.

Audience

If you are a software developer, system administrator, or a GNU/Linux loving person, then this tutorial is for you.

Prerequisites

You must have basic understanding of GNU/Linux operating system and shell scripting.

Copyright & Disclaimer

Table of Contents

1. OVERVIEW

The acronym SED stands for **Stream EDitor**. It is a simple yet powerful utility that parses the text and transforms it seamlessly. SED was developed during 1973–74 by Lee E. McMahon of Bell Labs. Today, it runs on all major operating systems.

McMahon wrote a general-purpose line-oriented editor, which eventually became SED. SED borrowed syntax and many useful features from *ed* editor. Since its beginning, it has support for regular expressions. SED accepts inputs from files as well as pipes. Additionally, it can also accept inputs from standard input streams.

SED is written and maintained by the Free Software Foundation (FSF) and it is distributed by GNU/Linux. Hence it is often referred to as **GNU SED**. To a novice user, the syntax of SED may look cryptic. However, once you get familiar with its syntax, you can solve many complex tasks with a few lines of SED script. This is the beauty of SED.

Typical Uses of SED

SED can be used in many different ways, such as:

- Text substitution,
- Selective printing of text files,
- In-a-place editing of text files,
- Non-interactive editing of text files, and many more.

2. ENVIRONMENT

This chapter describes how to set up the SED environment on your GNU/Linux system.

Installation Using Package Manager

Generally, SED is available by default on most GNU/Linux distributions. Use **which** command to identify whether it is present on your system or not. If not, then install SED on Debian based GNU/Linux using **apt** package manager as follows:

```
[jerry]$ sudo apt-get install sed
```

After installation, ensure that SED is accessible via command line.

```
[jerry]$ sed --version
```

On executing the above code, you get the following result:

```
sed (GNU sed) 4.2.2
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Jay Fenlason, Tom Lord, Ken Pizzini,
and Paolo Bonzini.
GNU sed home page: <http://www.gnu.org/software/sed/>.
General help using GNU software: <http://www.gnu.org/gethelp/>.
E-mail bug reports to: <bug-sed@gnu.org>.
Be sure to include the word "sed" somewhere in the "Subject:" field.
```

Similarly, to install SED on RPM based GNU/Linux, use **yum** package manager as follows:

```
[root]# yum -y install sed
```

After installation, ensure that SED is accessible via command line.

```
[root]# sed --version
```

On executing the above code, you get the following result:

```
GNU sed version 4.2.1
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE,
to the extent permitted by law.

GNU sed home page: <http://www.gnu.org/software/sed/>.
General help using GNU software: <http://www.gnu.org/gethelp/>.
E-mail bug reports to: <bug-gnu-utils@gnu.org>.
Be sure to include the word "sed" somewhere in the "Subject:" field.
```

Installation from Source Code

As GNU SED is a part of the GNU project, its source code is available for free download. We have already seen how to install SED using package manager. Let us now understand how to install SED from its source code.

The following installation is applicable to any GNU/Linux software, and for most other freely-available programs as well. Here are the installation steps:

- Download the source code from an authentic place. The command-line utility**wget** serves this purpose.

  ```
  [jerry]$ wget ftp://ftp.gnu.org/gnu/sed/sed-4.2.2.tar.bz2
  ```

- Decompress and extract the downloaded source code.

  ```
  [jerry]$ tar xvf sed-4.2.2.tar.bz2
  ```

- Change into the directory and run *configure*.

  ```
  [jerry]$ ./configure
  ```

- Upon successful completion, the **configure** generates Makefile. To compile the source code, issue a **make** command.

  ```
  [jerry]$ make
  ```

- You can run the test suite to ensure the build is clean. This is an optional step.

```
[jerry]$ make check
```

- Finally, install the SED utility. Make sure you have superuser privileges.

```
[jerry]$ sudo make install
```

That is it! You have successfully compiled and installed SED. Verify it by executing the **sed**command as follows:

```
[jerry]$ sed --version
```

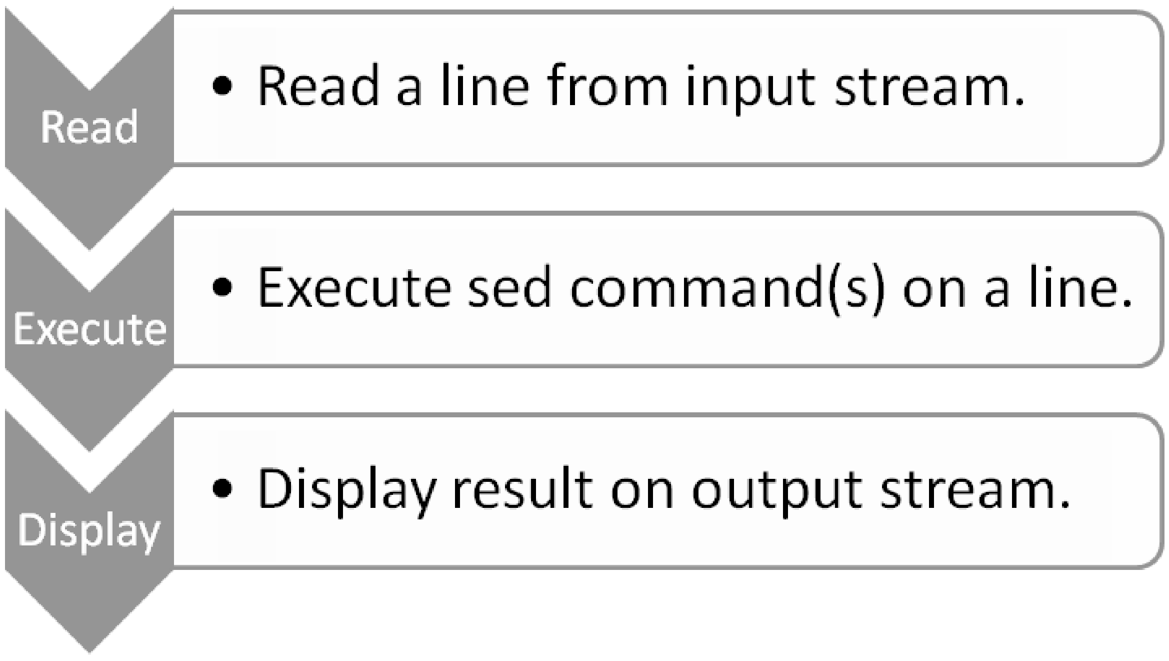
On executing the above code, you get the following result:

```
sed (GNU sed) 4.2.2
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Jay Fenlason, Tom Lord, Ken Pizzini,
and Paolo Bonzini.
GNU sed home page: <http://www.gnu.org/software/sed/>.
General help using GNU software: <http://www.gnu.org/gethelp/>.
E-mail bug reports to: <bug-sed@gnu.org>.
Be sure to include the word "sed" somewhere in the "Subject:" field.
```

3. WORKFLOW

In this chapter, we will explore how SED exactly works. To become an expert SED user, one needs to know its internals. SED follows a simple workflow: Read, Execute, and Display. The following diagram depicts the workflow.


Read
• Read a line from input stream.
Execute
• Execute sed command(s) on a line.
Display
• Display result on output stream.

- **Read**: SED reads a line from the input stream (file, pipe, or stdin) and stores it in its internal buffer called **pattern buffer**.
- **Execute**: All SED commands are applied sequentially on the pattern buffer. By default, SED commands are applied on all lines (globally) unless line addressing is specified.
- **Display**: Send the (modified) contents to the output stream. After sending the data, the pattern buffer will be empty.
- The above process repeats until the file is exhausted.

Points to Note

- Pattern buffer is a private, in-memory, volatile storage area used by the SED.
- By default, all SED commands are applied on the pattern buffer, hence the input file remains unchanged. GNU SED provides a way to modify the input file in-a-place. We will explore about it in later sections.

- There is another memory area called **hold buffer** which is also private, in-memory, volatile storage area. Data can be stored in a hold buffer for later retrieval. At the end of each cycle, SED removes the contents of the pattern buffer but the contents of the hold buffer remains persistent between SED cycles. However SED commands cannot be directly executed on hold buffer, hence SED allows data movement between the hold buffer and the pattern buffer.
- Initially both pattern and hold buffers are empty.
- If no input files are provided, then SED accepts input from the standard input stream (stdin).
- If address range is not provided by default, then SED operates on each line.

Examples

Let us create a text file **quote.txt** to contain a quote of the famous author Paulo Coelho.

```
[jerry]$ vi quote.txt

There is only one thing that makes a dream impossible to achieve: the fear of failure.

         - Paulo Coelho, The Alchemist
```

To understand the workflow of SED, let us display the contents of the file quote.txt using SED. This example simulates the **cat** command.

```
[jerry]$ sed '' quote.txt
```

When the above code is executed, it will produce the following result.

```
There is only one thing that makes a dream impossible to achieve: the fear of failure.
```

In the above example, quote.txt is the input file name and before that there is a pair of single quote that implies the SED command. Let us demystify this operation.

First SED reads a line from the input file quote.txt and stores it in its pattern buffer. Then it applies SED commands on the pattern buffer. In our case, no SED commands are there, hence no operation is performed on the pattern buffer. Finally it deletes and prints the contents of the pattern buffer on the standard output. Isn't it simple?

In the following example, SED accepts input from the standard input stream.

```
[jerry]$ sed '' <press enter>
```

When the above code is executed, it will produce the following result.

```
There is only one thing that makes a dream impossible to achieve: the fear of failure.
There is only one thing that makes a dream impossible to achieve: the fear of failure.
```

Here, the first line is entered through keyboard and the second is the output generated by SED. To exit from the SED session, press ctrl-D (^D).

4. BASIC SYNTAX

This chapter introduces the basic commands that SED supports and their command-line syntax. SED can be invoked in the following two forms:

```
sed [-n] [-e] 'command(s)' files

sed [-n] -f scriptfile files
```

The first form allows to specify the commands in-line and they are enclosed within single quotes. The later allows to specify a script file that contains SED commands. However, we can use both forms together multiple times. SED provides various command-line options to control its behavior.

Let us see how we can specify multiple SED commands. SED provides the **delete** command to delete certain lines. Let us delete the 1st, 2nd, and 5th lines. For the time being, ignore all the details of the delete command. We will discuss more about the delete command later.

First, display the file contents using the **cat** command.

```
[jerry]$ cat books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Now instruct SED to remove only certain lines. Here, to delete three lines, we have specified three separate commands with -e option.

```
[jerry]$ sed -e '1d' -e '2d' -e '5d' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
6) A Game of Thrones, George R. R. Martin, 864
```

Additionally, we can write multiple SED commands in a text file and provide the text file as an argument to SED. SED can apply each command on the pattern buffer. The following example illustrates the second form of SED.

First, create a text file containing SED commands. For easy understanding, let us use the same SED commands.

```
[jerry]$ echo -e "1d\n2d\n5d" > commands.txt

[jerry]$ cat commands.txt
```

On executing the above code, you get the following result:

```
1d
2d
5d
```

Now instruct the SED to read commands from the text file. Here, we achieve the same result as shown in the above example.

```
[jerry]$ sed -f commands.txt books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
6) A Game of Thrones,George R. R. Martin, 864
```

Standard Options

SED supports the following standard options:

- -n: Default printing of pattern buffer. For example, the following SED command does not show any output:

  ```
  [jerry]$ sed -n '' quote.txt
  ```

- -e <cmd> : Next argument is an editing command. Here, angular brackets imply mandatory parameter. By using this option, we can specify multiple commands. Let us print each line twice:

  ```
  [jerry]$ sed -e '' -e 'p' quote.txt
  ```

On executing the above code, you get the following result:

```
There is only one thing that makes a dream impossible to achieve: the fear of
failure.
There is only one thing that makes a dream impossible to achieve: the fear of
failure.
  - Paulo Coelho, The Alchemist
  - Paulo Coelho, The Alchemist
```

- -f <filename> : Next argument is a file containing editing commands. The angular brackets imply mandatory parameter. In the following example, we specify print command through file:

```
[jerry]$ echo "p" > commands
[jerry]$ sed -n -f commands quote.txt
```

On executing the above code, you get the following result:

```
There is only one thing that makes a dream impossible to achieve: the fear of
failure.
  - Paulo Coelho, The Alchemist
```

GNU Specific Options

Let us quickly go through the GNU specific SED options. Note that these options are GNU specific; and may not be supported by other variants of the SED. In later sections, we will discuss these options in more details.

- -n, --quiet, --silent: Same as standard -n option.
- -e script, --expression=script: Same as standard -e option.
- -f script-file, --file=script-file: Same as standard -f option.
- --follow-symlinks: If this option is provided, the SED follows symbolic links while editing files in place.
- -i[SUFFIX], --in-place[=SUFFIX]: This option is used to edit file in place. If suffix is provided, it takes a backup of the original file, otherwise it overwrites the original file.
- -l N, --line-lenght=N: This option sets the line length for l command to N characters.
- --posix: This option disables all GNU extensions.
- -r, --regexp-extended: This option allows to use extended regular expressions rather than basic regular expressions.

- -u, --unbuffered: When this option is provided, the SED loads minimal amount of data from the input files and flushes the output buffers more often. It is useful for editing the output of "tail -f" when you do not want to wait for the output.
- -z, --null-data: By default, the SED separates each line by a new-line character. If NULL-data option is provided, it separates the lines by NULL characters.

5. LOOPS

Like other programming languages, SED too provides a looping and branching facility to control the flow of execution. In this chapter, we are going to explore more about how to use loops and branches in SED.

A loop in SED works similar to a **goto** statement. SED can jump to the line marked by the label and continue executing the remaining commands. In SED, we can define a **label** as follows:

```
:label

:start

:end

:up
```

In the above example, a name after colon(:) implies the label name.

To jump to a specific label, we can use the **b** command followed by the label name. If the label name is omitted, then the SED jumps to the end of the SED file.

Let us write a simple SED script to understand the loops and branches. In our books.txt file, there are several entries of book titles and their authors. The following example combines a book title and its author name in one line separated by a comma. Then it searches for the pattern "Paulo". If the pattern matches, it prints a hyphen(-) in front of the line, otherwise it jumps to the **Print** label which prints the line.

```
[jerry]$ sed -n '

h;n;H;x

s/\n/, /

/Paulo/!b Print

s/^/- /

:Print

p' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords, George R. R. Martin

The Two Towers, J. R. R. Tolkien

- The Alchemist, Paulo Coelho

The Fellowship of the Ring, J. R. R. Tolkien

- The Pilgrimage, Paulo Coelho
```

```
A Game of Thrones, George R. R. Martin
```

At first glance, the above script may look cryptic. Let us demystify this.

- The first two commands are self-explanatory **h;n;H;x** and **s/\n/, /** combine the book title and its author separated by a comma(,).
- The third command jumps to the label **Print** only when the pattern does not match, otherwise substitution is performed by the fourth command.
- **:Print** is just a label name and as you already know, **p** is the print command.

To improve readability, each SED command is placed on a separate line. However, one can choose to place all the commands in one line as follows:

```
[jerry]$ sed -n 'h;n;H;x;s/\n/, /;/Paulo/!b Print; s/^/- /; :Print;p' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords, George R. R. Martin
The Two Towers, J. R. R. Tolkien
- The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
- The Pilgrimage, Paulo Coelho
A Game of Thrones, George R. R. Martin
```

6. BRANCHES

Branches can be created using the **t** command. The *t* command jumps to the label only if the previous substitute command was successful. Let us take the same example as in the previous chapter, but instead of printing a single hyphen(-), now we print four hyphens. The following example illustrates the usage of the **t** command.

```
[jerry]$ sed -n '
h;n;H;x
s/\n/, /
:Loop
/Paulo/s/^/-/
/----/!t Loop
p' books.txt
```

When the above code is executed, it will produce the following result.

```
A Storm of Swords, George R. R. Martin
The Two Towers, J. R. R. Tolkien
----The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
----The Pilgrimage, Paulo Coelho
A Game of Thrones, George R. R. Martin
```

In the above example, the first two commands are self-explanatory. The third command defines a label **Loop**. The fourth command prepends hyphen(-) if the line contains the string "Paulo" and the **t** command repeats the procedure until there are four hyphens at the beginning of the line.

To improve readability, each SED command is written on a separate line. Otherwise, we can write a one-liner SED as follows:

```
[jerry]$ sed -n 'h;n;H;x; s/\n/, /; :Loop;/Paulo/s/^/-/; /----/!t Loop; p' books.txt
```

When the above code is executed, it will produce the following result.

```
A Storm of Swords, George R. R. Martin
The Two Towers, J. R. R. Tolkien
----The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
----The Pilgrimage, Paulo Coelho
A Game of Thrones, George R. R. Martin
```

7. PATTERN BUFFER

One of the basic operations we perform on any file is display its contents. For this purpose, we can use the **print** command which prints the contents of the pattern buffer. So let us learn more about the pattern buffer.

First create a file containing the line number, the name of the book, its author, and the number of pages. In this tutorial, we will be using this file. You can use any text file according to your convenience. Our text file will look like this:

```
[jerry]$ vi books.txt
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho,288
6) A Game of Thrones, George R. R. Martin, 864
```

Now, let us print the file contents.

```
[jerry]$ sed 'p' books.txt
```

When the above code is executed, it will produce the following result.

```
1) A Storm of Swords, George R. R. Martin, 1216
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
6) A Game of Thrones, George R. R. Martin, 864
```

You might wonder why each line is being displayed twice. Let us find out.

Do you remember the workflow of SED? By default, SED prints the contents of the pattern buffer. In addition, we have included a print command explicitly in our command section. Hence each line is printed twice. But don't worry. SED has the **-n** option to suppress the default printing of the pattern buffer. The following command illustrates that.

```
[jerry]$ sed -n 'p' books.txt
```

When the above code is executed, it will produce the following result.

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Congratulations! we got the expected result. By default, SED operates on all lines. But we can force SED to operate only on certain lines. For instance, in the example below, SED only operates on the 3rd line. In this example, we have specified an address range before the SED command.

```
[jerry]$ sed -n '3p' books.txt
```

When the above code is executed, it will produce the following result.

```
3) The Alchemist, Paulo Coelho, 197
```

Additionally, we can also instruct SED to print only certain lines. For instance, the following code prints all the lines from 2 to 5. Here we have used the comma(,) operator to specify the address range.

```
[jerry]$ sed -n '2,5 p' books.txt
```

When the above code is executed, it will produce the following result.

```
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
```

There is also a special character Dollar($) which represents the last line of the file. So let us print the last line of the file.

```
[jerry]$ sed -n '$ p' books.txt
```

When the above code is executed, it will produce the following result.

```
6) A Game of Thrones, George R. R. Martin, 864
```

However we can also use Dollar($) character to specify address range. Below example prints through line 3 to last line.

```
[jerry]$ sed -n '3,$ p' books.txt
```

When the above code is executed, it will produce the following result.

```
3) The Alchemist, Paulo Coelho, 197 4) The Fellowship of the Ring, J. R. R. Tolkien, 432 5)
The Pilgrimage, Paulo Coelho, 288 6) A Game of Thrones, George R. R. Martin, 864
```

We learnt how to specify an address range using the comma(,) operator. SED supports two more operators that can be used to specify address range. First is the plus(+) operator and it can be used with the comma(,) operator. For instance **M, +n** will print the next **n** lines starting from line number **M**. Sounds confusing? Let us check it with a simple example. The following example prints the next 4 lines starting from line number 2.

```
[jerry]$ sed -n '2,+4 p' books.txt
```

When the above code is executed, it will produce the following result.

```
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Optionally, we can also specify address range using the tilde(~) operator. It uses **M~n** form. It indicates that SED should start at line number M and process every n(th) line. For instance, **50~5** matches line number 50, 55, 60, 65, and so on. Let us print only odd lines from the file.

```
[jerry]$ sed -n '1~2 p' books.txt
```

When the above code is executed, it will produce the following result.

```
1) A Storm of Swords, George R. R. Martin, 1216
3) The Alchemist, Paulo Coelho, 197
5) The Pilgrimage, Paulo Coelho, 288
```

The following code prints only even lines from the file.

```
[jerry]$ sed -n '2~2 p' books.txt
```

When the above code is executed, it will produce the following result.

```
2) The Two Towers, J. R. R. Tolkien, 352
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
6) A Game of Thrones, George R. R. Martin, 864
```

8. PATTERN RANGE

In the previous chapter, we learnt how SED handles an address range. This chapter covers how SED takes care of a pattern range. A pattern range can be a simple text or a complex regular expression. Let us take an example. The following example prints all the books of the author *Paulo Coelho*.

```
[jerry]$ sed -n '/Paulo/ p' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, Paulo Coelho, 197
5) The Pilgrimage, Paulo Coelho, 288
```

In the above example, the SED operates on each line and prints only those lines that match the string *Paulo*.

We can also combine a pattern range with an address range. The following example prints lines starting with the first match of *Alchemist* until the fifth line.

```
[jerry]$ sed -n '/Alchemist/, 5 p' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
```

We can use the Dollar($) character to print all the lines after finding the first occurrence of the pattern. The following example finds the first occurrence of the pattern *The* and immediately prints the remaining lines from the file.

```
[jerry]$ sed -n '/The/,$ p' books.txt
```

On executing the above code, you get the following result:

```
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
```

```
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

We can also specify more than one pattern ranges using the comma(,) operator. The following example prints all the lines that exist between the patterns *Two* and *Pilgrimage*.

```
[jerry]$ sed -n '/Two/, /Pilgrimage/ p' books.txt
```

On executing the above code, you get the following result:

```
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
```

Additionally, we can use the plus(+) operator within a pattern range. The following example finds the first occurrence of the pattern *Two* and prints the next 4 lines after that.

```
[jerry]$ sed -n '/Two/, +4 p' books.txt
```

On executing the above code, you get the following result:

```
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

We have supplied here only a few examples to get you acquainted with SED. You can always get to know more by trying a few examples on your own.

9. BASIC COMMANDS

This chapter describes several useful SED commands.

Delete Command

SED provides various commands to manipulate text. Let us first explore about the **delete**command. Here is how you execute a delete command:

```
[address1[,address2]]d
```

address1 and **address2** are the starting and the ending addresses respectively, which can be either line numbers or pattern strings. Both of these addresses are optional parameters.

As the name suggests, the delete command is used to perform delete operation and since the SED operates on line, we can say that this command is used to delete lines. Note that the delete command removes lines only from the pattern buffer; the line is not sent to the output stream and the original file remains unchanged. The following example illustrates the point.

```
[jerry]$ sed 'd' books.txt
```

But where is the output? If no line address is provided, then the SED operates on every line by default. Hence, it deletes all the lines from the pattern buffer. That is why the command does not print anything on the standard output.

Let us instruct the SED to operate only on certain lines. The following example removes the 4th line only.

```
[jerry]$ sed '4d' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Additionally, SED also accepts address range using comma(,). We can instruct the SED to remove N1 to N2 lines. For instance, the following example deletes all the lines from 2 through 4.

```
[jerry]$ sed '2, 4 d' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
```

Here, **address1** and **address2** are the starting and the ending address respectively, which can be either line numbers or pattern strings. Both of these addresses are optional parameters.

In the above syntax, **w** refers to the write command and **file** is the file name in which you store contents. Be careful with the **file** parameter. When a file name is provided, the SED creates a file on the fly if it is not present, and overwrites it if it is already present.

Let us make an exact copy of the file using SED. Note that there must be exactly one space between **w** and **file**.

```
[jerry]$ sed -n 'w books.bak' books.txt
```

We created another file called **books.bak**. Now verify that both the files have identical content.

```
[jerry]$ diff books.txt books.bak

[jerry]$ echo $?
```

On executing the above code, you get the following result:

```
0
```

You may assume that the **cp** command does exactly the same thing. Yes! The **cp** command does the same thing, but SED is a matured utility. It allows creating a file containing only certain lines from the source file. Let us store only even lines to another file.

```
[jerry]$ sed -n '2~2 w junk.txt' books.txt

[jerry]$ cat junk.txt
```

On executing the above code, you get the following result:

```
2) The Two Towers, J. R. R. Tolkien, 352

4) The Fellowship of the Ring, J. R. R. Tolkien, 432

6) A Game of Thrones, George R. R. Martin, 864
```

You can also use comma(,), dollar($), and plus(+) operators with the write command.

In addition to this, SED also supports pattern matching with the write command. Suppose you want to store all the books of individual authors into a separate file. One boring and lengthy way is do it manually, and the smarter way is to use SED.

```
[jerry]$ sed -n -e '/Martin/ w Martin.txt' -e '/Paulo/ w Paulo.txt' -e '/Tolkien/ w
Tolkien.txt' books.txt
```

In the above example, we are matching each line against a pattern and storing the matched line in a particular file. It is very simple. To specify multiple commands, we used **-e** switch of the SED command. Now let use see what each file contains:

```
[jerry]$ cat Martin.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
6) A Game of Thrones, George R. R. Martin, 864
```

Let us display the file contents.

```
[jerry]$ cat Paulo.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, Paulo Coelho, 197
5) The Pilgrimage, Paulo Coelho, 288
```

Let us display the file contents.

```
[jerry]$ cat Tolkien.txt
```

On executing the above code, you get the following result:

```
2) The Two Towers, J. R. R. Tolkien, 352
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
```

Excellent! We got the expected result. SED is really an amazing utility.

Append Command

One of the most useful operations of any text editor is to provide append functionality. SED supports this operation through its append command. Given below is the syntax of append:

```
[address]a\

Append text
```

Let us append a new book entry after line number 4. The following example shows how to do it.

```
[jerry]$ sed '4 a 7) Adultry, Paulo Coelho, 234' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216

2) The Two Towers, J. R. R. Tolkien, 352

3) The Alchemist, Paulo Coelho, 197

4) The Fellowship of the Ring, J. R. R. Tolkien, 432

7) Adultry, Paulo Coelho, 234

5) The Pilgrimage, Paulo Coelho, 288

6) A Game of Thrones, George R. R. Martin, 864
```

In the command section, **4** implies the line number, **a** is the append command, and the remaining part is the text to be appended.

Let us insert a text line at the end of the file. To do this, use **$** as the address. The following example illustrates this:

```
[jerry]$ sed '$ a 7) Adultry, Paulo Coelho, 234' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216

2) The Two Towers, J. R. R. Tolkien, 352

3) The Alchemist, Paulo Coelho, 197

4) The Fellowship of the Ring, J. R. R. Tolkien, 432

5) The Pilgrimage, Paulo Coelho, 288

6) A Game of Thrones, George R. R. Martin, 864

7) Adultry, Paulo Coelho, 234
```

Apart from line number, we can also specify an address using textual pattern. For instance, the following example appends text after matching the string **The Alchemist**.

```
[jerry]$ sed '/The Alchemist/ a 7) Adultry, Paulo Coelho, 234' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
7) Adultry, Paulo Coelho, 234
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Note that if there are multiple patterns matching, then the text is appended after each match. The following example illustrates this scenario.

```
[jerry]$ sed '/The/ a 7) Adultry, Paulo Coelho, 234' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
7) Adultry, Paulo Coelho, 234
3) The Alchemist, Paulo Coelho, 197
7) Adultry, Paulo Coelho, 234
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
7) Adultry, Paulo Coelho, 234
5) The Pilgrimage, Paulo Coelho, 288
7) Adultry, Paulo Coelho, 234
6) A Game of Thrones, George R. R. Martin, 864
```

Change Command

SED provides **change** or **replace** command which is represented by **c**. This command helps replace an existing line with new text. When line range is provided, all the lines are replaced as a group by a single text line. Given below is the syntax of the change command:

```
[address1[,address2]]c\
Replace text
```

Let us replace the third line with some other text.

```
[jerry]$ sed '3 c 3) Adultry, Paulo Coelho, 324' books.txt
```

On executing the above code, you get the following result:

```

1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) Adultry, Paulo Coelho, 324
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

SED also accepts patterns as an address. In the following example, a line is replaced when the pattern match succeeds.

```
[jerry]$ sed '/The Alchemist/ c 3) Adultry, Paulo Coelho, 324' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) Adultry, Paulo Coelho, 324
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

SED also allows replacement of multiple lines with a single line. The following example removes lines from fourth through sixth and replaces them with new text.

```
[jerry]$ sed '4, 6 c 4) Adultry, Paulo Coelho, 324' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) Adultry, Paulo Coelho, 324
```

Insert Command

The insert command works much in the same way as append does. The only difference is that it inserts a line before a specific position. Given below is the syntax of the insert command:

```
[address]i\
Insert text
```

Let us understand the insert command with some examples. The following command inserts a new entry before the fourth line.

```
[jerry]$ sed '4 i 7) Adultry, Paulo Coelho, 324' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
7) Adultry, Paulo Coelho, 324
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

In the above example, **4** is the location number, **i** implies the insert command, and the remaining part is the text to be inserted.

To insert text at the start of a file, provide the line address as **1**. The following command illustrates this:

```
[jerry]$ sed '1 i 7) Adultry, Paulo Coelho, 324' books.txt
```

On executing the above code, you get the following result:

```
7) Adultry, Paulo Coelho, 324
```

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Additionally, we can insert multiple lines. The following command inserts two lines before the last line.

```
[jerry]$ sed '$ i 7) Adultry, Paulo Coelho, 324\
```

On executing the above code, you get the following result:

```
8) Eleven Minutes, Paulo Coelho, 304' books.txt
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage,Paulo Coelho, 288
7) Adultry, Paulo Coelho, 324
8) Eleven Minutes, Paulo Coelho, 304
6) A Game of Thrones, George R. R. Martin, 864
```

Note that the entries to be inserted are entered on separate lines and delimited by the backslash(\) character.

Translate Command

SED provides a command to translate characters and it is represented as **y**. It transforms the characters by position. Given below is the syntax of the translate command:

```
[address1[,address2]]y/list-1/list-2/
```

Note that translation is based on the position of the character from **list 1** to the character in the same position in **list 2** and both lists must be explicit character lists. Regular expressions and character classes are unsupported. Additionally, the size of **list 1** and **list 2**must be same.

The following example converts Arabic numbers to Roman numbers.

```
[jerry]$ echo "1 5 15 20" | sed 'y/151520/IVXVXX/'
```

On executing the above code, you get the following result:

```
I V IV XX
```

l command

Can you differentiate between words separated by spaces and words separated by tab characters only by looking at them? Certainly not. But SED can do this for you. SED uses the **l** command to display hidden characters in the text. For example, tab character with **\t** and End-Of-Line with **$** character. Given below is the syntax of the **l** command.

```
[address1[,address2]]l

[address1[,address2]]l [len]
```

Let us create a file with tab characters for demonstration. For simplicity, we are going to use the same file, just by replacing spaces with tabs. Wait! But how to do that — by opening the file in a text editor and replacing each space with tab? Certainly not! We can make use of SED commands for that.

```
[jerry]$ sed 's/ /\t/g' books.txt > junk.txt
```

Now let us display the hidden characters by using the **l** command:

```
[jerry]$ sed -n 'l' junk.txt
```

On executing the above code, you get the following result:

```
1)\tA\tStorm\tof\tSwords,George\tR.\tR.\tMartin,1216$
2)\tThe\tTwo\tTowers,J.\tR.\tR.\tTolkien,352$
3)\tThe\tAlchemist,Paulo\tCoelho,197$
4)\tThe\tFellowship\tof\tthe\tRing,J.\tR.\tR.\tTolkien,432$
5)\tThe\tPilgrimage,Paulo\tCoelho,288$
6)\tA\tGame\tof\tThrones,George\tR.\tR.\tMartin\t,864$
```

Like other SED commands, it also accepts line numbers and patterns as an address. You can try it yourselves.

Let us take a close look at another interesting feature of SED. We can instruct the SED to perform line wrapping after a certain number of characters. The following example wraps lines after 25 characters.

```
[jerry]$ sed -n 'l 25' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords,Geo\
rge R. R. Martin,1216$
2) The Two Towers,J. R. \
R. Tolkien,352$
3) The Alchemist,Paulo C\
oelho,197$
4) The Fellowship of the\
 Ring,J. R. R. Tolkien,4\
32$
5) The Pilgrimage,Paulo \
Coelho,288$
6) A Game of Thrones,Geo\
rge R. R. Martin ,864$
```

Note that in the above example, wrap limit is provided after l command. In this case, it is 25 characters. This option is GNU specific and may not work with other variants of the SED.

A wrap limit of 0 means never break the line unless there is a new line character. The following simple command illustrates this.

```
[jerry]$ sed -n 'l 0' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords,George R. R. Martin,1216$
2) The Two Towers,J. R. R. Tolkien,352$
3) The Alchemist,Paulo Coelho,197$
4) The Fellowship of the Ring,J. R. R. Tolkien,432$
5) The Pilgrimage,Paulo Coelho,288$
6) A Game of Thrones,George R. R. Martin,864$
```

Quit Command

Quit command instructs the SED to quit the current execution flow. It is represented by the**q** command. Given below is the syntax of the quit command:

```
[address]q

[address]q [value]
```

Note that the quit command does not accept range of addresses, it only supports a single address. By default, SED follows read, execute, and repeat workflow; but when the quit command is encountered, it simply stops the current execution.

Let us print the first 3 lines from the file.

```
[jerry]$ sed '3 q' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216

2) The Two Towers, J. R. R. Tolkien, 352

3) The Alchemist, Paulo Coelho, 197
```

In addition to line number, we can also use textual patterns. The following command quits when pattern match succeeds.

```
[jerry]$ sed '/The Alchemist/ q' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216

2) The Two Towers, J. R. R. Tolkien, 352

3) The Alchemist, Paulo Coelho, 197
```

In addition to this, SED can also accept a **value** which can be used as the exit status. The following command shows its exit status as 100.

```
[jerry]$ sed '/The Alchemist/ q 100' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
```

```
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
```

Now let us verify the exit status.

```
[jerry]$ echo $?
```

On executing the above code, you get the following result:

```
100
```

Read Command

We can instruct the SED to read the contents of a file and display them when a specific condition matches. The command is represented by the alphabet **r**. Given below is the syntax of the read command.

```
[address]r file
```

Note that there must be exactly one space between the **r** command and the file name.

Let us understand it with a simple example. Create a sample file called **junk.txt**.

```
[jerry]$ echo "This is junk text." > junk.txt
```

The following command instructs the SED to read the contents of **junk.txt** and insert them after the third line.

```
[jerry]$ sed '3 r junk.txt' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
This is junk text.
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

In the above example, **3** implies the line address, **r** is the command name, and **junk.txt** is the file name the contents of which are to be displayed. Additionally, the GNU SED also accepts a range of addresses. For instance, the following command inserts the contents of**junk.txt** after the third, fourth, and fifth lines.

```
[jerry]$ sed '3, 5 r junk.txt' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
This is junk text.
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
This is junk text.
5) The Pilgrimage, Paulo Coelho, 288
This is junk text.
6) A Game of Thrones, George R. R. Martin, 864
```

Like other SED commands, the *read* command also accepts pattern as an address. For instance, the following command inserts the contents of **junk.txt** when the pattern match succeeds.

```
[jerry]$ sed '/Paulo/ r junk.txt' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
This is junk text.
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
This is junk text.
6) A Game of Thrones, George R. R. Martin, 864
```

Execute Command

We can execute external commands from SED using the **execute** command. It is represented by **e**. Given below is the syntax of the execute command.

```
[address1[,address2]]e [command]
```

Let us illustrate the execute command with a simple example. The following SED command executes the UNIX **date** command before the third line.

```
[jerry]$ sed '3 e date' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
Sun Sep  7 18:04:49 IST 2014
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Like other commands, it also accepts patterns as an address. For example, the following example executes **date** command when a pattern match succeeds. Note that after each pattern match, first the command is executed and then the contents of the pattern buffer are displayed.

```
[jerry]$ sed '/Paulo/ e date' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
Sun Sep  7 18:06:04 IST 2014
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
Sun Sep  7 18:06:04 IST 2014
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

If you observe the syntax of the **e** command carefully, you will notice that **command** is optional. When no command is provided after **e**, it treats the contents of the pattern buffer as an external command. To illustrate this, let us create a *commands.txt* file with a few simple commands.

```
[jerry]$ echo -e "date\ncal\nuname" > commands.txt
```

```
[jerry]$ cat commands.txt
```

On executing the above code, you get the following result:

```
date
cal
uname
```

Commands from the file are self-explanatory. In the absence of **command** after **e**, SED executes all these commands one by one. The following simple example illustrates this.

```
[jerry]$ sed 'e' commands.txt
```

On executing the above code, you get the following result:

```
Sun Sep  7 18:14:20 IST 2014
   September 2014
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

Linux
```

Like other SED commands, the execute command also accepts all valid ranges of addresses.

Miscellaneous Commands

By default, SED operates on single line, however it can operate on multiple lines as well. Multi-line commands are denoted by uppercase letters. For example, unlike the **n**command, the **N** command does not clear and print the pattern space. Instead, it adds a newline (\n) at the end of the current pattern space and appends the next line from the input-file to the current pattern space and continues with the SED's standard flow by executing the rest of the SED commands. Given below is the syntax of the **N** command.

```
[address1[,address2]]N
```

Let us print a comma-separated list of book titles and their respective authors. The following example illustrates this.

```
[jerry]$ sed 'N; s/\n/, /g' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords, George R. R. Martin
The Two Towers, J. R. R. Tolkien
The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
The Pilgrimage, Paulo Coelho
A Game of Thrones, George R. R. Martin
```

Let us understand how the above example works. The **N** command reads the first line, i.e.,*A Storm of Swords* into the pattern buffer and appends **\n** followed by the next line. The pattern space now contains *A Storm of Swords\nGeorge R. R. Martin*. In the next step, we are replacing the newline with a comma.

Like **p** command, we have a **P** command to print the first part (up to embedded newline) of the multi-line pattern space created by the **N** command. Given below is the syntax of the **P**command which is similar to the **p** command.

```
[address1[,address2]]P
```

In the previous example, we saw that the **N** command creates a newline-separated list of book titles and their authors. Let us print only the first part of it, i.e., only the titles of the book. The following command illustrates this.

```
[jerry]$ sed -n 'N;P' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords
The Two Towers
The Alchemist
The Fellowship of the Ring
The Pilgrimage
A Game of Thrones
```

Note that in the absence of **N**, it behaves same as the **p** command. The following simple command illustrates this scenario.

```
[jerry]$ sed -n 'P' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords
George R. R. Martin
The Two Towers
J. R. R. Tolkien
The Alchemist
Paulo Coelho
The Fellowship of the Ring
J. R. R. Tolkien
The Pilgrimage
Paulo Coelho
A Game of Thrones
George R. R. Martin
```

In addition to this, SED also provides a **v** command which checks for version. If the provided version is greater than the installed SED version, then the command execution fails. Note that this option is GNU specific and may not work with other variants of SED. Given below is the syntax of the **v** command.

```
[address1[,address2]]v [version]
```

First, find out the current version of SED.

```
[jerry]$ sed --version
```

On executing the above code, you get the following result:

```
sed (GNU sed) 4.2.2
```

In the following example, the SED version is greater than version 4.2.2, hence the SED command aborts its execution.

```
[jerry]$ sed 'v 4.2.3' books.txt
```

On executing the above code, you get the following result:

```
sed: -e expression #1, char 7: expected newer version of sed
```

But if the provided version is lesser than or equal to version 4.2.2, then the command works as expected.

```
[jerry]$ sed 'v 4.2.2' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords
George R. R. Martin
The Two Towers
J. R. R. Tolkien
The Alchemist
Paulo Coelho
The Fellowship of the Ring
J. R. R. Tolkien
The Pilgrimage
Paulo Coelho
A Game of Thrones
George R. R. Martin
```

10. SPECIAL CHARACTERS

SED provides two special characters which are treated as commands. This chapter illustrates the usage of these two special characters.

= Command

The "=" command deals with line numbers. Given below is the syntax of the "=" command:

```
[/pattern/]=

[address1[,address2]]=
```

The = command writes the line number followed by its contents on the standard output stream. The following example illustrates this.

```
[jerry]$ sed '=' books.txt
```

On executing the above code, you get the following result:

```
1
1) A Storm of Swords, George R. R. Martin, 1216
2
2) The Two Towers, J. R. R. Tolkien, 352
3
3) The Alchemist, Paulo Coelho, 197
4
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5
5) The Pilgrimage, Paulo Coelho, 288
6
6) A Game of Thrones, George R. R. Martin, 864
```

Let us print the line numbers and the contents of the first four lines. The following command prints the first four lines with line numbers and the remaining without line numbers.

```
[jerry]$ sed '1, 4=' books.txt
```

On executing the above code, you get the following result:

```
1
1) A Storm of Swords, George R. R. Martin, 1216
2
2) The Two Towers, J. R. R. Tolkien, 352
3
3) The Alchemist, Paulo Coelho, 197
4
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Additionally, we can instruct the SED to print line numbers when a pattern match succeeds. The following example prints the line number that contains the pattern *"Paulo"*.

```
[jerry]$ sed '/Paulo/ =' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

Can you guess what the following SED command does?

```
[jerry]$ sed -n '$ =' books.txt
```

On executing the above code, you get the following result:

```
6
```

Yes, you are right. It counts the total number of lines present in the file. Let us demystify the code. In the command section, we used "$ =" which prints the line number of the last line followed by its contents. But we also provided the **-n** flag which suppresses the default printing of the pattern buffer. Hence, only the last line number is displayed.

& Command

SED supports the special character &. Whenever a pattern match succeeds, this special character stores the matched pattern. It is often used with the substitution command. Let us see how we can leverage this efficient feature.

Each line in the book.txt file is numbered. Let us add the words **Book number** at the beginning of each line. The following example illustrates this.

```
[jerry]$ sed 's/[[:digit:]]/Book number &/' books.txt
```

On executing the above code, you get the following result:

```
Book number 1) A Storm of Swords, George R. R. Martin, 1216
Book number 2) The Two Towers, J. R. R. Tolkien, 352
Book number 3) The Alchemist, Paulo Coelho, 197
Book number 4) The Fellowship of the Ring, J. R. R. Tolkien, 432
Book number 5) The Pilgrimage, Paulo Coelho, 288
Book number 6) A Game of Thrones, George R. R. Martin, 864
```

This example is very simple. First, we search for the first occurrence of a digit, which is the line number (that is why we used [[:digit:]]) and the SED automatically stores the matched pattern in the special character &. In the second step, we insert the words **Book number**before each matched pattern, i.e., before every line.

Let us take another example. In the book.txt file, the last digit implies the number of pages of the book. Let us add "Pages =" before that. To do this, find the last occurrence of the digit and replace it with "Pages = &". Here, & stores the matched pattern, i.e., the number of pages.

```
[jerry]$ sed 's/[[:digit:]]*$/Pages = &/' books.txt
```

On executing the above syntax, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, Pages = 1216
2) The Two Towers, J. R. R. Tolkien, Pages = 352
3) The Alchemist, Paulo Coelho, Pages = 197
4) The Fellowship of the Ring, J. R. R. Tolkien, Pages = 432
5) The Pilgrimage, Paulo Coelho,Pages = 288
6) A Game of Thrones, George R. R. Martin, Pages = 864
```

For the time being, just remember that **[[:digit:]]*$** finds the last occurrence of the digit. In the chapter "Regular Expressions, we will explore more about regular expressions.

11. STRINGS

Substitute Command

Text substitution operations like "find and replace" are common in any text editor. In this section, we illustrate how SED performs text substitution. Given below is the syntax of the substitution command.

```
[address1[,address2]]s/pattern/replacement/[flags]
```

Here, **address1** and **address2** are the starting and ending addresses respectively, which can be either line numbers or pattern strings. Both these addresses are optional parameters. The *pattern* is the text which we want to replace with the *replacement* string. Additionally, we can specify optional *flags* with the SED.

In the books.txt file, we have used comma(,) to separate each column. Let us use vertical bar(|) to separate each column. To do this, replace comma(,) with vertical bar(|).

```
[jerry]$ sed 's/,/ | /' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords | George R. R. Martin, 1216
2) The Two Towers | J. R. R. Tolkien, 352
3) The Alchemist | Paulo Coelho, 197
4) The Fellowship of the Ring | J. R. R. Tolkien, 432
5) The Pilgrimage | Paulo Coelho, 288
6) A Game of Thrones | George R. R. Martin, 864
```

If you observe carefully, only the first comma is replaced and the second remains as it is. Why? As soon as the pattern matches, SED replaces it with the replacement string and moves to the next line. By default, it replaces only the first occurrence. To replace all occurrences, use the global flag (g) with SED as follows:

```
[jerry]$ sed 's/,/ | /g' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords | George R. R. Martin | 1216
2) The Two Towers | J. R. R. Tolkien | 352
3) The Alchemist | Paulo Coelho | 197
4) The Fellowship of the Ring | J. R. R. Tolkien | 432
5) The Pilgrimage | Paulo Coelho | 288
6) A Game of Thrones | George R. R. Martin | 864
```

Now all occurrences of commas(,) are replaced with vertical bar(|).

We can instruct the SED to perform text substitution only when a pattern match succeeds. The following example replaces comma(,) with vertical bar(|) only when a line contains the pattern *The Pilgrimage*.

```
[jerry]$ sed '/The Pilgrimage/ s/,/ | /g' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage | Paulo Coelho | 288
6) A Game of Thrones, George R. R. Martin, 864
```

In addition to this, SED can replace a specific occurrence of the pattern. Let us replace only the second instance of comma(,) with vertical bar(|).

```
[jerry]$ sed 's/,/ | /2' books.txt
```

On executing the above code, you get the following result:

```
1) A Storm of Swords, George R. R. Martin | 1216
2) The Two Towers, J. R. R. Tolkien | 352
3) The Alchemist, Paulo Coelho | 197
4) The Fellowship of the Ring, J. R. R. Tolkien | 432
5) The Pilgrimage,Paulo Coelho | 288
6) A Game of Thrones, George R. R. Martin  | 864
```

In the above example, the number at the end of the SED command (or at the place of *flag*) implies the 2nd occurrence.

SED provides an interesting feature. After performing substitution, SED provides an option to show only the changed lines. For this purpose, SED uses the **p** flag which refers to print. The following example lists only changed lines.

```
[jerry]$ sed -n 's/Paulo Coelho/PAULO COELHO/p' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, PAULO COELHO, 197
5) The Pilgrimage, PAULO COELHO, 288
```

We can store changed lines in another file as well. To achieve this result, use the **w** flag. The following example shows how to do it.

```
[jerry]$ sed -n 's/Paulo Coelho/PAULO COELHO/w junk.txt' books.txt
```

We used the same SED command. Let us verify the contents of the **junk.txt** file.

```
[jerry]$ cat junk.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, PAULO COELHO, 197
5) The Pilgrimage, PAULO COELHO, 288
```

To perform case-insensitive substitution, use the **i** flag which implies ignore case. The following example performs case-insensitive substitution.

```
[jerry]$ sed  -n 's/pAuLo CoElHo/PAULO COELHO/pi' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, PAULO COELHO, 197
5) The Pilgrimage, PAULO COELHO, 288
```

So far, we have used only the foreslash(/) character as a delimiter, but we can also use vertical bar(|), at sign(@), caret(^), exclamation mark(!) as a delimiter. The following example shows how to use other characters as a delimiter.

Let us assume you need to replace the path **/bin/sed** with **/home/jerry/src/sed/sed-4.2.2/sed**. Hence, your SED command looks like this:

```
[jerry]$ echo "/bin/sed" | sed 's/\/bin\/sed/\/home\/jerry\/src\/sed\/sed-4.2.2\/sed/'
```

On executing the above code, you get the following result:

```
/home/jerry/src/sed/sed-4.2.2/sed
```

We can make this command more readable and easy to understand. Let us use vertical bar(|) as delimiter and see the result.

```
[jerry]$ echo "/bin/sed" | sed 's|/bin/sed|/home/jerry/src/sed/sed-4.2.2/sed|'
```

On executing the above code, you get the following result:

```
/home/jerry/src/sed/sed-4.2.2/sed
```

Indeed! We got the same result and the syntax is more readable. Similarly, we can use the "at" sign (@) as a delimiter as follows:

```
[jerry]$ echo "/bin/sed" | sed 's@/bin/sed@/home/jerry/src/sed/sed-4.2.2/sed@'
```

On executing the above code, you get the following result:

```
/home/jerry/src/sed/sed-4.2.2/sed
```

In addition to this, we can use caret(^) as a delimiter.

```
[jerry]$ echo "/bin/sed" | sed 's^/bin/sed^/home/jerry/src/sed/sed-4.2.2/sed^'
```

On executing the above code, you get the following result:

```
/home/jerry/src/sed/sed-4.2.2/sed
```

We can also use exclamation mark (!) as a delimiter as follows:

```
[jerry]$ echo "/bin/sed" | sed 's!/bin/sed!/home/jerry/src/sed/sed-4.2.2/sed!'
```

On executing the above code, you get the following result:

```
/home/jerry/src/sed/sed-4.2.2/sed
```

Generally, backslash(/) is used as a delimiter but sometimes it is more convenient to use other supported delimiters with SED.

Creating a Substring

We learnt the powerful substitute command. Let us see if we can find a substring from a matched text. Let us understand how to do it with the help of an example.

Let us consider the following text:

```
[jerry]$ echo "Three One Two"
```

Suppose we have to arrange it into a sequence. Means, it should print *One* first, then *Two*, and finally *Three*. The following one-liner does the needful.

```
echo "Three One Two" | sed 's|\(\w\+\) \(\w\+\) \(\w\+\)|\2 \3 \1|'
```

Note that in the above example, vertical bar (|) is used as a delimiter.

In SED, substrings can be specified by using a grouping operator and it must be prefixed with an escape character, i.e., **\(** and **\)**.

\w is a regular expression that matches any letter, digit, or underscore and "+" is used to match more than one characters. In other words, the regular expression **\(\w\+\)** matches the single word from the input string.

In the input string, there are three words separated by space, hence there are three regular expressions separated by space. The first regular expression stores the first word, i.e.,**Three**, the second stores the word **One**, and the third stores the word **Two**.

These substrings are referred by **\N**, where N is the substring number. Hence, **\2** prints the second substring, i.e., **One**; **\3** prints the third substring, i.e., **Two**; and **\1** prints the first substring, i.e., **Three**.

Let us separate these words by commas(,) and modify the regular expression accordingly.

```
[jerry]$ echo "Three,One,Two" | sed 's|\(\w\+\),\(\w\+\),\(\w\+\)|\2,\3,\1|'
```

On executing the above code, you get the following result:

```
One,Two,Three
```

Note that now there is comma(,) instead of space in the regular expression.

String Replacement Flags (GNU SED only)

In the previous section, we saw some examples of the substitution command. The GNU SED provides some special escape sequences which can be used in the replacement string. Note that these string replacement flags are GNU specific and may not work with other variants of SED. Here we will discuss string replacement flags.

- \l: When \l is specified in the replacement string, it treats the immediate character after \l as a lowercase character. The following SED command replaces *Paulo* with*PAULO*.

  ```
  [jerry]$ sed -n 's/Paulo/PAULO/p' books.txt
  ```

  On executing the above code, you get the following result:

  ```
  3) The Alchemist, PAULO Coelho, 197
  5) The Pilgrimage, PAULO Coelho, 288
  ```

  Now let us specify \l in the replacement string and observe the result.

  ```
  [jerry]$ sed -n 's/Paulo/\lPAUL\lO/p' books.txt
  ```

  On executing the above code, you get the following result:

  ```
  3) The Alchemist, pAULo Coelho, 197
  5) The Pilgrimage, pAULo Coelho, 288
  ```

  In the above example, \l is used before the characters 'P' and 'O'. Hence, SED treats these characters as lowercase letters.

- \L: When \L is specified in the replacement string, it treats all the remaining characters of the the word after \L as lowercase characters. For example, the characters "ULO" are treated as lowercase characters.

  ```
  [jerry]$ sed -n 's/Paulo/PA\LULO/p' books.txt
  ```

  On executing the above code, you get the following result:

  ```
  3) The Alchemist, PAulo Coelho, 197
  ```

```
5) The Pilgrimage, PAulo Coelho, 288
```

- \u: When \u is specified in the replacement string, it treats the immediate character after \u as an uppercase character. In the following example, \u is used before the characters 'a' and 'o'. Hence SED treats these characters as uppercase letters.

```
[jerry]$ sed -n 's/Paulo/p\uaul\uo/p' books.txt
```

  On executing the above code, you get the following result:

```
3) The Alchemist, pAulO Coelho, 197
5) The Pilgrimage, pAulO Coelho, 288
```

- \U: When \U is specified in the replacement string, it treats all the remaining characters of the the word after \U as uppercase characters.

```
[jerry]$ sed -n 's/Paulo/\Upaulo/p' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, PAULO Coelho, 197
5) The Pilgrimage, PAULO Coelho, 288
```

- \E: This flag should be used with \L or \U. It stops the conversion initiated by the flag \L or \U. In the following example, only the first word is replaced with uppercase letters.

```
[jerry]$ sed -n 's/Paulo Coelho/\Upaulo \Ecoelho/p' books.txt
```

On executing the above code, you get the following result:

```
3) The Alchemist, PAULO coelho, 197
5) The Pilgrimage, PAULO coelho, 288
```

12. MANAGING PATTERNS

We have already discussed the use of pattern and hold buffer. In this chapter, we are going to explore more about their usage. Let us discuss the **n** command which prints the pattern space. It will be used in conjunction with other commands. Given below is the syntax of the**n** command.

```
[address1[,address2]]n
```

Let us take an example.

```
[jerry]$ sed 'n' books.txt
```

When the above code is executed, it will produce the following result:

```
1) A Storm of Swords, George R. R. Martin, 1216
2) The Two Towers, J. R. R. Tolkien, 352
3) The Alchemist, Paulo Coelho, 197
4) The Fellowship of the Ring, J. R. R. Tolkien, 432
5) The Pilgrimage, Paulo Coelho, 288
6) A Game of Thrones, George R. R. Martin, 864
```

The **n** command prints the contents of the pattern buffer, clears the pattern buffer, fetches the next line into the pattern buffer, and applies commands on it.

Let us consider there are three SED commands before **n** and two SED commands after **n** as follows:

```
Sed command #1
Sed command #2
Sed command #3
n command
Sed command #4
Sed command #5
```

In this case, SED applies the first three commands on the pattern buffer, clears the pattern buffer, fetches the next line into the pattern buffer, and thereafter applies the fourth and fifth commands on it. This is a very important concept. Do not go ahead without having a clear understanding of this.

The hold buffer holds data, but SED commands cannot be applied directly on the hold buffer. Hence, we need to bring the hold buffer data into the pattern buffer. SED provides the **x** command to exchange the contents of pattern and hold buffers. The following commands illustrate the **x** command.

Let us slightly modify the books.txt file. Say, the file contains book titles followed by their author names. After modification, the file should look like this:

```
[jerry]$ cat books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords
George R. R. Martin
The Two Towers
J. R. R. Tolkien
The Alchemist
Paulo Coelho
The Fellowship of the Ring
J. R. R. Tolkien
The Pilgrimage
Paulo Coelho
A Game of Thrones
George R. R. Martin
```

Let us exchange the contents of the two buffers. For instance, the following example prints only the names of authors.

```
[jerry]$ sed -n 'x;n;p' books.txt
```

On executing the above code, you get the following result:

```
George R. R. Martin
J. R. R. Tolkien
Paulo Coelho
J. R. R. Tolkien
Paulo Coelho
George R. R. Martin
```

Let us understand how this command works.

1. Initially, SED reads the first line, i.e., *A Storm of Swords* into the pattern buffer.
2. **x** command moves this line to the hold buffer.
3. **n** fetches the next line, i.e., *George R. R. Martin* into the pattern buffer.
4. The control passes to the command followed by **n** which prints the contents of the pattern buffer.
5. The process repeats until the file is exhausted.

Now let us exchange the contents of the buffers before printing. Guess, what happens? Yes, it prints the titles of books.

```
[jerry]$ sed -n 'x;n;x;p' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords
The Two Towers
The Alchemist
The Fellowship of the Ring
The Pilgrimage
A Game of Thrones
```

The **h** command deals with the hold buffer. It copies data from the pattern buffer to the hold buffer. Existing data from the hold buffer gets overwritten. Note that the **h** command does not move data, it only copies data. Hence, the copied data remains as it is in the pattern buffer. Given below is the syntax of the **h** command.

```
[address1[,address2]]h
```

The following command prints only the titles of the author *Paulo Coelho*.

```
[jerry]$ sed -n '/Paulo/!h; /Paulo/{x;p}' books.txt
```

On executing the above code, you get the following result:

```
The Alchemist
The Pilgrimage
```

Let us understand how the above command works. The contents of books.txt follow a specific format. The first line is the book title followed by the author of the book. In the above command, "!" is used to reverse the condition, i.e., line is copied to the hold buffer only when a pattern match does not succeed. And curly braces {} are used to group multiple SED commands.

In the first pass of the command, SED reads the first line, i.e., *A Storm of Swords* into the pattern buffer and checks whether it contains the pattern *Paulo* or not. As the pattern match does not succeed, it copies this line to the hold buffer. Now both the pattern buffer and the hold buffer contain the same line i.e., *A Storm of Swords*. In the second step, it checks whether the line contains the pattern *Paulo* or not. As the pattern does not match, it does not do anything.

In second pass, it reads the next line *George R. R. Martin* into the pattern buffer and applies the same steps. For the next three lines, it does the same thing. At the end of the fifth pass, both the buffers contain *The Alchemist*. At the start of the sixth pass, it reads the line *Paulo Coelho* and as the pattern matches, it does not copy this line into the hold buffer. Hence, the pattern buffer contains *Paulo Coelho*, and the hold buffer contains *The Alchemist*.

Thereafter, it checks whether the pattern buffer contains the pattern *Paulo*. As the pattern match succeeds, it exchanges the contents of the pattern buffer with the hold buffer. Now the pattern buffer contains *The Alchemist* and the hold buffer contains *Paulo Coelho*. Finally, it prints the contents of the pattern buffer. The same steps are applied to the pattern *The Pilgrimage*.

The **h** command destroys the previous contents of the hold buffer. This is not always acceptable, as sometimes we need to preserve the contents. For this purpose, SED provides the **H** command which appends the contents to the hold buffer by adding a new line at the end. The only difference between **h** and **H** command is, the former overwrites data from the hold buffer, while the later appends data to the hold buffer. Its syntax is similar to the **h**command.

```
[address1[,address2]]H
```

Let us take another example. This time, instead of printing only book titles, print the names of their authors too. The following example prints the book titles followed by their author names.

```
[jerry]$ sed -n '/Paulo/!h; /Paulo/{H;x;p}' books.txt
```

On executing the above code, you get the following result:

```
The Alchemist
Paulo Coelho
The Pilgrimage
```

```
Paulo Coelho
```

We learnt how to copy/append the contents of pattern buffer to hold buffer. Can we perform the reverse function as well? Yes certainly! For this purpose, SED provides the **g**command which copies data from the hold buffer to the pattern buffer. While copying, existing data from the pattern space gets overwritten. Given below is the syntax of the **g**command.

```
[address1[,address2]]g
```

Let us consider the same example - printing book titles and their authors. This time, we will first print the name of the author and on the next line, the corresponding book title. The following command prints the name of the author *Paulo Coelho*, followed by its book title.

```
[jerry]$ sed -n '/Paulo/!h; /Paulo/{p;g;p}' books.txt
```

On executing the above code, you get the following result:

```
Paulo Coelho
The Alchemist
Paulo Coelho
The Pilgrimage
```

The first command is kept as it is. At the end of fifth pass, both the buffers contain *The Alchemist*. At the start of the sixth pass, it reads the line *Paulo Coelho* and as the pattern matches, it does not copy this line into the hold buffer. Hence, the pattern space contains*Paulo Coelho* and the hold space contains *The Alchemist*.

Thereafter, it checks whether the pattern space contains the pattern *Paulo*. As the pattern match succeeds, it first prints the contents of the pattern space, i.e., *Paulo Coelho*, then it copies the hold buffer to the pattern buffer. Hence, both the pattern and hold buffers contain *The Alchemist*. Finally, it prints the contents of the pattern buffer. The same steps are applied to the pattern *The Pilgrimage*.

Similarly, we can append the contents of the hold buffer to the pattern buffer. SED provides the **G** command which appends the contents to the pattern buffer by adding a new line at the end.

```
[address1[,address2]]G
```

Now let us take the previous example which prints the name of author *Paulo Coelho*followed by its book title. To achieve the same result, execute the following SED command.

```
[jerry]$ sed -n '/Paulo/!h; /Paulo/{G;p}' books.txt
```

On executing the above code, you get the following result:

```
Paulo Coelho
The Alchemist
Paulo Coelho
The Pilgrimage
```

Can you modify the above example to display the book titles followed by their authors? Simple, just exchange the buffer contents before the **G** command.

```
[jerry]$ sed -n '/Paulo/!h; /Paulo/{x;G;p}' books.txt
```

On executing the above code, you get the following result:

```
The Alchemist
Paulo Coelho
The Pilgrimage
Paulo Coelho
```

13. REGULAR EXPRESSIONS

It is the regular expressions that make SED powerful and efficient. A number of complex tasks can be solved with regular expressions. Any command-line expert knows the power of regular expressions.

Like many other GNU/Linux utilities, SED too supports regular expressions, which are often referred to as as **regex**. This chapter describes regular expressions in detail. The chapter is divided into three sections: Standard regular expressions, POSIX classes of regular expressions, and Meta characters.

Standard Regular Expressions

Start of line (^)

In regular expressions terminology, the caret(^) symbol matches the start of a line. The following example prints all the lines that start with the pattern "The".

```
[jerry]$ sed -n '/^The/ p' books.txt
```

On executing the above code, you get the following result:

```
The Two Towers, J. R. R. Tolkien
The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
The Pilgrimage, Paulo Coelho
```

End of Line ($)

End of line is represented by the dollar($) symbol. The following example prints the lines that end with "Coelho".

```
[jerry]$ sed -n '/Coelho$/ p' books.txt
```

On executing the above code, you get the following result:

```
The Alchemist, Paulo Coelho
The Pilgrimage, Paulo Coelho
```

Single Character (.)

The Dot(.) matches any single character except the end of line character. The following example prints all three letter words that end with the character "t".

```
[jerry]$ echo -e "cat\nbat\nrat\nmat\nbatting\nrats\nmats" | sed -n '/^..t$/p'
```

On executing the above code, you get the following result:

```
cat
bat
rat
mat
```

Match Character Set ([])

In regular expression terminology, a character set is represented by square brackets ([]). It is used to match only one out of several characters. The following example matches the patterns "Call" and "Tall" but not "Ball".

```
[jerry]$ echo -e "Call\nTall\nBall" | sed -n '/[CT]all/ p'
```

On executing the above code, you get the following result:

```
Call
Tall
```

Exclusive Set ([^])

In exclusive set, the caret negates the set of characters in the square brackets. The following example prints only "Ball".

```
[jerry]$ echo -e "Call\nTall\nBall" | sed -n '/[^CT]all/ p'
```

On executing the above code, you get the following result:

```
Ball
```

Character Range ([-])

When a character range is provided, the regular expression matches any character within the range specified in square brackets. The following example matches "Call" and "Tall" but not "Ball".

```
[jerry]$ echo -e "Call\nTall\nBall" | sed -n '/[C-Z]all/ p'
```

On executing the above code, you get the following result:

```
Call
Tall
```

Now let us modify the range to "A-P" and observe the result.

```
[jerry]$ echo -e "Call\nTall\nBall" | sed -n '/[A-P]all/ p'
```

On executing the above code, you get the following result:

```
Call
Ball
```

Zero on One Occurrence (\?)

In SED, the question mark (\?) matches zero or one occurrence of the preceding character. The following example matches "Behaviour" as well as "Behavior". Here, we made "u" as an optional character by using "\?".

```
[jerry]$ echo -e "Behaviour\nBehavior" | sed -n '/Behaviou\?r/ p'
```

On executing the above code, you get the following result:

```
Behaviour
Behavior
```

One or More Occurrence (\+)

In SED, the plus symbol(\+) matches one or more occurrences of the preceding character. The following example matches one or more occurrences of "2".

```
[jerry]$ echo -e "111\n22\n123\n234\n456\n222"  | sed -n '/2\+/ p'
```

On executing the above code, you get the following result:

```
22
123
234
222
```

Zero or More Occurrence (*)

Asterisks (*) matches the zero or more occurrence of the preceding character. The following example matches "ca", "cat", "catt", and so on.

```
[jerry]$ echo -e "ca\ncat" | sed -n '/cat*/ p'
```

On executing the above code, you get the following result:

```
ca
```

```
cat
```

Exactly N Occurrences {n}

{n} matches exactly "n" occurrences of the preceding character. The following example prints only three digit numbers. But before that, you need to create the following file which contains only numbers.

```
[jerry]$ cat numbers.txt
```

On executing the above code, you get the following result:

```
1
10
100
1000
10000
100000
1000000
10000000
100000000
1000000000
```

Let us write the SED expression.

```
[jerry]$ sed -n '/^[0-9]\{3\}$/ p' numbers.txt
```

On executing the above code, you get the following result:

```
100
```

Note that the pair of curly braces is escaped by the "\" character.

At least n Occurrences {n,}

{n,} matches at least "n" occurrences of the preceding character. The following example prints all the numbers greater than or equal to five digits.

```
[jerry]$ sed -n '/^[0-9]\{5,\}$/ p' numbers.txt
```

On executing the above code, you get the following result:

```
10000
100000
1000000
```

```
10000000
100000000
1000000000
```

M to N Occurrence {m, n}

{m, n} matches at least "m" and at most "n" occurrences of the preceding character. The following example prints all the numbers having at least five digits but not more than eight digits.

```
[jerry]$ sed -n '/^[0-9]\{5,8\}$/ p' numbers.txt
```

On executing the above code, you get the following result:

```
10000
100000
1000000
10000000
```

Pipe (|)

In SED, the pipe character behaves like logical OR operation. It matches items from either side of the pipe. The following example either matches "str1" or "str3".

```
[jerry]$ echo -e "str1\nstr2\nstr3\nstr4" | sed -n '/str\(1\|3\)/ p'
```

On executing the above code, you get the following result:

```
str1
str3
```

Note that the pair of the parenthesis and pipe (|) is escaped by the "\" character.

Escaping Characters

There are certain special characters. For example, newline is represented by "\n", carriage return is represented by "\r", and so on. To use these characters into regular ASCII context, we have to escape them using the backward slash(\) character. This chapter illustrates escaping of special characters.

Escaping "\"

The following example matches the pattern "\".

```
[jerry]$ echo 'str1\str2' | sed -n '/\\/ p'
```

On executing the above code, you get the following result:

```
str1\str2
```

Escaping "\n"

The following example matches the new line character.

```
[jerry]$ echo 'str1\nstr2' | sed -n '/\\n/ p'
```

On executing the above code, you get the following result:

```
str1\nstr2
```

Escaping "\r"

The following example matches the carriage return.

```
[jerry]$ echo 'str1\rstr2' | sed -n '/\\r/ p'
```

On executing the above code, you get the following result:

```
str1\rstr2
```

Escaping "\dnnn"

This matches a character whose decimal ASCII value is "nnn". The following example matches only the character "a".

```
[jerry]$ echo -e "a\nb\nc" | sed -n '/\d97/ p'
```

On executing the above code, you get the following result:

```
a
```

Escaping "\onnn"

This matches a character whose octal ASCII value is "nnn". The following example matches only the character "b".

```
[jerry]$ echo -e "a\nb\nc" | sed -n '/\o142/ p'
```

On executing the above code, you get the following result:

```
b
```

Escaping "\xnnn"

This matches a character whose hexadecimal ASCII value is "nnn". The following example matches only the character "c".

```
[jerry]$ echo -e "a\nb\nc" | sed -n '/\x63/ p'
```

On executing the above code, you get the following result:

```
c
```

POSIX Classes of Regular Expressions

There are certain reserved words which have special meaning. These reserved words are referred to as POSIX classes of regular expression. This section describes the POSIX classes supported by SED.

[:alnum:]

It implies alphabetical and numeric characters. The following example matches only "One" and "123", but does not match the tab character.

```
[jerry]$ echo -e "One\n123\n\t" | sed -n '/[[:alnum:]]/ p'
```

On executing the above code, you get the following result:

```
One

123
```

[:alpha:]

It implies alphabetical characters only. The following example matches only the word "One".

```
[jerry]$ echo -e "One\n123\n\t" | sed -n '/[[:alpha:]]/ p'
```

On executing the above code, you get the following result:

```
One
```

[:blank:]

It implies blank character which can be either space or tab. The following example matches only the tab character.

```
[jerry]$ echo -e "One\n123\n\t" | sed -n '/[[:space:]]/ p' | cat -vte
```

On executing the above code, you get the following result:

```
^I$
```

Note that the command "cat -vte" is used to show tab characters (^I).

[:digit:]

It implies decimal numbers only. The following example matches only digit "123".

```
[jerry]$ echo -e "abc\n123\n\t" | sed -n '/[[:digit:]]/ p'
```

On executing the above code, you get the following result:

```
123
```

[:lower:]

It implies lowercase letters only. The following example matches only "one".

```
[jerry]$ echo -e "one\nTWO\n\t" | sed -n '/[[:lower:]]/ p'
```

On executing the above code, you get the following result:

```
one
```

[:upper:]

It implies uppercase letters only. The following example matches only "TWO".

```
[jerry]$ echo -e "one\nTWO\n\t" | sed -n '/[[:upper:]]/ p'
```

On executing the above code, you get the following result:

```
TWO
```

[:punct:]

It implies punctuation marks which include non-space or alphanumeric characters.

```
[jerry]$ echo -e "One,Two\nThree\nFour" | sed -n '/[[:punct:]]/ p'
```

On executing the above code, you get the following result:

```
One,Two
```

[:space:]

It implies whitespace characters. The following example illustrates this.

```
[jerry]$ echo -e "One\n123\f\t" | sed -n '/[[:space:]]/ p' | cat -vte
```

On executing the above code, you get the following result:

```
123^L^I$
```

Metacharacters

Like traditional regular expressions, SED also supports metacharacters. These are Perl style regular expressions. Note that metacharacter support is GNU SED specific and may not work with other variants of SED. Let us discuss metacharacters in detail.

Word Boundary(\b)

In regular expression terminology, "\b" matches the word boundary. For example, "\bthe\b" matches "the" but not "these", "there", "they", "then", and so on. The following example illustrates this.

```
[jerry]$ echo -e "these\nthe\nthey\nthen" | sed -n '/\bthe\b/ p'
```

On executing the above code, you get the following result:

```
the
```

Non-Word Boundary (\B)

In regular expression terminology, "\B" matches non-word boundary. For example, "the\B" matches "these" and "they" but not "the". The following example illustrates this.

```
[jerry]$ echo -e "these\nthe\nthey" | sed -n '/the\B/ p'
```

On executing the above code, you get the following result:

```
these
they
```

Single Whitespace (\s)

In SED, "\s" implies single whitespace character. The following example matches "Line\t1" but does not match "Line1".

```
[jerry]$ echo -e "Line\t1\nLine2" | sed -n '/Line\s/ p'
```

On executing the above code, you get the following result:

```
Line	1
```

Single Non-Whitespace (\S)

In SED, "\S" implies single whitespace character. The following example matches "Line2" but does not match "Line\t1".

```
[jerry]$ echo -e "Line\t1\nLine2" | sed -n '/Line\S/ p'
```

On executing the above code, you get the following result:

```
Line2
```

Single Word Character (\w)

In SED, "\w" implies single word character, i.e., alphabetical characters, digits, and underscore (_). The following example illustrates this.

```
[jerry]$ echo -e "One\n123\n1_2\n&;#" | sed -n '/\w/ p'
```

On executing the above code, you get the following result:

```
One
123
1_2
```

Single Non-Word Character (\W)

In SED, "\W" implies single non-word character which is exactly opposite to "\w". The following example illustrates this.

```
[jerry]$ echo -e "One\n123\n1_2\n&;#" | sed -n '/\W/ p'
```

On executing the above code, you get the following result:

```
&;#
```

Beginning of Pattern Space(\`)

In SED, "\`" implies the beginning of the pattern space. The following example matches only the word "One".

```
[jerry]$ echo -e "One\nTwo One" | sed -n '/\`One/ p'
```

On executing the above code, you get the following result:

```
One
```

14. USEFUL RECIPES

SED is an amazing utility that allows multiple ways to solve a problem. This is the UNIX way and SED perfectly proves that. GNU/Linux provides many useful utilities to perform day-to-day tasks. Let us simulate a few utilities using SED. Sometimes it may appear we are solving an easy problem the hard way, but the purpose is just to demonstrate the power of SED.

Cat Command

In the following example, each line is printed as a part of the default workflow.

```
[jerry]$ sed '' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords, George R. R. Martin
The Two Towers, J. R. R. Tolkien
The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
The Pilgrimage, Paulo Coelho
A Game of Thrones, George R. R. Martin
```

The following example uses print command to display the file contents.

```
[jerry]$ sed -n 'p' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords, George R. R. Martin
The Two Towers, J. R. R. Tolkien
The Alchemist, Paulo Coelho
The Fellowship of the Ring, J. R. R. Tolkien
The Pilgrimage, Paulo Coelho
A Game of Thrones, George R. R. Martin
```

Removing Empty Lines

In the following example, "^$" implies empty line, and empty lines are deleted when a pattern match succeeds.

```
[jerry]$ echo -e "Line #1\n\n\nLine #2" | sed '/^$/d'
```

On executing the above code, you get the following result:

```
Line #1
Line #2
```

Similarly, the following example prints the line only when it is non-empty.

```
[jerry]$ echo -e "Line #1\n\n\nLine #2" | sed -n '/^$/!p'
```

On executing the above code, you get the following result:

```
Line #1
Line #2
```

Removing Commented Lines from a C++ Program

Let us create a sample C++ program.

```
#include <iostream>
using namespace std;
int main(void)
{
         // Displays message on stdout.
         cout << "Hello, World !!!" << endl;

         return 0; // Return success.
}
```

Now remove the comments using the following regular expression.

```
[jerry]$ sed 's|//.*||g' hello.cpp
```

On executing the above code, you get the following result:

```
#include <iostream>
using namespace std;
int main(void)
{
         cout << "Hello, World !!!" << endl;
         return 0;
}
```

Adding Comments Before Certain Lines

The following example adds comments before line numbers 3 to 5.

```
[jerry]$ sed '3,5 s/^/#/' hello.sh
```

On executing the above code, you get the following result:

```
#!/bin/bash
#pwd
#hostname
#uname -a
who
who -r
lsb_release -a
```

Wc -l command

The "wc -l" command counts the number of lines present in the file. The following SED expression simulates the same.

```
[jerry]$ sed -n '$ =' hello.sh
```

On executing the above code, you get the following result:

```
8
```

Head Command

By default, the head command prints the first 10 lines of the file. Let us simulate the same behavior with SED.

```
[jerry]$ sed '10 q' books.txt
```

On executing the above code, you get the following result:

```
A Storm of Swords
George R. R. Martin
The Two Towers
J. R. R. Tolkien
The Alchemist
Paulo Coelho
The Fellowship of the Ring
J. R. R. Tolkien
The Pilgrimage
```

```
Paulo Coelho
```

Tail -1 Command

The "tail -1" prints the last line of the file. The following syntax shows its simulation.

```
[jerry]$ echo -e "Line #1\nLine #2" > test.txt

[jerry]$ cat test.txt
```

On executing the above code, you get the following result:

```
Line #1

Line #2
```

Let us write the SED script.

```
[jerry]$ sed -n '$p' test.txt
```

On executing the above code, you get the following result:

```
Line #2
```

Dos2unix Command

In DOS environment, a newline is represented by a combination of CR/LF characters. The following simulation of "dos2unix" command converts a DOS newline character to UNIX newline character. In GNU/Linux, this character is often treated as "^M" (Control M) character.

```
[jerry]$ echo -e "Line #1\r\nLine #2\r" > test.txt

[jerry]$ file test.txt
```

On executing the above code, you get the following result:

```
test.txt: ASCII text, with CRLF line terminators
```

Let us simulate the command using SED.

```
[jerry]$ sed 's/^M$//' test.txt > new.txt   # Press "ctrl+v" followed "ctrl+m" to generate
"^M" character.
[jerry]$ file new.txt
```

On executing the above code, you get the following result:

```
new.txt: ASCII text
```

Now let us display the file contents.

```
[jerry]$ cat -vte new.txt
```

On executing the above code, you get the following result:

```
Line #1$
Line #2$
```

Unix2dos command

Similar to "dos2unix", there is "unix2dos" command which converts UNIX newline character to DOS newline character. The following example shows simulation of the same.

```
[jerry]$ echo -e "Line #1\nLine #2" > test.txt
[jerry]$ file test.txt
```

On executing the above code, you get the following result:

```
test.txt: ASCII text
```

Let us simulate the command using SED.

```
[jerry]$ sed 's/$/\r/' test.txt  > new.txt
[jerry]$ file new.txt
```

On executing the above code, you get the following result:

```
new.txt: ASCII text, with CRLF line terminators
```

Now let us display the file contents.

```
[jerry]$ cat -vte new.txt
```

On executing the above code, you get the following result:

```
Line #1^M$
Line #2^M$
```

Cat -E command

The "cat -E" command shows the end of line by Dollar($) character. The following SED example is simulation of the same.

```
[jerry]$ echo -e "Line #1\nLine #2" > test.txt
[jerry]$ cat -E test.txt
```

On executing the above code, you get the following result:

```
Line #1$
Line #2$
```

Let us simulate the command using SED.

```
[jerry]$ sed 's|$|&$|' test.txt
```

On executing the above code, you get the following result:

```
Line #1$
Line #2$
```

Cat -ET Command

The "cat -ET" command shows the Dollar($) symbol at the end of each line and displays the TAB characters as "^I". The following example shows the simulation of "cat -ET" command using SED.

```
[jerry]$ echo -e "Line #1\tLine #2" > test.txt
[jerry]$ cat -ET test.txt
```

On executing the above code, you get the following result:

```
Line #1^ILine #2$
```

Let us simulate the command using SED.

```
[jerry]$ sed -n 'l' test.txt | sed 'y/\\t/^I/'
```

On executing the above code, you get the following result:

```
Line #1^ILine #2$
```

nl Command

The "nl" command simply numbers the lines of files. The following SED script simulates this behavior.

```
[jerry]$ echo -e "Line #1\nLine #2" > test.txt

[jerry]$ sed = test.txt | sed 'N;s/\n/\t/'
```

On executing the above code, you get the following result:

```
1         Line #1

2         Line #2
```

The first SED expression prints line numbers followed by their contents, and the second SED expression merges these two lines and converts newline characters to TAB characters.

cp Command

The "cp" command crates another copy of the file. The following SED script simulates this behavior.

```
[jerry]$ sed -n 'w dup.txt' data.txt

[jerry]$ diff data.txt dup.txt

[jerry]$ echo $?
```

On executing the above code, you get the following result:

```
0
```

Expand Command

The "expand" command converts TAB characters to whitespaces. The following code shows its simulation.

```
[jerry]$ echo -e "One\tTwo\tThree" > test.txt
[jerry]$ expand test.txt > expand.txt
[jerry]$ sed 's/\t/     /g' test.txt > new.txt
[jerry]$ diff new.txt expand.txt
[jerry]$ echo $?
```

On executing the above code, you get the following result:

```
0
```

Tee Command

The "tee" command dumps the data to the standard output stream as well as file. Given below is the simulation of the "tee" command.

```
[jerry]$ echo -e "Line #1\nLine #2" | tee test.txt
Line #1
Line #2
```

Let us simulate the command using SED.

```
[jerry]$ sed -n 'p; w new.txt' test.txt
```

On executing the above code, you get the following result:

```
Line #1
Line #2
```

cat -s Command

UNIX "cat -s" command suppresses repeated empty output lines. The following code shows the simulation of "cat -s" command.

```
[jerry]$ echo -e "Line #1\n\n\n\nLine #2\n\n\nLine #3" > test.txt
[jerry]$ cat -s test.txt
```

On executing the above code, you get the following result:

```
Line #1

Line #2

Line #3
```

Let us simulate the command using SED.

```
[jerry]$ sed '1s/^$//p;/./,/^$/!d' test.txt
```

On executing the above code, you get the following result:

```
Line #1

Line #2

Line #3
```

grep Command

By default, the "grep" command prints a line when a pattern match succeeds. The following code shows its simulation.

```
[jerry]$ echo -e "Line #1\nLine #2\nLine #3" > test.txt

[jerry]$ grep "Line #1" test.txt
```

On executing the above code, you get the following result:

```
Line #1
```

Let us simulate the command using SED.

```
[jerry]$ sed -n '/Line #1/p' test.txt
```

On executing the above code, you get the following result:

```
Line #1
```

grep -v Command

By default, the "grep -v" command prints a line when a pattern match fails. The following code shows its simulation.

```
[jerry]$ echo -e "Line #1\nLine #2\nLine #3" > test.txt

[jerry]$ grep -v "Line #1" test.txt
```

On executing the above code, you get the following result:

```
Line #2

Line #3
```

Let us simulate the command using SED.

```
[jerry]$ sed -n '/Line #1/!p' test.txt
```

On executing the above code, you get the following result:

```
Line #2

Line #3
```

tr Command

The "tr" command translates characters. Given below is its simulation.

```
[jerry]$ echo "ABC" | tr "ABC" "abc"
```

On executing the above code, you get the following result:

```
abc
```

Let us simulate the command using SED.

```


[jerry]$ echo "ABC" | sed 'y/ABC/abc/'
```

On executing the above code, you get the following result:

```
abc
```