Linux Handbook

Complete Sed Command Guide [Explained with Practical Examples]

Last Updated on MARCH 23, 2019 *By* SYLVAIN LEROUX (HTTPS://LINUXHANDBOOK.COM/AUTHOR/SYLVAIN/)

In a previous article, I showed the basic usage of Sed (https://linuxhandbook.com/sed-command-basics/), the stream editor, on a practical use case. Today, be prepared to gain more insight about Sed as we will take an in-depth tour of the sed execution model. This will be also an opportunity to make an exhaustive review of all Sed commands and to dive into their details and subtleties. So, if you are ready, launch a terminal, download the test files (https://gist.github.com/s-leroux/5cb36435bac46c10cfced26e4bf5588c) and sit comfortably before your keyboard: we will start our exploration right now!

Table of Contents

A little bit of theory on Sed

Linux Handbook
SED COMMAND REFERENCE GUIDE
EXPLAINED WITH PRACTICAL EXAMPLES

A first look at the sed execution model

To truly understand Sed you must first understand the tool execution model.

When processing data, Sed reads one line of input at a time and stores it into the so-called *pattern space*. All Sed's transformations apply to the pattern space. Transformations are described by one-letter commands provided on the command line or in an external Sed script file. Most Sed commands can be preceded by an address, or an address range, to limit their scope.

By default, Sed prints the content of the pattern space at the end of each processing cycle, that is, just before overwriting the pattern space with the next line of input. We can summarize that model like that:

1. Try to read the next input line into the pattern space

2. If the read was successful:

    1. Apply in the script order all commands whose address matches the current input line

2. If sed was not launched in quiet mode (`-n`) print the content of the (potentially modified) pattern space

3. got back to 1.

Since the content of the pattern space is lost after each line is processed, it is not suitable for long-term storage. For that purpose, Sed has a second buffer, the *hold space*. Sed never clears, puts or gets data from the hold space unless you explicitly request it. We will investigate that more in depth later when studying the exchange, get and hold commands.

The Sed abstract machine

The model explained above is what you will see described in many Sed tutorials. Indeed, it is correct enough to understand the most basic Sed programs. But when you start digging into more advanced commands, you will see it is not sufficient. So let's try to be a little bit more formal now.

Actually, Sed can be viewed as implementing an abstract machine (http://mathworld.wolfram.com/AbstractMachine.html) whose state (https://en.wikipedia.org/wiki/State_(computer_science)) is defined by three buffers (https://en.wikipedia.org/wiki/Data_buffer), two registers (https://en.wikipedia.org/wiki/Processor_register#Categories_of_registers), and two flags (https://www.computerhope.com/jargon/f/flag.htm):

- **three buffers** to store arbitrary length text. Yes: *three*! In the basic execution model we talked about the pattern- and hold-space, but Sed has a third buffer: the *append queue*. From the Sed script perspective, it is a write-only buffer that Sed will flush automatically at predefined moments of its execution (broadly speaking before reading a new line from the input, or just before quitting).

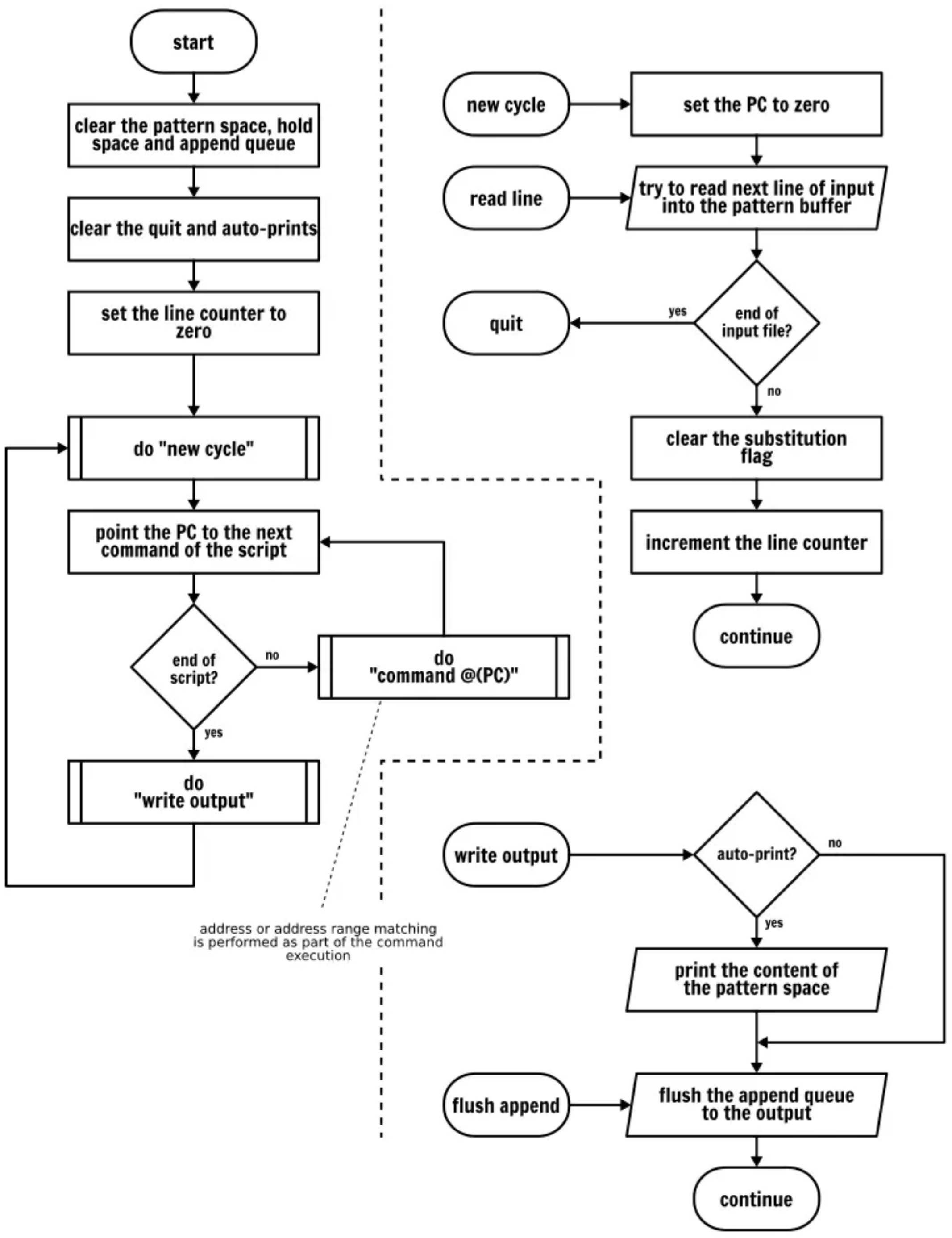
- Sed also maintains **two registers**: the *line counter* (LC) which holds the number of lines read from the input, and the *program counter* (PC) which always hold the index ("position" in the script) of the next command to execute. Sed automatically increments the PC as part of its main loop. But using specific commands, a script can also directly modify the PC to skip or repeat parts of the program. This is how loops or conditional statements can be implemented with Sed. More on that in the dedicated branches section below.

- Finally **two flags** can modify the behavior of certain Sed commands: the *auto-print flag* (AP) the *substitution flag* (SF). When the auto-print flag is *set*, Sed will automatically print the content of the pattern space *before* overwriting it (notably before reading a new line of input but not only). When the auto-print flag is *clear* ("not set"), Sed will never print the content of the pattern space without an explicit command in the script. You can clear the auto-print flag by running Sed in "quiet mode" (using the `-n` command line option or by using the special comment `#n` on the very first line or the script). The "substitution flag" is set by the substitution command (the `s` command) when both its address and search pattern match the content of the pattern space. The substitution flag is cleared at the start of each new cycle, or when a new line is read from input, or after a conditional branch is taken. Here again, we will revisit that topic in details in the branches section.

In addition, Sed maintains the list of commands having entered their address range (more on that of the range addresses section) as well as a couple of file handles to read and write data. You will find some more information on that in the read and write command description.

A more accurate Sed execution model

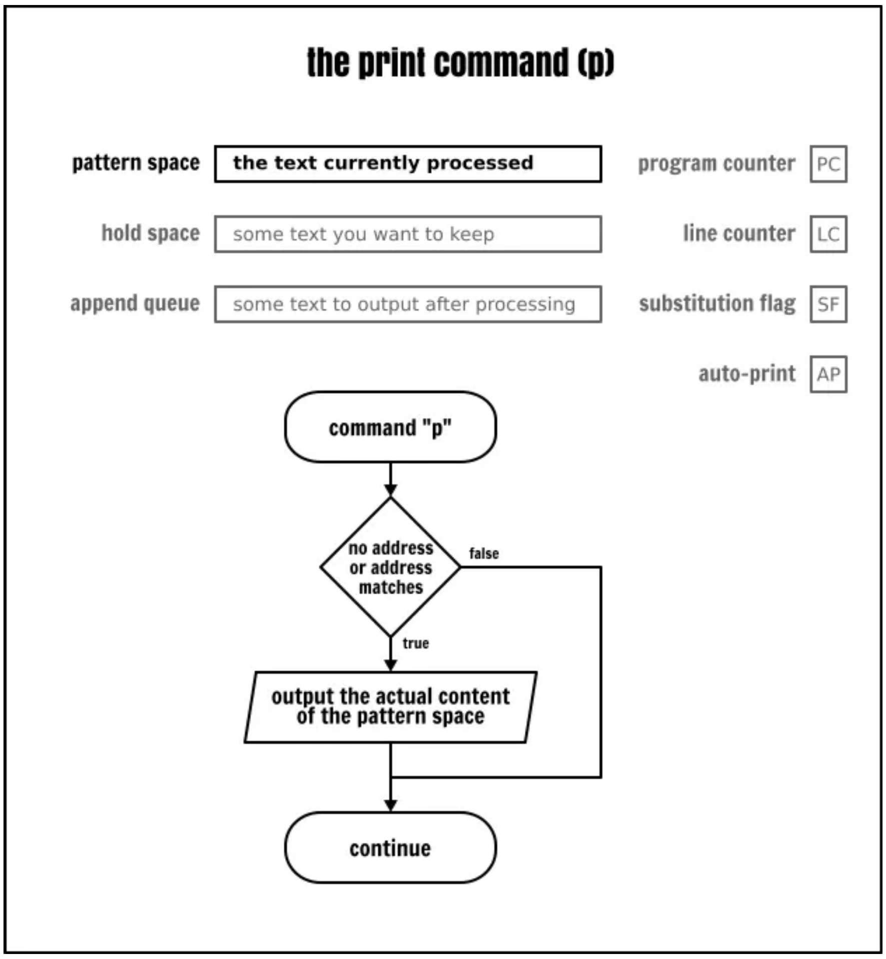
As a picture worth thousands of words, I draw a flowchart describing the Sed execution model. I left a couple of things aside, like dealing with multiple input files or error handling, but I think this should be sufficient for you to understand the behavior of any Sed program and to avoid wasting your time by groping around while writing your own Sed scripts.


start
clear the pattern space, hold space and append queue
clear the quit and auto-prints
set the line counter to zero
do "new cycle"
point the PC to the next command of the script
end of script?
no
yes
do "command @(PC)"
do "write output"
address or address range matching is performed as part of the command execution
new cycle
set the PC to zero
read line
try to read next line of input into the pattern buffer
end of input file?
yes
quit
no
clear the substitution flag
increment the line counter
continue
write output
auto-print?
no
yes
print the content of the pattern space
flush append
flush the append queue to the output
continue

You may have noticed I didn't describe the command-specific actions in the flowchart above. We will see that in detail for each command. So, without further ado, let's start our tour!

The print command

The print command (p) displays the content of the pattern space at the moment it is executed. It does not change in any way the state of the Sed abstract machine.


the print command (p)
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter PC
line counter LC
substitution flag SF
auto-print AP
command "p"
no address or address matches
false
true
output the actual content of the pattern space
continue

For example:

```
sed -e 'p' inputfile
```

The command above will print each line of the input file … twice. Once because you *explicitly* requested it using the `print` command, and a second time *implicitly* at the end of the processing loop (because we didn't launch Sed in "quiet mode" here).

If we are not interested in seeing each line twice, we have two way of fixing that:

```
  sed -n -e 'p' inputfile # quiet mode with explicit print
 sed -e '' inputfile  # empty "do nothing" program, implicit print
```

Note: the `-e` option introduces a Sed command. It is used to distinguish between commands and file names. Since a Sed invocation must contain at least one command, the `-e` flag is optional for that first command. However, I have the habit of using it, mostly for consistency with more complex cases where I have to give multiple Sed expressions on the command line. I let you figure by yourself if this is a good or bad habit, but I will follow that convention in the rest of the article.

Addresses

Obviously, the print command is not very useful by itself. However, if you add an address before the command to apply it only to some lines of the input file, it suddenly becomes able to filter out unwanted lines from a file. But what's an address for Sed? And how are identified the "lines" of the input file?

Line numbers

A Sed address can be either a *line number* (with the extension of `$` to mean "the last line") or a *regular expression*. When using line numbers, you have to remember lines are numbered starting from one in Sed— and not starting from zero.

```
 sed -n -e '1p' inputfile # print only the first line of the file
sed -n -e '5p' inputfile # print only the line 5
sed -n -e '$p' inputfile # print the last line of the file
sed -n -e '0p' inputfile # will result in an error because 0 is not a valid line number
```

According to the POSIX specifications (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html), line numbers are cumulative if you specify several input files. In other words, the line counter is not reset when Sed opens a new input file. So, the two commands below will do the same thing, printing only one line of text to the output:

```
 sed -n -e '1p' inputfile1 inputfile2 inputfile3
cat inputfile1 inputfile2 inputfile3 | sed -n -e '1p'
```

Actually, this is exactly how POSIX defines multiple file handling:

> *If multiple file operands are specified, the named files shall be read in the order specified and the concatenation shall be edited.*

However, some Sed implementations offer command line options to change that behavior, like the GNU Sed `-s` flag (which is implicitly applied too when using the GNU Sed `-i` flags):

```
 sed -sn -e '1p' inputfile1 inputfile2 inputfile3
```

If your implementation of Sed supports such non-standard options, I let you check into the `man (https://linux.die.net/man/1/man)` for the details regarding them.

Regular expressions

I've said Sed addresses could be line numbers or regular expressions. But what is this *regex* thing?

In just a few words, a regular expression (https://www.regular-expressions.info/) is a way to describe a *set* of strings. If a given string pertains to the set described by a regular expression, we said the string matches the regular expression.

A regular expression may contain literal characters that must be present verbatim into a string to have a match. For example, all letters and digits behave that way, as well as most printable characters. However, some symbols have a special meaning:

- They could represent anchors, like `^` and `$` that respectively denotes the start or end of a line;
- other symbols can serve as placeholders for entire sets of characters (like the dot that matches any single characters or the square brackets that are used to define a custom character set);
- others again are quantifiers that serve to denotes repetitions (like the Kleene star (https://chortle.ccsu.edu/FiniteAutomata/Section07/sect07_16.html) that means 0, 1 or several occurrences of the previous pattern).

My goal here is not to give you a regular expression tutorial. So I will stick with just a few examples. However, feel free to search for more on the web about that topic: regular expressions are a really powerful feature available in many standard Unix commands and programming language and a skill every Unixien should master.

Here are few examples used in the context of a Sed address:

```
 sed -n -e '/systemd/p' inputfile # print only lines *containing* the literal string "systemd"
sed -n -e '/nologin$/p' inputfile # print only lines ending with "nologin"
sed -n -e '/^bin/p' inputfile # print only lines starting with "bin"
sed -n -e '/^$/p' inputfile # print only empty lines (i.e.: nothing between the start and end c
sed -n -e '/./p' inputfile # print only lines containing a character (i.e. print only non-empty
sed -n -e '/^.$/p' inputfile # print only lines containing exactly one character
sed -n -e '/admin.*false/p' inputfile # print only lines containing "admin" followed by "false"
sed -n -e '/1[0,3]/p' inputfile # print only lines containing a "1" followed by a "0" or "3"
sed -n -e '/1[0-2]/p' inputfile # print only lines containing a "1" followed by a "0", "1", "2"
sed -n -e '/1.*2/p' inputfile # print only lines containing the character "1" followed by a "2"
sed -n -e '/1[0-9]*2/p' inputfile # print only lines containing the character "1" followed by z
```

If you want to remove the special meaning of a character in a regular expression (including the regex delimiter symbol), you have to precede it with a slash:

```
 # Print all lines containing the string "/usr/sbin/nologin"
sed -ne '/\/usr\/sbin\/nologin/p' inputfile
```

You are not limited to use only the slash as the regular expression delimiter in an address. You can use any other character that could suit your needs and tastes by preceding the first delimiter by a backslash. This is particularly useful when you have addresses that should match literal slashes like when working with file paths:

```
 # Both commands are perfectly identical
sed -ne '/\/usr\/sbin\/nologin/p' inputfile
sed -ne '\=/usr/sbin/nologin=p' inputfile
```

Extended regular expressions

By default, the Sed regular expression engine only understands the POSIX basic regular expression (https://www.regular-expressions.info/posix.html#bre) syntax. If you need extended regular expressions (https://www.regular-expressions.info/posix.html#ere), you must add the `-E` flag to the Sed command. Extended regular expressions add a couple of extra feature to basic regular expressions and, most important maybe, they require far fewer backslashes. I let you compare:

```
  sed -n -e '/\(www\)\|\(mail\)/p' inputfile
 sed -En -e '/(www)|(mail)/p' inputfile
```

The bracket quantifier

One powerful feature of regular expressions is the range quantifier (https://www.regular-expressions.info/repeat.html#limit) `{,}`. Actually, when written exactly like that, this quantifier is a perfect synonym for the `*` quantifier. However, you can explicitly add a lower and upper bound on each side of the coma, something that gives a tremendous amount of flexibility. When the lower bound of the range quantifier is omitted, it is assumed to be zero. When the upper bound is omitted, it is assumed to be the infinity:

> **READ 4 Essential and Practical Usage of Cut Command in Linux (https://linuxhandbook.com/cut-command/)**

Bracket notation	Shorthand	Description
{,}	*	zero, one or many occurrences of th e preceding regex
{,1}	?	zero or one occurrence of the prece ding regex
{1,}	+	one or many occurrences of the pre ceding regex
{n,n}	{n}	exactly n occurrences of the precedi ng regex

The bracket notation is available in basic regular expressions too, but it requires backslashes. According to POSIX, the only quantifiers available in basic regular expression are the star and the bracket notation (with backslashes `\{m,n\}`). Many regex engines do support the `\?` and `\+` notation as an extension. However, why tempting the devil? If you need those quantifiers, using extended regular expressions will be both easier to write and more portable.

If I took the time to talk about the bracket notation for regex quantifiers, this is because that feature is often useful in Sed scripts to count characters.

```
 sed -En -e '/^.{35}$/p' inputfile # Print lines containing exactly 35 characters
sed -En -e '/^.{0,35}$/p' inputfile # Print lines containing 35 characters or less
sed -En -e '/^.{,35}$/p' inputfile # Print lines containing 35 characters or less
sed -En -e '/^.{35,}$/p' inputfile # Print lines containing 35 characters or more
sed -En -e '/.{35}/p' inputfile # I let you figure out this one by yourself (test it!)
```

Range addresses

All the addresses we used so far were unique addresses. When using an unique address, the command is applied only the line(s) matching that address. However, Sed also supports range addresses. They are used to apply a command to all lines *between* the start and the end address of the range:

```
 sed -n -e '1,5p' inputfile # print only lines 1 through 5
sed -n -e '5,$p' inputfile # print from line 5 to the end of the file

sed -n -e '/home/217283.cloudwaysapps.com/wgnxdnqkza/public_html/,/systemd/p' inputfile # print
```

If the same line *number* is used both for the start and end address, the range is reduced to that line. Actually, if the second address is a number less than or equal to the line number of the first selected line of the range, only one line will be selected:

```
 printf "%s\n" {a,b,c}{d,e,f} | cat -n | sed -ne '4,4p'
     4	bd
printf "%s\n" {a,b,c}{d,e,f} | cat -n | sed -ne '4,3p'
     4	bd
```

This is somewhat tricky, but the rule given in the previous paragraph also applies when the *start address is a regular expression*. In that case, Sed will compare the *line number* of the first line matching the regex with the explicit *line number* given as end address. Once again, if the end line number is lower or equal to the start line number, the range will be reduced to one line:

```
 # The /b/,4 address will match *three* one-line range
# since each matching line has a line number >= 4
printf "%s\n" {a,b,c}{d,e,f} | cat -n | sed -ne '/b/,4p'
     4  bd
     5  be
     6  bf

# I let you figure by yourself how many ranges are matched
# by that second example:
printf "%s\n" {a,b,c}{d,e,f} | cat -n | sed -ne '/d/,4p'
     1  ad
     2  ae
     3  af
     4  bd
     7  cd
```

However, the behavior of Sed is different when the end address is a *regular expression*. In that case, the first line of the range is *not* tested against the end address, so the range will contain at least two lines (except of course if there is not enough input data):

```
printf "%s\n" {a,b,c}{d,e,f} | cat -n | sed -ne '/b/,/d/p'
    4    bd
    5    be
    6    bf
    7    cd

printf "%s\n" {a,b,c}{d,e,f} | cat -n | sed -ne '4,/d/p'
    4    bd
    5    be
    6    bf
    7    cd
```

Complement

Adding an exclamation mark (`!`) after an address select lines *not* matching that address. For example:

```
 sed -n -e '5!p' inputfile # Print all lines except line 5
sed -n -e '5,10!p' inputfile # Print all lines except line 5 to 10
sed -n -e '/sys/!p' inputfile # Print all lines except those containing "sys"
```

Conjunctions

Sed allows to group commands in blocks using brackets (`{…}`). You can leverage that feature to combine several addresses. For example, let’s compare the output of those two commands:

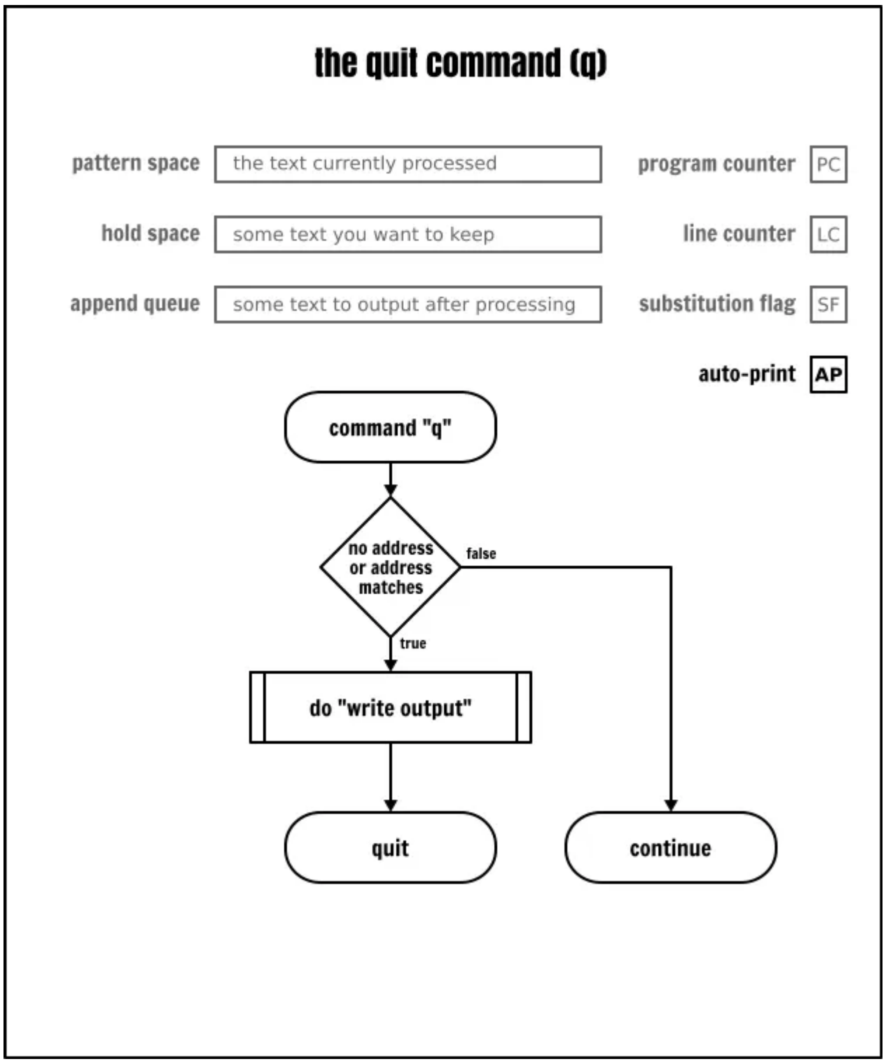
```
 sed -n -e '/usb/{
/daemon/p
}' inputfile

sed -n -e '/usb.*daemon/p' inputfile
```

By nesting commands in a block, we will select lines containing “usb” and “daemon” in any order. Whereas the regular expression “usb.*daemon” would only match lines where the “usb” string appears before the “daemon” string.

After that long digression, let’s go back now to our review of the various Sed commands.

The quit command

The quit command will stop Sed at the end of the current processing loop iteration.



The quit command is a way to stop input processing before reaching the end of the input file. Why would someone want to do that?

Well, if you remember, we've seen you can print the lines 1 through 5 of a file using the following command:

```
sed -n -e '1,5p' inputfile
```

With most implementations of Sed, the tool will read and cycle over all the remaining input lines, even if only the first five can produce a result. This may have a significant impact if your file contains millions of rows (or even worst, if you read from an infinite stream of data like `/dev/urandom` for example).

Using the quit command, the same program can be rewritten much more efficiently:

```
sed -e '5q' inputfile
```

Here, since I did not use the `-n` option, Sed will implicitly print the pattern space at the end of each cycle, but it will quit, and thus not reading more data, after having processed the line 5.

We could use a similar trick to print only a specific line of a file. That will be a good occasion to see several ways of providing multiple Sed expressions from the command line. The three variations bellow take benefit of Sed accepting several commands, either as different `-e` options, or in the same expression, but separated by newlines or semi-colon:

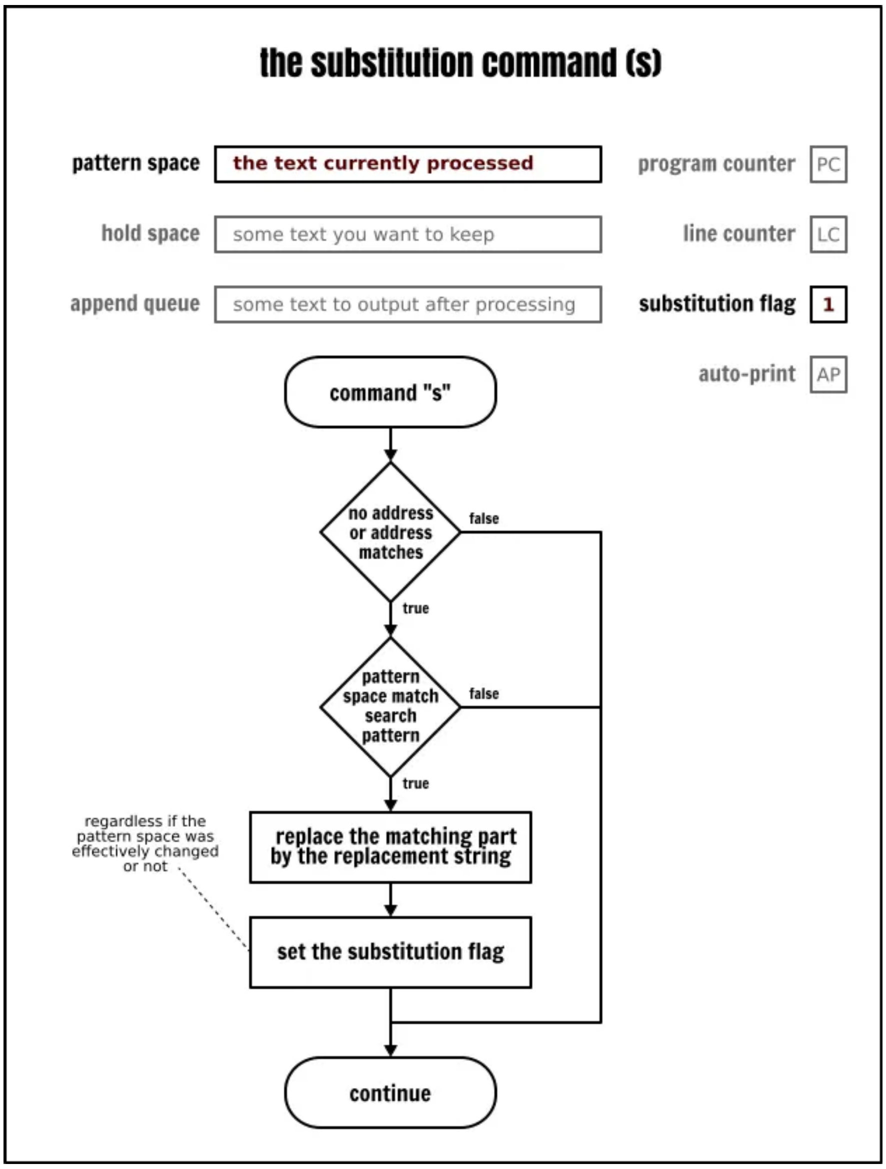
```
 sed -n -e '5p' -e '5q' inputfile

sed -n -e '
  5p
  5q
' inputfile

sed -n -e '5p;5q' inputfile
```

If you remember it, we've seen earlier we can group commands using brackets, something that we could use here to avoid repeating the same address twice:

```
 # By grouping commands
sed -e '5{
  p
  q
}' inputfile

# Which can be shortened as:
sed '5{p;q;}' inputfile

# As a POSIX extension, some implementations makes the semi-colon before the closing bracket op
sed '5{p;q}' inputfile
```

The substitution command

You can imagine the substitution command as the Sed equivalent to the search-replace feature you can find on most WYSIWYG editors. An equivalent, albeit a more powerful one though. The substitution command being one of the best-known Sed commands, it is largely documented on the web.


the substitution command (s)
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
1
auto-print
AP
command "s"
no address or address matches
false
true
pattern space match search pattern
false
true
replace the matching part by the replacement string
regardless if the pattern space was effectively changed or not
set the substitution flag
continue

We already have covered it in a previous article (http://linuxhandbook.com/?p=128) so I will not repeat myself here. However there are some key points to remember if you're not yet completely familiar with it:

- The substitution command takes two parameters: the search pattern and the replacement string: `sed s/:/-----/ inputfile`
- The command and its arguments are separated by an arbitrary character. Mostly by habit, 99% of the time we use a slash but any other character can be used: `sed s%:%-----% inputfile`, `sed sX:X-----X inputfile` or even `sed 's : ----- ' inputfile`
- By default, the substitution is applied only to the first matching substring of the pattern space. You can change that by specifying the match index as a flag after the command: `sed 's/:/-----/1' inputfile`, `sed 's/:/-----/2' inputfile`, `sed 's/:/-----/3' inputfile`, …

- If you want to perform a substitution globally (i.e.: on each non-overlapping match of the pattern space), you need to add the `g` flag: `sed 's/:/-----/g' inputfile`

- In the replacement string, any occurrence of the ampersand (`&`) will be replaced by the substring matching the search pattern: `sed 's/:/-&&&-/g' inputfile` , `sed 's/…./& /g' inputfile`

- Parenthesis (`(…)` in extended regex or `\(…\)` in basic regex) introduce a *capturing group*. That is a part of the matching string that can be referenced in the replacement string. `\1` is the content of the first capturing group, `\2` the content of the second one and so on: `sed -E 's/(.)(.)/\2\1/g' inputfile` , `sed -E 's/(.):x:(.):(.*)/\1:\3/' inputfile` (that latter works because the star regular expression quantifier is greedy (https://www.regular-expressions.info/repeat.html#greedy), and matches as many characters as it can)

- In the search pattern or the replacement string you can remove the special meaning of any character by preceding it with a backslash: `sed 's/:/--\&--/g' inputfile` , `sed 's/\//\\/g' inputfile`

As all this might seem a little bit abstract, here are a couple of examples. To start, let's say I want to display the first field of my test input file padded with spaces on the right up to 20 characters, I could write something like that:

```
sed < inputfile -E -e '
 s/:/                    / # replace the first field separator by 20 spaces
 s/(.{20}).*/\1/           # keep only the first 20 characters of the line
 s/.*/| & |/               # add vertical bars for a nice output
'
```

As a second example, if I want to change the UID/GID of the user sonia to 1100, I could write something like that:

```
sed -En -e '
 /sonia/{
   s/[0-9]+/1100/g
   p
 }' inputfile
```

Notice the `g` option at the end of the substitution command. It modifies its behavior, so all occurrences of the search pattern are replaced. Without that option, only the first one would be.

By the way, this is also a good occasion to mention the print command displays the content of the pattern space at the moment the command is executed. So, I can obtain a before-after output like that:

```
sed -En -e '
 /sonia/{
   p
   s/[0-9]+/1100/g
   p
 }' inputfile
```

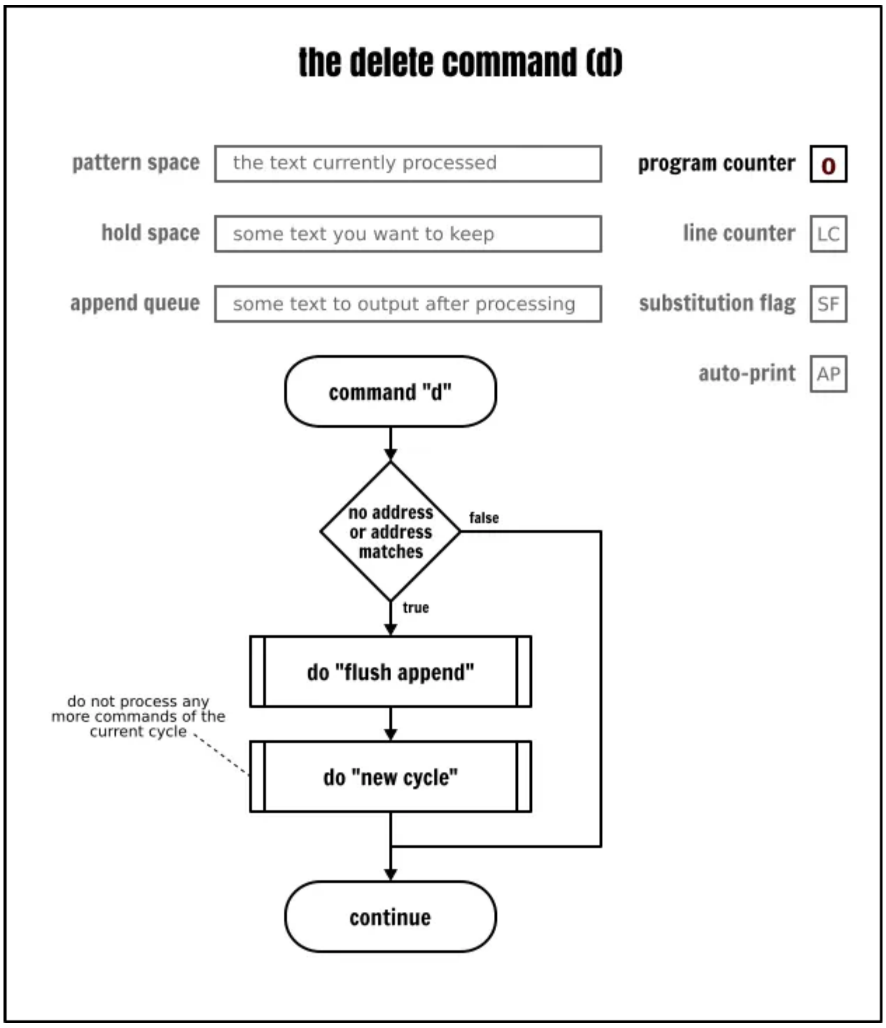
Actually, since printing a line after a substitution is a common use case, the substitution command also accepts the `p` option for that purpose:

```
sed -En -e '/sonia/s/[0-9]+/1100/gp' inputfile
```

Finally, I won't be exhaustive without mentioning the `w` option of the substitution command. We will examine it in detail later.

The delete command

The delete command (`d`) is used to clear the pattern space and immediately start the next cycle. By doing so, it will also skip the implicit print of the pattern space even if the auto-print flag is set.



A particularly inefficient way of printing only the first five lines of a file would be:

```
sed -e '6,$d' inputfile
```

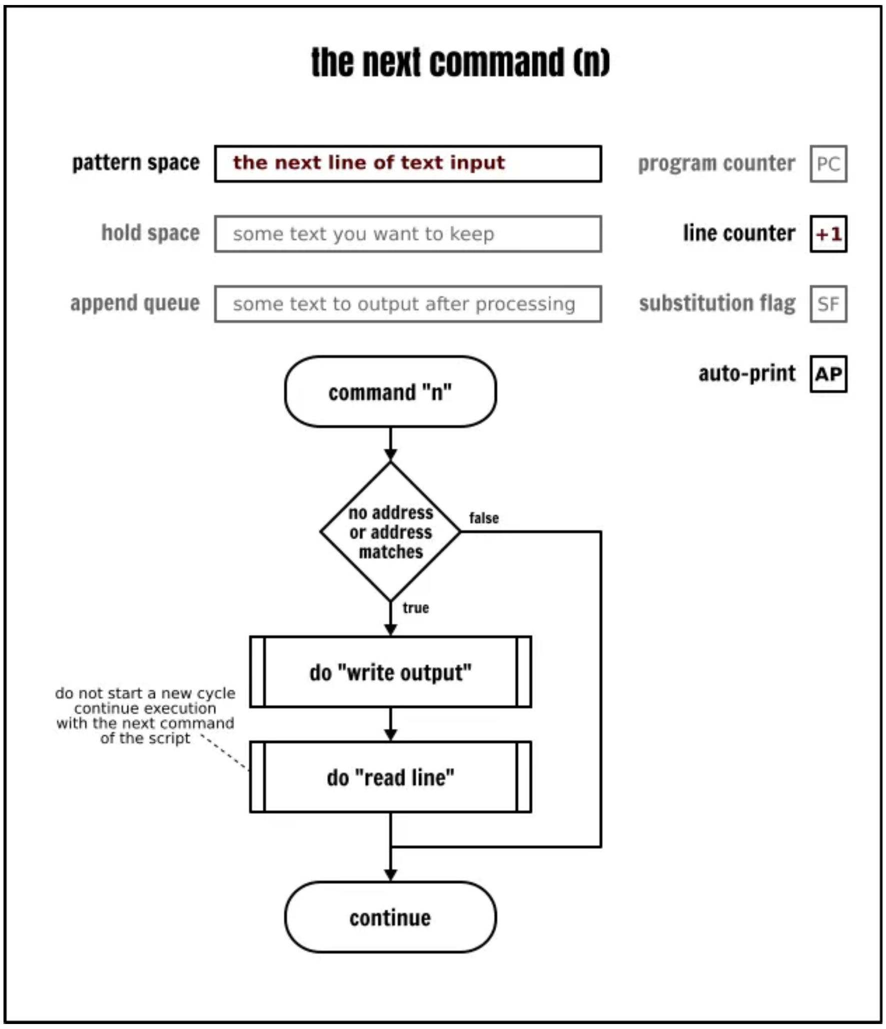
I let you guess why I said this was inefficient. If this is not obvious, try to re-read the section concerning the quit command. The answer is there!

The delete command is particularly useful when combined with regular expression-based addresses to remove matching lines from the output:

```
sed -e '/systemd/d' inputfile
```

The next command

This command prints the current pattern space if Sed is not running in quiet mode, then, in all cases, it reads the next input line into the pattern space and executes the *remaining commands of the current cycle* with the new pattern space.


the next command (n)
pattern space
the next line of text input
program counter
PC
hold space
some text you want to keep
line counter
+1
append queue
some text to output after processing
substitution flag
SF
auto-print
AP
command "n"
no address or address matches
false
true
do "write output"
do not start a new cycle continue execution with the next command of the script
do "read line"
continue

A common use case of the next command is to skip lines:

```
cat -n inputfile | sed -n -e 'n;n;p'
```

In the example above, Sed will implicitly read the first line of the input file. But the `next` command discards (and does not display because of the `-n` option) the pattern space and replaces it with the next line from the input. And the second `next` command will do the same thing, skipping now the line 2 of the input. And finally, the script explicitly prints the pattern space which now contains the third line of the input. Then Sed will start a new cycle, implicitly reading the line 4, then skipping it, as well as the line 5, because of the `next` commands, and it will print the line 6. And again and again until the end of the file. Concretely, this prints one line over three of the input file.

READ **tr command in Linux Explained With Examples**
(https://linuxhandbook.com/tr-command/)

Using the next command, we can also find a couple of other ways to display the first five lines of a file:

```
  cat -n inputfile | sed -n -e '1{p;n;p;n;p;n;p;n;p}'
cat -n inputfile | sed -n -e 'p;n;p;n;p;n;p;n;p;q'
cat -n inputfile | sed -e 'n;n;n;n;q'
```

More interestingly, the next command is also very useful when you want to process lines *relative* to some address:

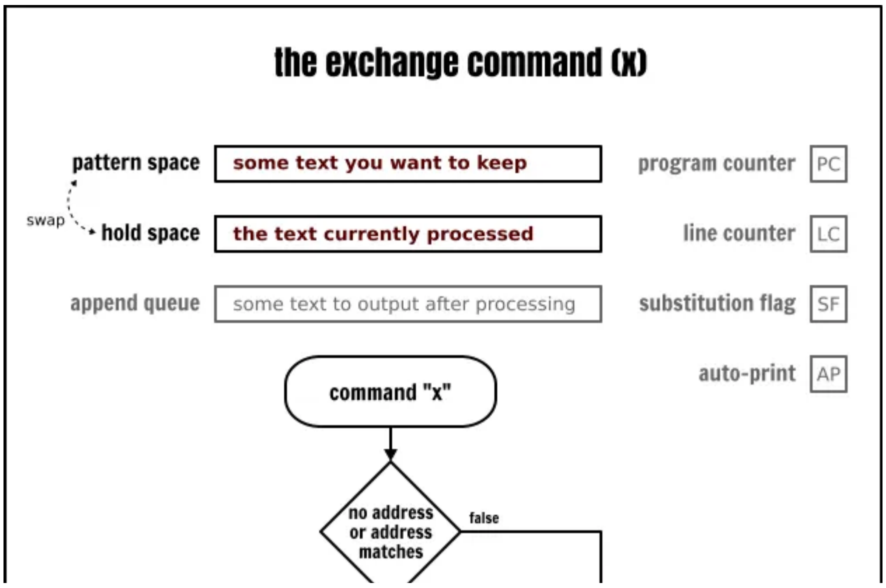
```
  cat -n inputfile | sed -n '/pulse/p'       # print lines containing "pulse"
cat -n inputfile | sed -n '/pulse/{n;p}'   # print the line following
                                           # the line containing "pulse"
cat -n inputfile | sed -n '/pulse/{n;n;p}' # print the line following
                                           # the line following
                                           # the line containing "pulse"
```

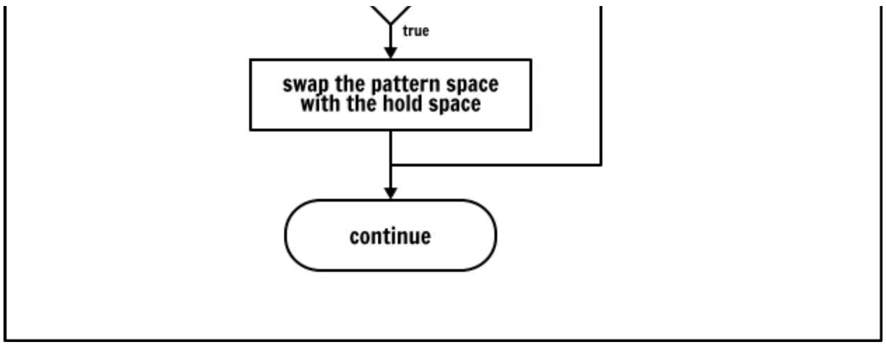
Working with the hold space

Until now, the command we've seen dealt only with the pattern space. However, as we've mentioned it at the very top of this article, there is a second buffer, the hold space, entirely under the control of the user. This will be the purpose of the commands described in this section.

The exchange command

As it names implies it, the exchange command (x) will swap the content of the hold and pattern space. Remember as long as you didn't put anything into the hold space, it is empty.





As a first example, we may use the exchange command to print in reverse order the first two lines of a file:

```
cat -n inputfile | sed -n -e 'x;n;p;x;p;q'
```

Of course, you don't have to use the content of the hold space immediately after having set it, since the hold space remains untouched as long as you don't explicitly modify it. In the following example, I use it to move the first line of the input after the fifth one:

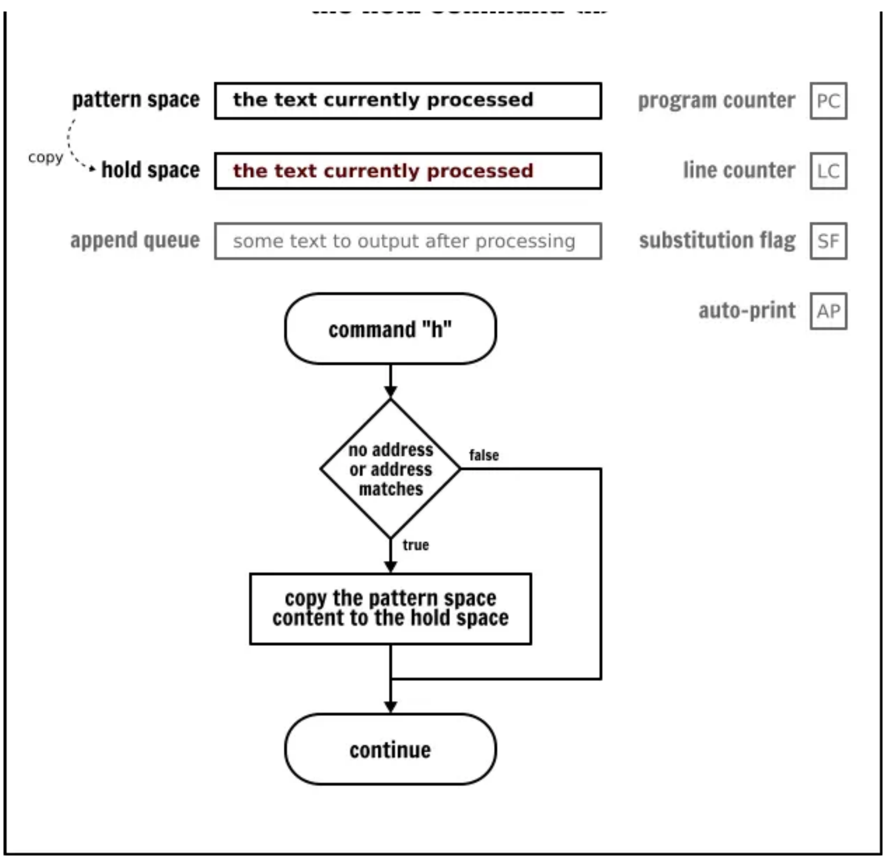
```
cat -n inputfile | sed -n -e '
 1{x;n}                # Swap the hold and pattern space
                       # to store line 1 into the hold buffer
                       # and then read line two
 5{
   p                   # print line 5
   x                   # swap the hold and pattern space to get
                       # back the content of line one into the
                       # pattern space
 }

 1,5p                  # triggered on lines 2 through 5
                       # (not a typo! try to figure why this rule
                       # is NOT executed for line 1;)
'
```

The hold commands

The hold command (h) is used to store the content of the pattern space into the hold space. However, as opposed to the exchange command, this time the content of the pattern space is left unchanged. The hold commands came in two flavors:

`h`
: that will copy the content of the pattern space into the hold space, overwriting any value already present

`H`
: that will append the content of the pattern space to the hold space, using a newline as separator


pattern space
the text currently processed
copy
hold space
the text currently processed
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command "h"
no address or address matches
false
true
copy the pattern space content to the hold space
continue

The example above using the exchange command can be rewritten using the hold command instead:

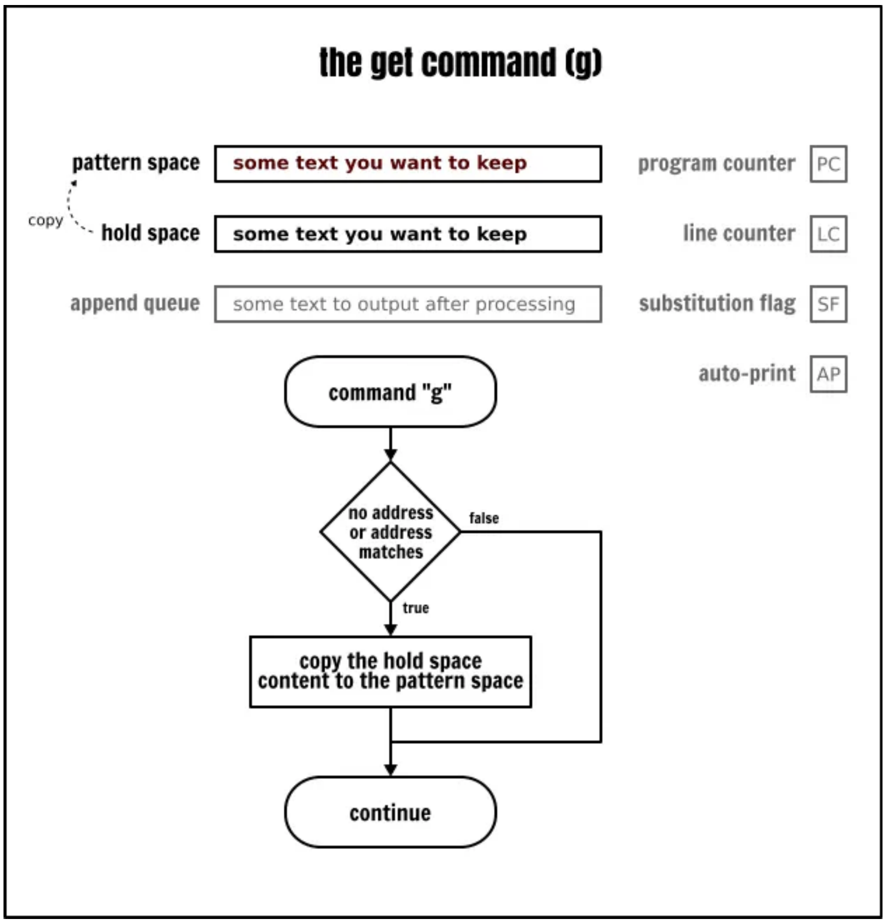
```
cat -n inputfile | sed -n -e '
 1{h;n}                # Store line 1 into the hold buffer and continue
 5{                    # on line 5
   x                   # switch the pattern and hold space
                       # (now the pattern space contains the line 1)
   H                   # append the line 1 after the line 5 in the hold space
   x                   # switch again to get back lines 5 and 1 into
                       # the pattern space
 }

 1,5p                  # triggered on lines 2 through 5
                       # (not a typo! try to figure why this rule
                       # is NOT executed for line 1;)
'
```

The get command

The get command (g) does the exact opposite of the hold command: it takes the content of the hold space and put it into the pattern space. Here again, it comes in two flavors:

`g`
  that will copy the content of the hold space into the pattern space, overwriting any value already present

`G`
: that will append the content of the hold space to the pattern space, using a newline as separator


the get command (g)
pattern space
some text you want to keep
copy
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command "g"
no address or address matches
false
true
copy the hold space content to the pattern space
continue

Together, the hold and get commands allow to store and recall data. As a little challenge, I let you rewrite the example of the previous section to put the line 1 of the input file after the line 5, but this time using the get and hold commands (lower- or upper-case version), but *without* using the exchange command. With a little bit of luck, it should be simpler that way!

In the meantime, I can show you another example that could serve for your inspiration. The goal here is to separate the users having a login shell from the others:

```
cat -n inputfile | sed -En -e '
 \=(/usr/sbin/nologin|/bin/false)$= { H;d; }
             # Append matching lines to the hold buffer
             # and continue to next cycle
 p           # Print other lines
 $ { g;p }   # On the last line,
             # get and print the content of the hold buffer
'
```

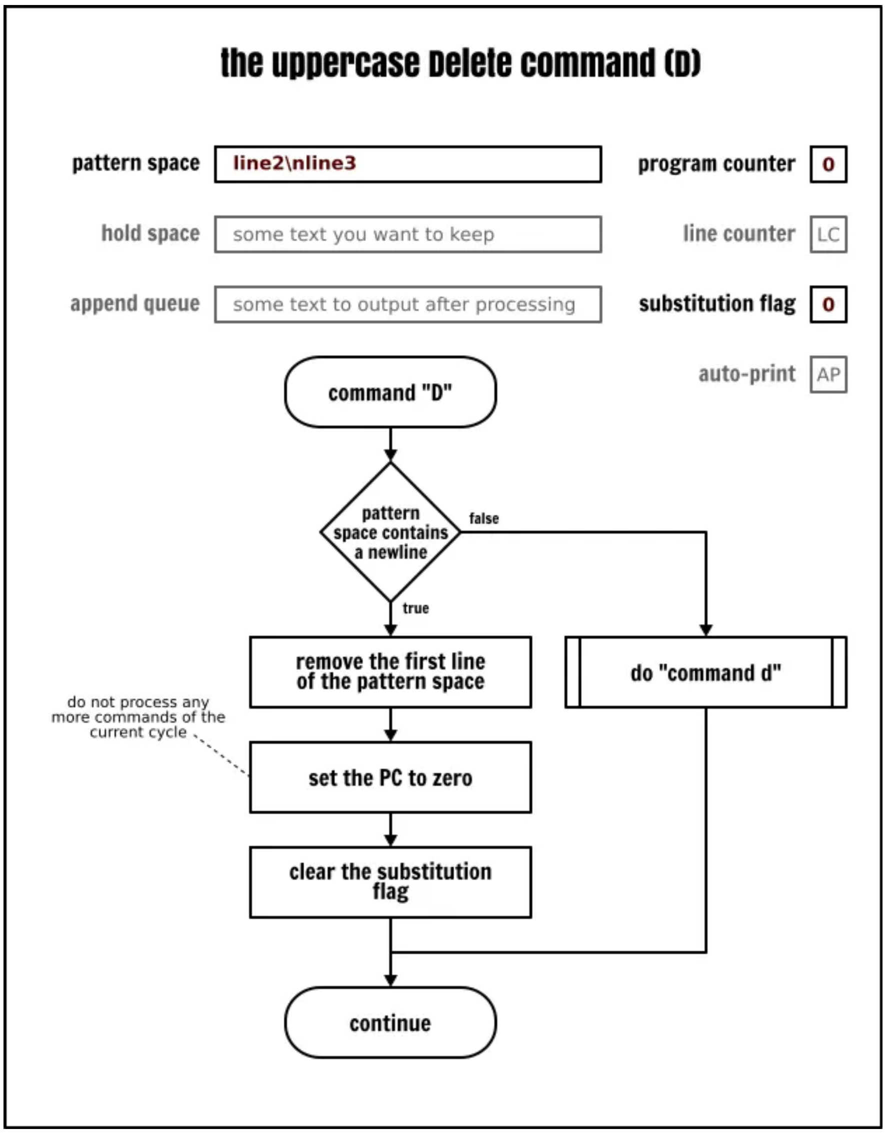
print, delete and next revisited

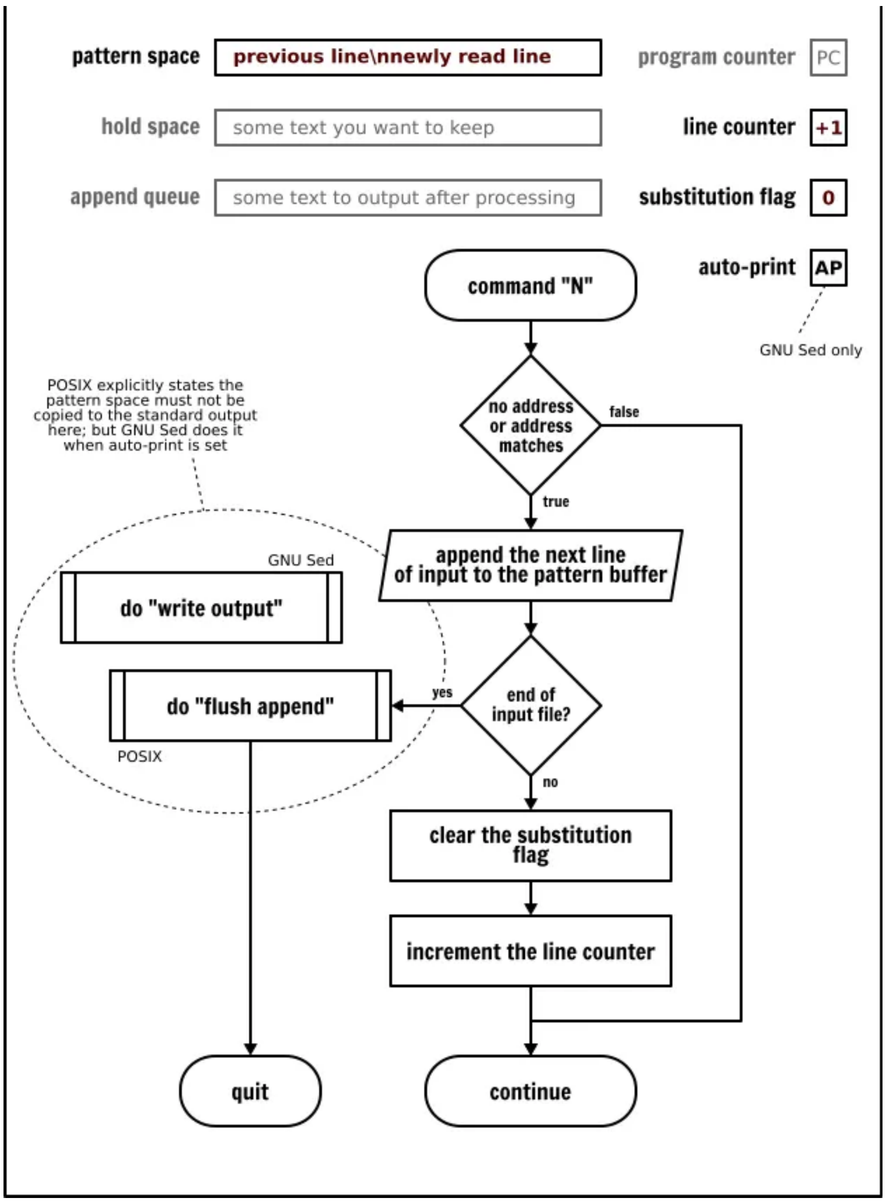
Now you've gained more familiarities with the hold space, let me go back on the `print`, `delete` and `next` commands. We already talked about the lower case `p`, `d` and `n` commands. But they also have an upper case version. As it seems to be a convention with Sed, the uppercase version of those commands will be related to multi-line buffers:

`P`
: print the content of the pattern space *up to the first newline*

`D`
: delete the content of the pattern space *up and including the first newline* then restart a cycle with the remaining text *without* reading any new input

`N`
: *read and append a new line of input* to the pattern space using the newline character as a separator between the old and new data. Continue the execution of the current cycle.


the uppercase Delete command (D)
pattern space
line2\nline3
program counter
0
hold space
some text you want to keep
line counter
LC
append queue
some text to output after processing
substitution flag
0
auto-print
AP
command "D"
pattern space contains a newline
false
true
remove the first line of the pattern space
do "command d"
do not process any more commands of the current cycle
set the PC to zero
clear the substitution flag
continue


pattern space
previous line\nnewly read line
program counter
PC
hold space
some text you want to keep
line counter
+1
append queue
some text to output after processing
substitution flag
0
auto-print
AP
GNU Sed only
command "N"
POSIX explicitly states the pattern space must not be copied to the standard output here; but GNU Sed does it when auto-print is set
no address or address matches
false
true
append the next line of input to the pattern buffer
GNU Sed
do "write output"
do "flush append"
POSIX
yes
end of input file?
no
clear the substitution flag
increment the line counter
quit
continue

The main use case for those commands is to implement queues (FIFO lists (https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))). The canonical example being removing the last 5 lines from a file:

```
cat -n inputfile | sed -En -e '
 1 { N;N;N;N }     # ensure the pattern space contains *five* lines

 N                 # append a sixth line onto the queue
 P                 # Print the head of the queue
 D                 # Remove the head of the queue
'
```

As a second example, we could display input data on two columns:

```
 # Print on two columns
sed < inputfile -En -e '
 $!N                # Append a new line to the pattern space
                    # *except* on the last line of input
                    # This is a trick required to deal with
                    # inconsistencies between GNU Sed and POSIX Sed
                    # when using N on the last line of input
                    # https://www.gnu.org/software/sed/manual/sed.html#N_005fcommand_005flast_0

                    # Right pad the first field of the first line
                    # with spaces and discard rest of the line
 s/:.*\n/                    \n/
 s/:.*//            # Discard all but the first field on the second line
 s/(.{20}).*\n/\1/  # Trim and join lines
 p                  # Print result
'
```

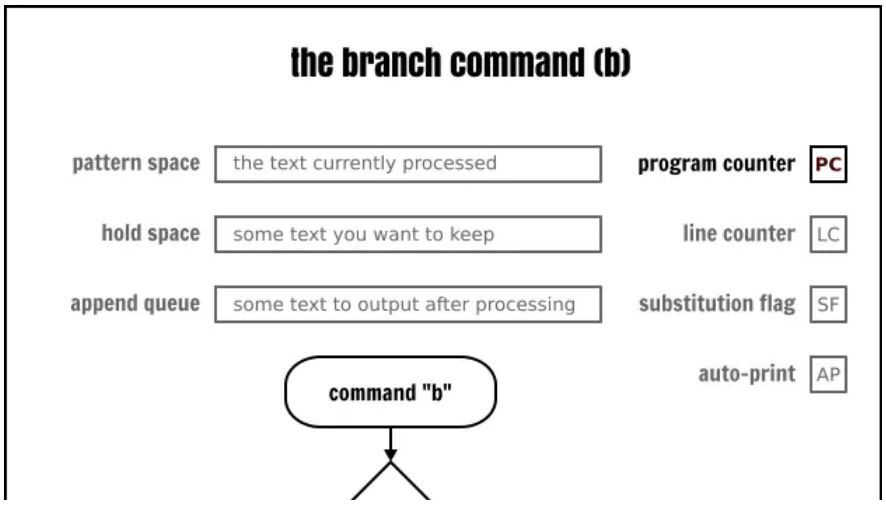
Branching

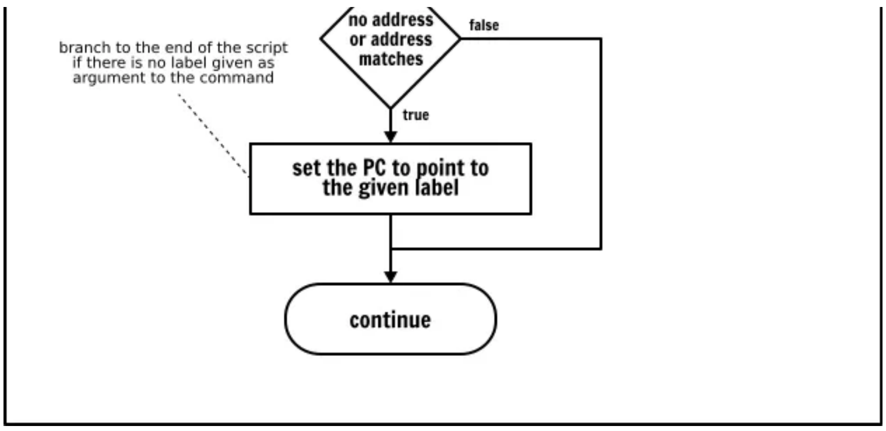
We just saw Sed has buffer capabilities through the hold space. But it also has test and branch instructions. Having both those features makes Sed a Turing complete (https://chortle.ccsu.edu/StructuredC/Chap01/struct01_5.html) language. It may sound silly, but that means you can write any program using Sed. You can do it, but that does not mean it would be an easy task, nor that the result would be particularly efficient of course.

However, don't panic. In this article, we will stay with simple examples of tests and branches. Even if these capabilities seem limited at first sight, remember some people have written http://www.catonmat.net/ftp/sed/dc.sed (http://www.catonmat.net/ftp/sed/dc.sed) [calculators], http://www.catonmat.net/ftp/sed/sedtris.sed (http://www.catonmat.net/ftp/sed/sedtris.sed) [Tetris] or many other kinds of applications using sed!

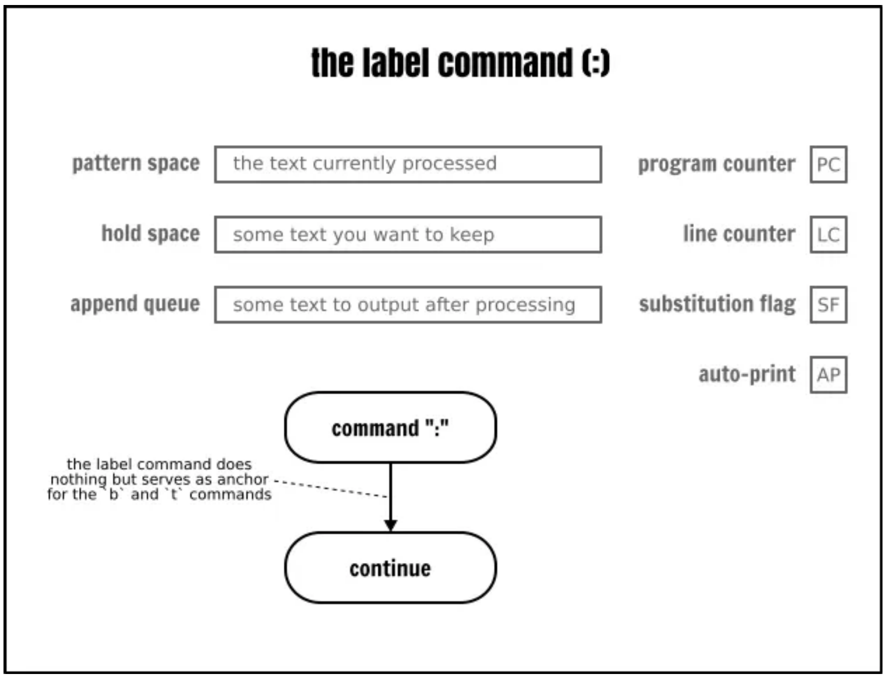
labels and branches

By some aspects, you can see Sed as a very limited assembly language. So you won't find high-level "for" or "while" loops or "if … else" statements, but you can implement them using branches.




no address or address matches
false
true
branch to the end of the script if there is no label given as argument to the command
set the PC to point to the given label
continue

If you take a look at the flowchart describing the Sed execution model at the top of this article, you can see Sed automatically increments the program counter, resulting in the execution of the commands in the program's instructions order. However, using branch instructions, you can break that sequential flow by continuing the execution with any command of your choice in the program. The destination of a jump is explicitly defined using a label.


the label command (:)
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command ":"
the label command does nothing but serves as anchor for the `b` and `t` commands
continue

Here is an example:

```
  echo hello | sed -ne '
   :start      # Put the "start" label on that line of the program
   p           # Print the pattern buffer
   b start     # Continue execution at the :start label
 ' | less
```

The behavior of that Sed program is very close to the `yes (https://linux.die.net/man/1/yes)` command: it takes a string and produces an infinite stream of lines containing that string.

Branching to a label as we did bypass all Sed automatic features: it does not read any input, nor print anything, nor update any buffer. It just jumps to a different instruction instead of executing the next one in the source program order.

Worth mentioning without any label specified as an argument, the branch command (`b`) will branch to the end of the program. So Sed will start a new cycle. This may be useful to bypass some instructions and thus may be used as an alternative to blocks:

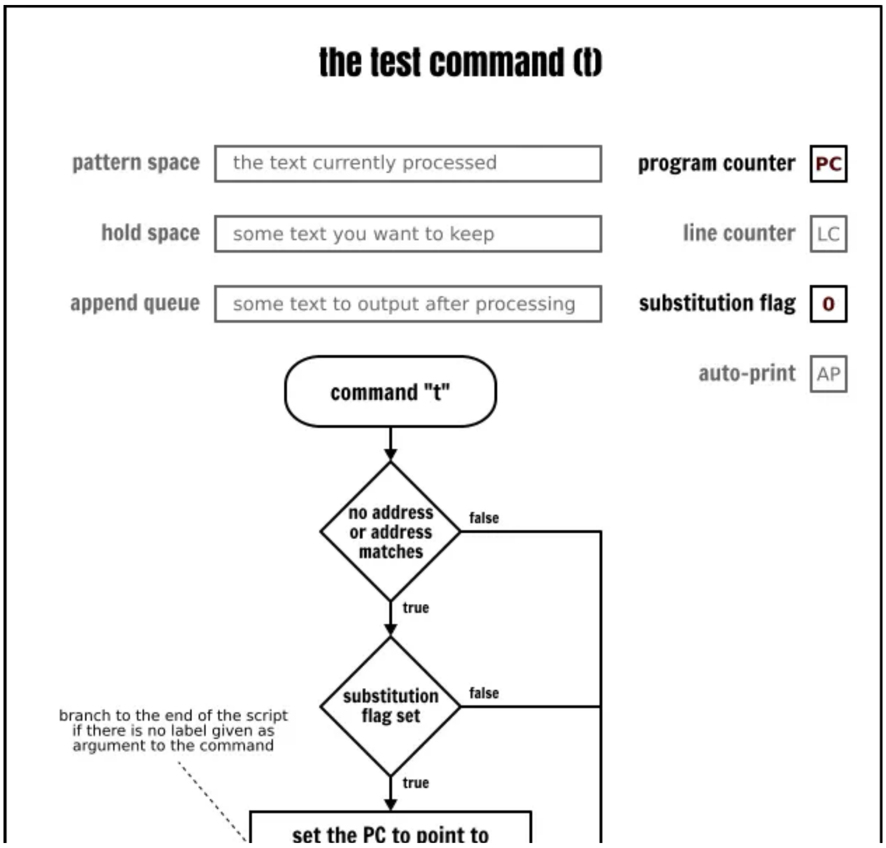
```
 cat -n inputfile | sed -ne '
/usb/!b
/daemon/!b
p
'
```

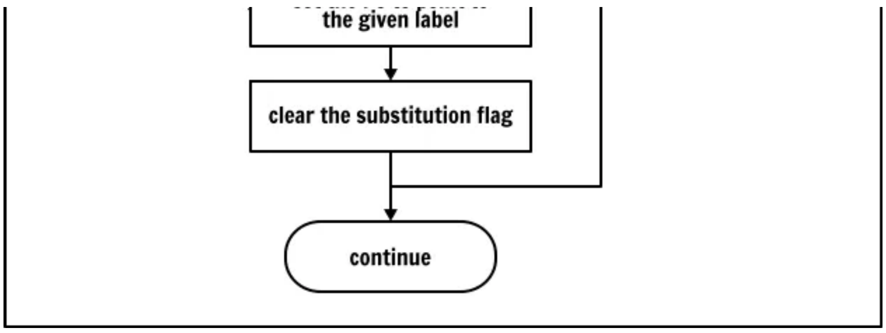
Conditional branch

Until now, we've seen the so-called unconditional branches, even if the term is somewhat misleading in this context since Sed commands are always conditional based on their optional address.

However, in a more traditional sense, an *unconditional* branch is a branch that, when executed, will *always jump* to the specified destination, whereas a *conditional* branch *may or may not jump* to the specified instruction depending on the current state of the system.

Sed has only one conditional instruction, the test (`t`) command. It jumps to a different instruction only if a substitution was executed since the start of the current cycle or since the previous conditional branch. More formally, the test command will branch only if the *substitution flag* is set.


the test command (t)
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
0
auto-print
AP
command "t"
no address or address matches
false
true
substitution flag set
false
true
branch to the end of the script if there is no label given as argument to the command
set the PC to point to



With the test instruction, you can easily perform loops in a Sed program. As a practical example, you can use that to pad lines to a certain length (something you can't do with regex only):

```
 # Center text
cut -d: -f1 inputfile | sed -Ee '
  :start
  s/^(.{,19})$/ \1 /    # Pad lines shorter than 20 chars with
                        # a space at the start and another one
                        # at the end
  t start               # Go back to :start if we added a space
  s/(.{20}).*/| \1 |/   # Keep only the first 20 char of the line
                        # to fix the off-by-one error caused by
                        # odd lines
'
```

If you carefully read the previous example, you've noticed I cheated a little bit by using the cut command to pre-process the data before feeding them to sed.

We can, however, perform the same task using only sed at the cost of a small modification to the program:

```
 cat inputfile | sed -Ee '
  s/:.*//               # Remove all but the first field
  t start
  :start
  s/^(.{,19})$/ \1 /    # Pad lines shorter than 20 chars with
                        # a space at the start and another one
                        # at the end
  t start               # Go back to :start if we added a space
  s/(.{20}).*/| \1 |/   # Keep only the first 20 char of the line
                        # to fix the off-by-one error caused by
                        # odd lines
'
```

In the above example, you may be surprised by the following construct:

```
 t start
:start
```

At first sight, the branch here seems useless since it will jump to the instruction that would have been executed anyway. However, if you read the definition of the `test` command attentively, you will see it branches only *if there was a substitution* since the start of the current cycle or *since the previous test command was executed*. In other words, the test instruction has the side effect of clearing the *substitution flag*. This is exactly the purpose of the code fragment above. This is a trick you will often see in Sed programs containing conditional branches to avoid false positive when using several substitutions commands.

I agree though it wasn't absolutely mandatory here to clear the substitution flag since the specific substitution command I used is idempotent once it has padded the string to the right length. So one extra iteration will not change the result. However, look at that second example now:

```
 # Classify user accounts based on their login program
cat inputfile | sed -Ene '
  s/^/login=/
  /nologin/s/^/type=SERV /
  /false/s/^/type=SERV /
  t print
  s/^/type=USER /
  :print
  s/:.*//p
'
```

My hope here was to tag the user accounts with either "SERV" or "USER" depending on the configured default login program. If you ran it, you've seen the "SERV" tag as expected. However, no trace of the "USER" tag in the output. Why? Because the `t print` instruction will *always* branch since whatever was the content of the line, the *substitution flag* was set by the very first substitution command of the program. Once set, the flag remains set until a next line is read— or until the next test command. And gives us the solution to fix that program:

```
 # Classify user accounts based on the login program
cat inputfile | sed -Ene '
  s/^/login=/

  t classify # clear the "substitution flag"
  :classify

  /nologin/s/^/type=SERV /
  /false/s/^/type=SERV /
  t print
  s/^/type=USER /
  :print
  s/:.*//p
'
```

Handling verbatim text

Sed is a text editor. A non-interactive one. But a text editor nevertheless. It wouldn't be complete without some facility to insert literal text in the output. I'm not a big fan of that feature since I find the syntax awkward (even by the Sed standards), but sometimes you can't avoid it.

> **READ [Getting Started With AWK Command [Beginner's Guide]](https://linuxhandbook.com/awk-command-tutorial/)**
> **(https://linuxhandbook.com/awk-command-tutorial/)**

In the strict POSIX syntax, all the three commands to change (`c`), insert (`i`) or append (`a`) some literal text to the output follow the same specific syntax: the command letter is followed by a backslash, and the text to insert start on the next line of the script:

```
 head -5 inputfile | sed '
1i\
# List of user accounts
$a\
# end
'
```

To insert multiple lines of text, you must end each of them with a backslash:

```
 head -5 inputfile | sed '
1i\
# List of user accounts\
# (users 1 through 5)
$a\
# end
'
```

Some Sed implementations, like GNU Sed, makes the newline after the initial backslash optional, even when forced in `--posix` mode. I didn't find anything in the standard that authorizes that alternate syntax. So use it at your own risks if portability is a premium (or leave a comment if I missed that feature in the specifications!):

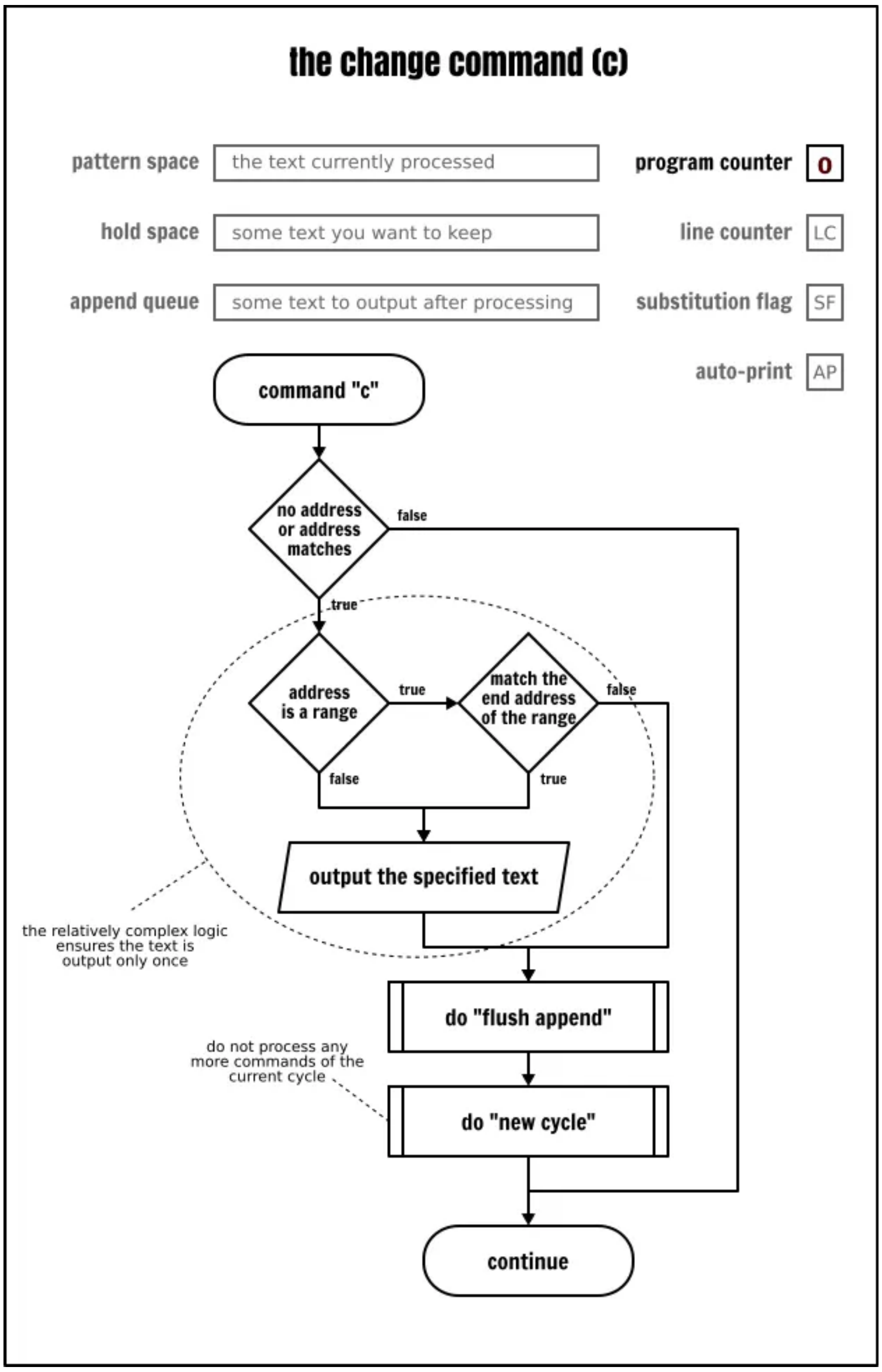
```
 # non-POSIX syntax:
head -5 inputfile | sed -e '
1i \# List of user accounts
$a\# end
'
```

Some Sed implementation also makes the initial backslash completely optional. Since this is, without any doubt this time, a vendor-specific extension to the POSIX specifications, I let you check the manual for the version of sed you use to check if it supports that syntax.

After that quick overview, let's review those commands in more details now, starting with the change command I didn't have presented yet.

The change command

The change command (`c\`) deletes the pattern space and starts a new cycle just like the `d` command. The only difference is the user provided text is written on the output when the command is executed.


the change command (c)
pattern space
the text currently processed
program counter
0
hold space
some text you want to keep
line counter
LC
append queue
some text to output after processing
substitution flag
SF
auto-print
AP
command "c"
no address or address matches
false
true
address is a range
true
match the end address of the range
false
false
true
output the specified text
the relatively complex logic ensures the text is output only once
do "flush append"
do not process any more commands of the current cycle
do "new cycle"
continue

```
 cat -n inputfile | sed -e '
/systemd/c\
# :REMOVED:
s/:.*// # This will NOT be applied to the "changed" text
'
```

If the change command is associated with a range address, the text is output only once, when reaching the last line of the range. Which somehow makes it an exception to the convention a Sed command is repeatedly applied to all lines of its range address:

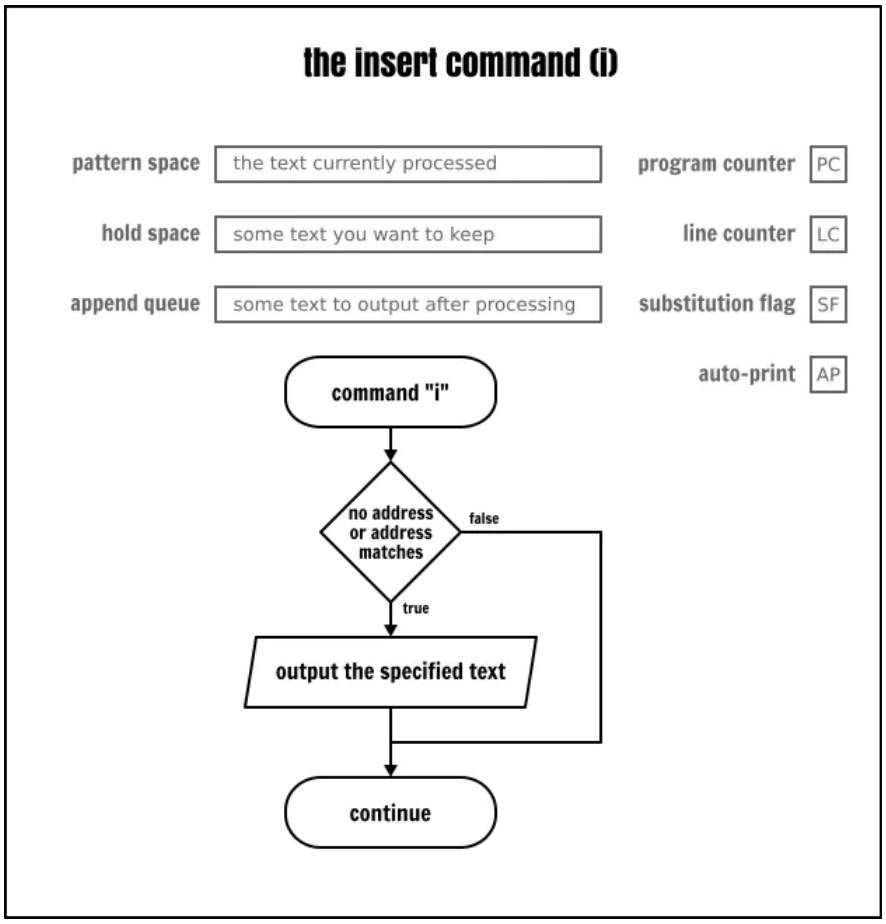
```
 cat -n inputfile | sed -e '
19,22c\
# :REMOVED:
s/:.*// # This will NOT be applied to the "changed" text
'
```

As a consequence, if you want the change command to be repeated for every line in a range, you have no other choice than wrapping it inside a block:

```
 cat -n inputfile | sed -e '
19,22{c\
# :REMOVED:
}
s/:.*// # This will NOT be applied to the "changed" text
'
```

The insert command

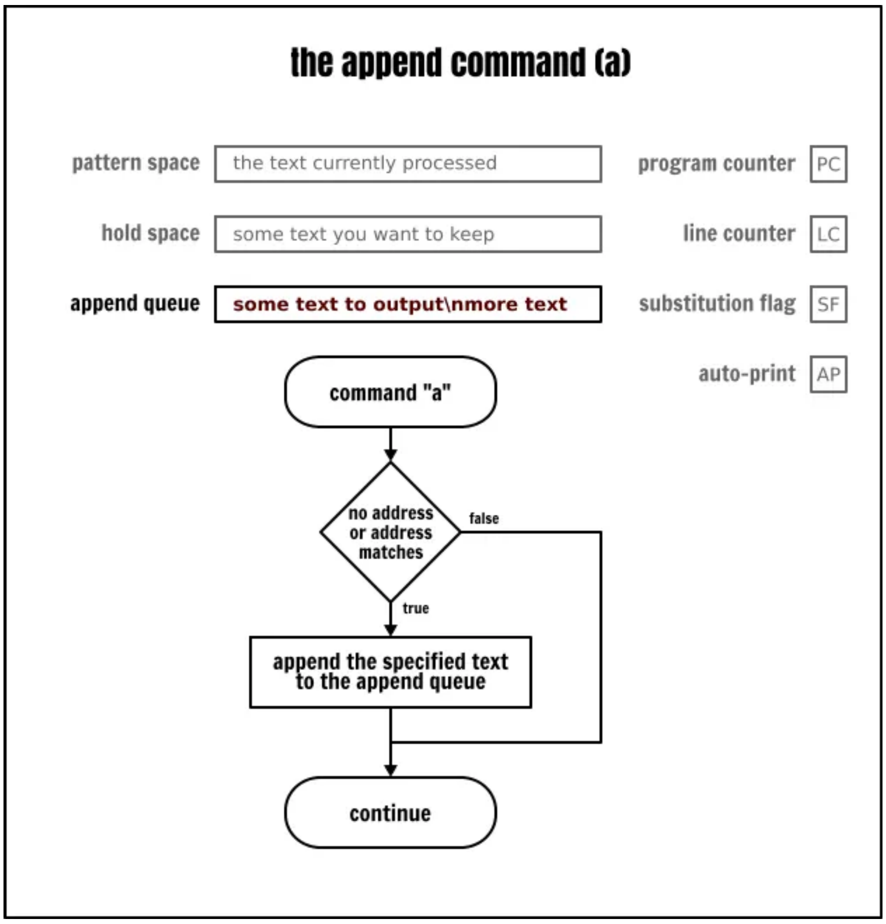
The insert command (`i\`) immediately print the user-provided text on the output. It does not alter in any way the program flow or buffer content.



```
 # display the first five user names with a title on the first row
sed < inputfile -e '
1i\
USER NAME
s/:.*//
5q
'
```

The append command

The append command (`a\`) queues some text to be displayed when the next line of input will be read. The text is output at the end of the current cycle (including at the end of the program) or when a new line is read from the input using either the `n` or `N` command.


the append command (a)
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output\nmore text
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command "a"
no address or address matches
false
true
append the specified text to the append queue
continue

Same example as above, but inserts this time a footer instead of a header:

```
 sed < inputfile -e '
5a\
USER NAME
s/:.*//
5q
'
```

The read command

There is a fourth command to insert literal content into the output stream: the read command (r). It works exactly like the append command, but instead of taking the text hardcoded from the Sed script itself, it will write the content of a file on the output.

The read command only schedules the file to be read. That latter is effectively read when the append queue is flushed. Not when the read command is executed. This may have implications if there are concurrent accesses to the file to be read, if that file is not a regular file (for example, if it's a character device or a named pipe), or if the file is modified during processing.

As an illustration, if you use the write command we will see in detail in the next section together with the read command to write and re-read from a temporary file, you may obtain some creative results (using a French equivalent of the Shiritori (https://en.wikipedia.org/wiki/Shiritori) game as an illustration):

```
  printf "%s\n" "Trois p'tits chats" "Chapeau d' paille" "Paillasson" |
 sed -ne '
   r temp
   a\
   ----
   w temp
 '
```

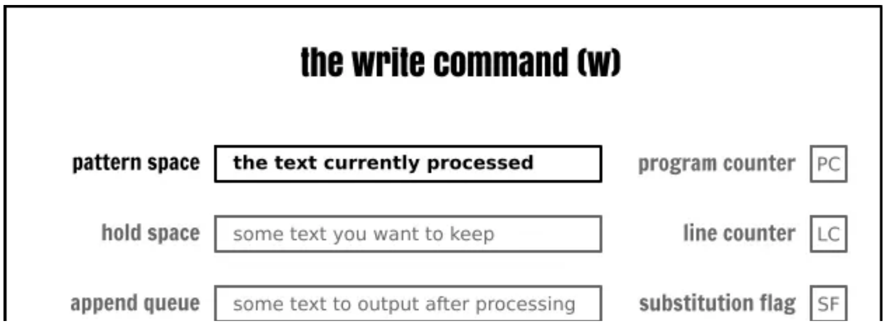
We've now ended the list of the Sed commands dedicated to the insertion of literal text in the stream output. My last example was mostly for fun, but since I mentioned there the write command, that makes a perfect transition with the next section where we will see how to write data to an external file from Sed.

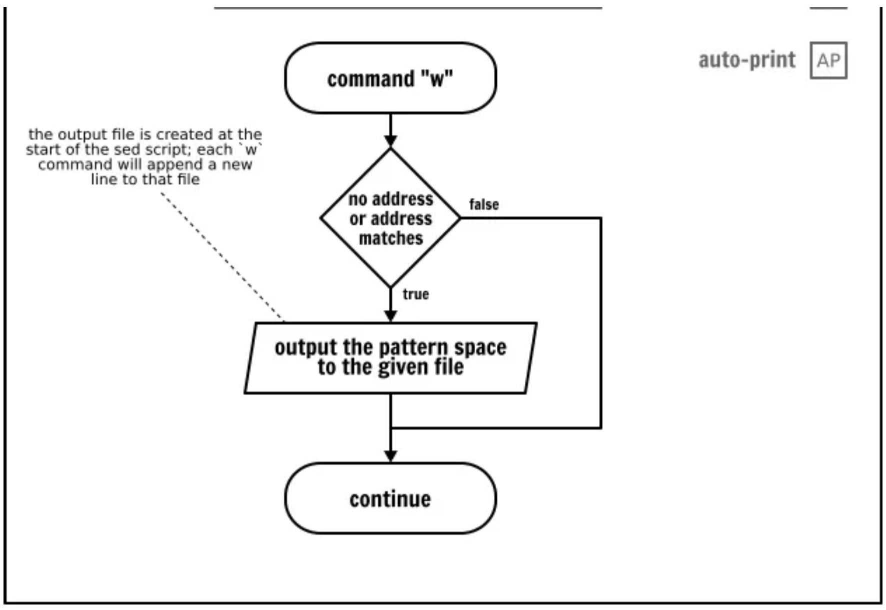
Alternate output

Sed is designed with the idea all text transformations will end-up being written on the standard output of the process. However, Sed also has some provision to send data to alternate destinations. You have two ways to do that: using the dedicated write command, or by adding the write flag to a substitution command.

The write command

The write command (w) appends the content of the pattern space to the given destination file. POSIX requires the destination file to be created by Sed before it starts processing any input data. If the file already exists, it is overwritten.




command "w"
the output file is created at the start of the sed script; each `w` command will append a new line to that file
no address or address matches
false
true
output the pattern space to the given file
continue
auto-print
AP

As a consequence, even if you never really write to a file, it will be created anyway. For example, the following Sed program will create/overwrite the “output” file, even if the write command is never executed:

```
 echo | sed -ne '
q         # immediately quit
w output  # this command is never executed
'
```

You can have several write commands referencing the same destination file. All write commands at the destination of the same file will append content to that file (more or less in the same manner as the `>>` shell redirection). :

```
 sed < inputfile -ne '
/:\/bin\/false$/w server
/:\/usr\/sbin\/nologin$/w server
w output
'
cat server
```

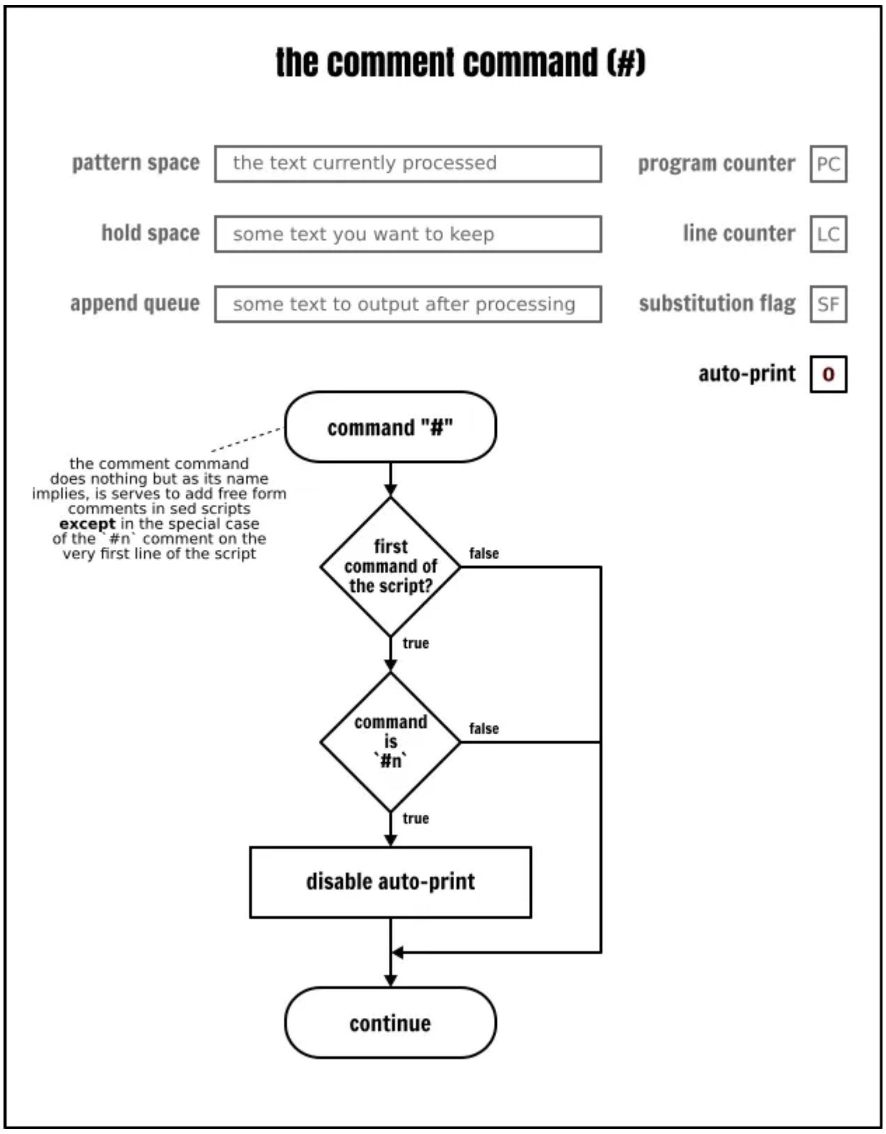
The substitution command `write` flag

A long time ago now, we had seen the substitution command has the `p` option for the common use case of printing the pattern space after a substitution. In a very similar manner it also has a `w` option to write the pattern space to a file after a substitution:

```
 sed < inputfile -ne '
s/:.*\/nologin$//w server
s/:.*\/false$//w server
'
cat server
```

Comments

I already used them countless times, but I never took the time to introduce them formally, so, let's fix that: like in most programming languages, a *comment* is a way to add free-form text the software will not try to interpret. The Sed syntax being rather cryptic, I can't insist enough on the need to comment your scripts. Otherwise, they will be hardly understandable by anyone except their author.


the comment command (#)
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
0
command "#"
the comment command does nothing but as its name implies, is serves to add free form comments in sed scripts except in the special case of the `#n` comment on the very first line of the script
first command of the script?
false
true
command is `#n`
false
true
disable auto-print
continue

However, like many other parts of Sed, comments have their own share of subtleties. First, and the most important, comments are not a syntactic construct, but are full-fledged commands in Sed. A do-nothing ("no-op") command, but a command anyway. At least, that is how there are defined by POSIX. So, strictly speaking, they should only be allowed where other commands are allowed.

Most Sed implementation relaxes that requirement by allowing inline commands as I used them all over the place in that article.

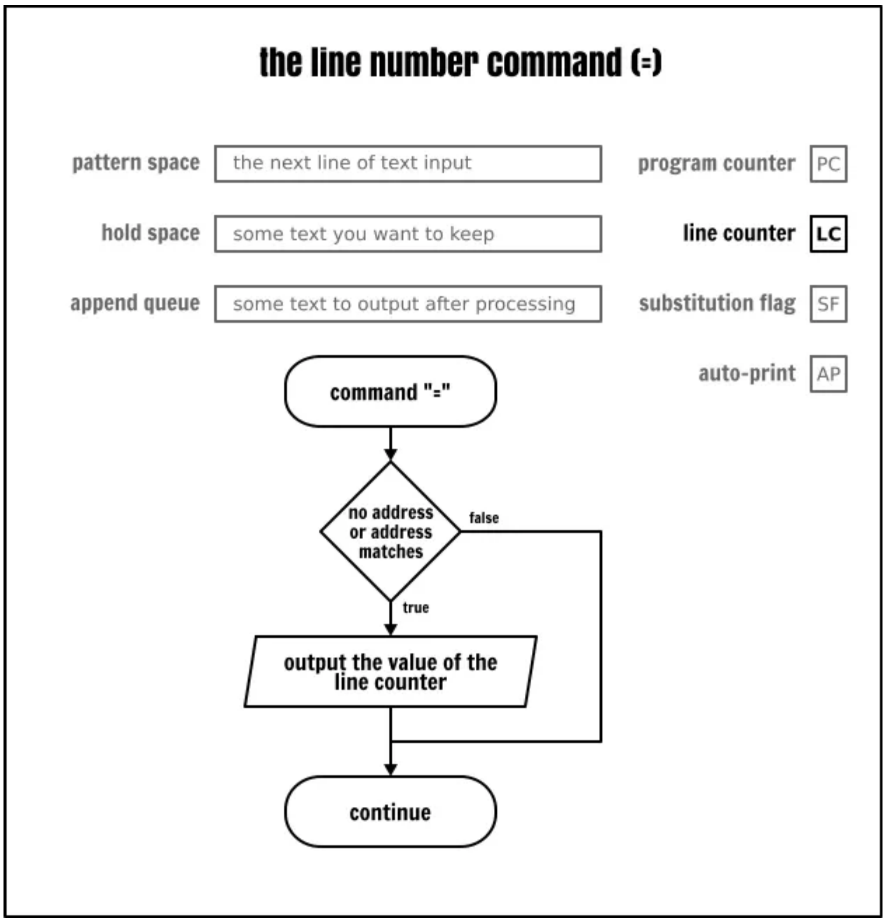
To close on that topic, worth mentioning the very special case of the `#n` comment (an octothorpe followed by the letter n without any space). If that exact comment if found on the very first line of a script, Sed should switch to quiet mode (i.e., clearing the auto-print flag) just like if the `-n` option was specified on the command line.

The commands you will rarely need

Now, we have reviewed the commands that will allow you to write 99.99% of your scripts. But this tour wouldn't be exhaustive if I didn't mention the last remaining Sed commands. I left them aside until now because I rarely needed them. But maybe did you have on your side examples of practical use cases where you find them useful. If that is the case do not hesitate to share that with us using the comment section.

The line number command

The `=` command writes on the standard output the number of lines currently read by Sed, that is the content of the line counter register. There is no way to capture that number in one of the Sed buffers, nor to format the output. Two limitations that severely reduce the usefulness of that command.

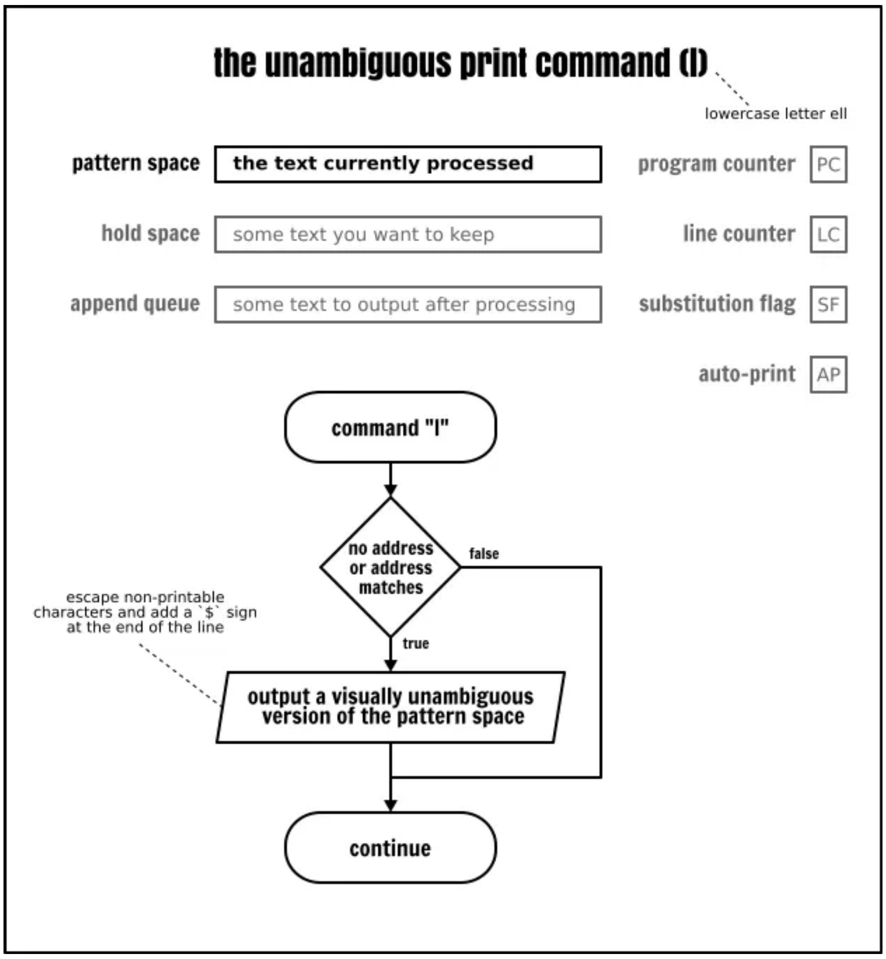

the line number command (=)
pattern space
the next line of text input
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command "="
no address or address matches
false
true
output the value of the line counter
continue

Remember in strict POSIX compliance mode, when several input files are given on the command line, Sed does not reset that counter but continue to increment it just like if all the files where concatenated. Some Sed implementations like GNU Sed have options to reset the counter after each input file.

The unambiguous print command

The `l` (lowercase letter ell) is similar to the print command (`p`), but the content of the pattern space will be written in an unambiguous form. To quote the POSIX standard (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html):

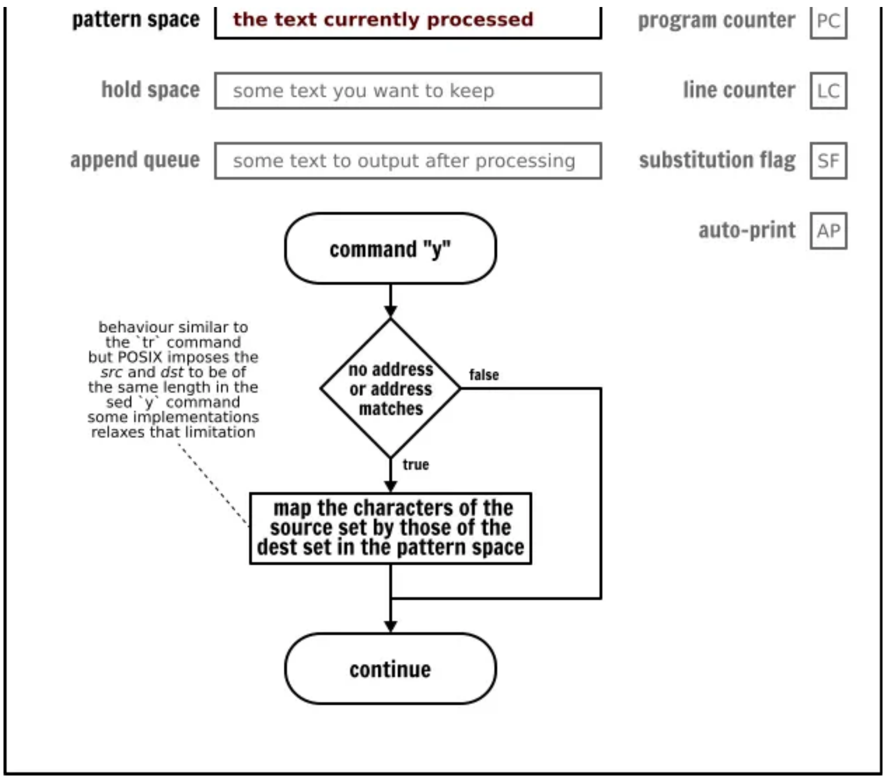
- *The characters listed in XBD Escape Sequences and Associated Actions ('\\', '\a', '\b', '\f', '\r', '\t', '\v') shall be written as the corresponding escape sequence; the '\n' in that table is not applicable. Non-printable characters not in that table shall be written as one three-digit octal number (with a preceding <backslash>) for each byte in the character (most significant byte first). Long lines shall be folded, with the point of folding indicated by writing a <backslash> followed by a <newline>; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line shall be marked with a '$'.*


the unambiguous print command (l)
lowercase letter ell
pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command "l"
no address or address matches
false
true
escape non-printable characters and add a `$` sign at the end of the line
output a visually unambiguous version of the pattern space
continue

I suspect this command was once used to exchange data over non 8-bits clean channels (https://en.wikipedia.org/wiki/8-bit_clean). As of myself, I never used it for anything else than for debugging purposes.

The transliterate command

The transliterate (y) command allows mapping characters of the pattern space from a source set to a destination set. It is quite similar to the `tr (https://linux.die.net/man/1/tr)` command, although more limited.


pattern space
the text currently processed
hold space
some text you want to keep
append queue
some text to output after processing
program counter
PC
line counter
LC
substitution flag
SF
auto-print
AP
command "y"
behaviour similar to the `tr` command but POSIX imposes the *src* and *dst* to be of the same length in the sed `y` command some implementations relaxes that limitation
no address or address matches
false
true
map the characters of the source set by those of the dest set in the pattern space
continue

```
 # The `y` c0mm4nd 1s for h4x0rz only
sed < inputfile -e '
  s/:.*//
  y/abcegio/48<3610/
'
```

While the transliterate command syntax bears some resemblances with the substitution command syntax, it does not accept any option after the replacement string. The transliteration is always global.

Beware that the transliterate command requires all the characters both in the original and destination set to be given verbatim. That means the following Sed program does not do what you might think at first sight:

```
 # BEWARE: this doesn't do what you may think!
sed < inputfile -e '
  s/:.*//
  y/[a-z]/[A-Z]/
'
```

The last word

```
 # What will this do?
# Hint: the answer is not far away...
sed -E '
  s/.*\W(.*)/\1/
  h
  ${ x; p; }
  d' < inputfile
```

I can't believe we did it! We've reviewed all the Sed commands. If you reached that point, you deserve congratulations, especially if you took the time to try the different examples on your system!

As you've seen, Sed is a complex beast, not only because of its sparse syntax but also because of all the various corner cases or subtle differences in the command behavior. No doubt, we can blame historical reasons for that. Despite these drawbacks, it is a powerful tool and like AWK command (https://linuxhandbook.com/awk-command-tutorial/), even today remains one of the most useful commands of the Unix toolbox.

If it is time for me to conclude that article, I wouldn't do it without first asking you a favor: please, share with us using the command section your most favorite or creative piece of Sed script. If we have enough of them, we could publish a compilation of those Sed gems!

About Sylvain Leroux

Engineer by Passion, Teacher by Vocation. My goals : to share my enthusiasm for what I teach and prepare my students to develop their skills by themselves. You can find me on my website (https://www.yesik.it/) as well.

2 comments Oldest comments first

carls

Labor of love. Many thanks and deepest admiration.

Vote: Share

Taf Bulewa

Many thanks to you, Mr. Leroux. This material is invaluable.

Vote: Share