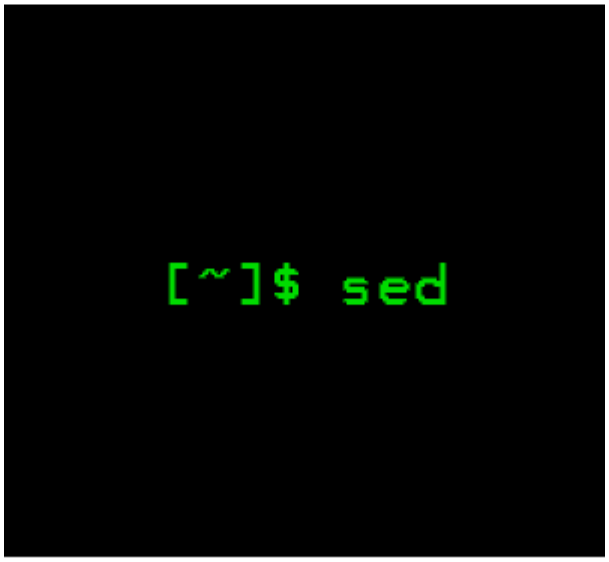
Linux and Unix sed command



http://www.computerhope.com

About sed

sed, short for "stream editor", allows you to filter and transform text.

Description

sed is a *stream editor*. A stream editor is used to perform basic text transformations on an input stream (a file, or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as ed), **sed** works by making only one pass over the input(s), and is consequently more efficient. But it is **sed**'s ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

sed syntax

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

If you do not specify *INPUTFILE*, or if *INPUTFILE* is "**-**", **sed** filters the contents of the standard input. The script is actually the first non-option parameter, which **sed** specially considers a script and not an input file if and only if none of the other options specifies a script to be executed (that is, if neither of the **-e** and **-f** options is specified).

Options

-n, **--quiet**, **--silent**	Suppress automatic printing of pattern space.
-e *script*, **--expression**=*script*	Add the script *script* to the commands to be executed.
-f *script-file*, **--file**=*script-file*	Add the contents of *script-file* to the commands to be executed.
--follow-symlinks	Follow symlinks when processing in place.
-i[*SUFFIX*], **--in-place**[=*SUFFIX*]	Edit files in place (this makes a backup with file extension *SUFFIX*, if *SUFFIX* is supplied).
-l *N*, **--line-length**=*N*	Specify the desired line-wrap length, *N*, for the "**l**" command.
--POSIX	Disable all GNU extensions.
-r, **--regexp-extended**	Use extended regular expressions in the script.
-s, **--separate**	Consider files as separate rather than as a single continuous long stream.
-u, **--unbuffered**	Load minimal amounts of data from the input files and flush the output buffers more often.
--help	Display a help message, and exit.
--version	Output version information, and exit.

About sed Programs

A **sed** program consists of one or more **sed** commands, passed in by one or more of the **-e**, **-f**, **--expression**, and **--file** options, or the first non-option argument if none of these options are used. This documentation frequently refers to "the" **sed** script; this should be understood to mean the in-order catenation of all of the scripts and script-files passed in.

Commands within a script or script-file can be separated by semicolons ("**;**") or newlines (ASCII code 10). Some commands, due to their syntax, cannot be followed by semicolons working as command separators and thus should be terminated with newlines or be placed at the end of a script or script-file. Commands can also be preceded with optional non-significant whitespace characters.

Each **sed** command consists of an optional address or address range (for instance, line numbers specifying what part of the file to operate on; see Selecting Lines for details), followed by a one-character command name and any additional command-specific code.

How sed Works

sed maintains two data buffers: the active *pattern space*, and the auxiliary *hold space*. Both are initially empty.

sed operates by performing the following cycle on each line of input: first, **sed** reads one line from the input stream, removes any trailing newline, and places it in the pattern space. Then commands are executed; each command can have an address associated to it: addresses are a kind of condition code, and a command is only executed if the condition is verified before the command is to be executed.

When the end of the script is reached, unless the **-n** option is in use, the contents of pattern space are printed out to the output stream, adding back the trailing newline if it was removed. Then the next cycle starts for the next input line.

Unless special commands (like '**D**') are used, the pattern space is deleted between two cycles. The hold space, on the other hand, keeps its data between cycles (see commands **‘h’**, **‘H’**, **‘x’**, **‘g’**, **‘G’** to move data between both buffers).

Selecting Lines With sed

Addresses in a **sed** script can be in any of the following forms:

number	Specifying a line number will match only that line in the input. (Note that sed counts lines continuously across all input files unless **-i** or **-s** options are specified.)
first~step	This GNU extension of **sed** matches every *step*th line starting with line *first*. In particular, lines will be selected when there exists a non-negative *n* such that the current line-number equals *first* + (*n* * *step*). Thus, to select the odd-numbered lines, one would use **1~2**; to pick every third line starting with the second, ‘**2~3**’ would be used; to pick every fifth line starting with the tenth, use ‘**10~5**’; and ‘**50~0**’ is just another way of saying **50**.
$	This address matches the last line of the last file of input, or the last line of each file when the **-i** or **-s** options are specified.
/regexp/	This will select any line which matches the regular expression *regexp*. If *regexp* itself includes any "/" characters, each must be escaped by a backslash ("\\"). The empty regular expression ‘//’ repeats the last regular expression match (the same holds if the empty regular expression is passed to the s command). Note that modifiers to regular expressions are evaluated when the regular expression is compiled, thus it is invalid to specify them together with the empty regular expression.
\%regexp%	(The *%* may be replaced by any other single character.)

This also matches the regular expression *regexp*, but allows one to use a different delimiter than "/". This is particularly useful if the *regexp* itself contains a lot of slashes, since it avoids the tedious escaping of every "/". If *regexp* itself includes any delimiter characters, each must be escaped by a backslash ("\").

*/regexp/***I**

*\%regexp%***I**

The **I** modifier to regular-expression matching is a GNU extension which causes the *regexp* to be matched in a case-insensitive (as opposed to case-sensitive) manner.

*/regexp/***M**

*\%regexp%***M**

The **M** modifier to regular-expression matching is a GNU **sed** extension which causes **^** and **$** to match respectively (in addition to the normal behavior) the empty string after a newline, and the empty string before a newline. There are special character sequences ("\`" and "\'") which always match the beginning or the end of the buffer. **M** stands for multi-line.

If no addresses are given, then all lines are matched; if one address is given, then only lines matching that address are matched.

An address range can be specified by specifying two addresses separated by a comma ("**,**"). An address range matches lines starting from where the first address matches, and continues until the second address matches (inclusively).

If the second address is a *regexp*, then checking for the ending match will start with the line following the line which matched the first address: a range will always span at least two lines (except of course if the input stream ends).

If the second address is a number less than (or equal to) the line matching the first address, then only the one line is matched.

GNU **sed** also supports some special two-address forms; all these are GNU extensions:

0,/*regexp*/	A line number of **0** can be used in an address specification like **0,**/*regexp*/ so that **sed** will try to match *regexp* in the first input line too. In other words, **0,**/*regexp*/ is similar to **1,**/*regexp*/, except that if *addr2* matches the very first line of input the **0,**/*regexp*/ form will consider it to end the range, whereas the **1,**/*regexp*/ form will match the beginning of its range and hence make the range span up to the second occurrence of the regular expression. Note that this is the only place where the **0** address makes sense; there is no "0th" line, and commands which are given the **0** address in any other way will give an error.
addr1,+*N*	Matches *addr1* and the *N* lines following *addr1*.
addr1,~*N*	Matches *addr1* and the lines following *addr1* until the next line whose input line number is a multiple of *N*.

Appending the **!** character to the end of an address specification negates the sense of the match. That is, if the **!** character follows an address range, then only lines which do not match the address range will be selected. This also works for singleton addresses, and, perhaps perversely, for the null address.

Overview Of Regular Expression Syntax

To know how to use **sed**, you should understand regular expressions ("regexp" for short). A regular expression is a pattern that is matched against a subject string from left to right. Most characters are ordinary: they stand for themselves in a pattern, and match the corresponding characters in the subject. As a simple example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of special characters, which do not stand for themselves but instead are interpreted in some special way. Here is a brief description of regular expression syntax as used in **sed**:

char	A single ordinary character matches itself.
*	Matches a sequence of zero or more instances of matches for the preceding regular expression, which must be an ordinary character, a special character preceded by "\", a "**.**", a grouped regexp (see below), or a bracket expression. As a GNU extension, a postfixed regular expression can also be followed by "*"; for example, **a**** is equivalent to **a***. POSIX 1003.1-2001 says that * stands for itself when it appears at the start of a regular expression or subexpression, but many nonGNU implementations do not support this, and portable scripts should instead use "*" in these contexts.
\+	Like *, but matches one or more. It is a GNU extension.
\?	Like *, but only matches zero or one. It is a GNU extension.
\{*i*\}	Like *, but matches exactly *i* sequences (*i* is a decimal integer; for compatibility, you should keep it between **0** and **255**, inclusive).
\{*i*,*j*\}	Matches between *i* and *j*, inclusive, sequences.
\{*i*,\}	Matches more than or equal to *i* sequences.
\(*regexp*\)	Groups the inner regexp as a whole; this is used to: » Apply postfix operators, like **\(abcd\)***: this will search for zero or more whole sequences of '**abcd**', while

abcd* would search for ‘**abc**’ followed by zero or more occurrences of ‘**d**’. Note that support for **\(abcd\)*** is required by POSIX 1003.1-2001, but many non-GNU implementations do not support it and hence it is not universally portable.

» Use back references (see below).

.	Matches any character, including a newline.
^	Matches the null string at beginning of the pattern space, i.e. what appears after the ^ must appear at the beginning of the pattern space. In most scripts, pattern space is initialized to the content of each line. So, it is a useful simplification to think of **^#include** as matching only lines where ‘**#include**’ is the first thing on line—if there are spaces before, for example, the match fails. This simplification is valid as long as the original content of pattern space is not modified, for example with an **s** command. ^ acts as a special character only at the beginning of the regular expression or subexpression (that is, after \**(** or \\\|). Portable scripts should avoid ^ at the beginning of a subexpression, though, as POSIX allows implementations that treat ^ as an ordinary character in that context.
$	It is the same as ^, but refers to end of pattern space. **$** also acts as a special character only at the end of the regular expression or subexpression (that is, before \**)** or \\\|), and its use at the end of a subexpression is not portable.

[*list***]**

[^*list***]**

Matches any single character in *list*: for example, **[aeiou]** matches all vowels. A *list* may include sequences like *char1***-***char2*, which matches any character between *char1* and *char2*. For examble, **[b-e]** matches any of the characters **b**, **c**, **d**, or **e**.

A leading **^** reverses the meaning of *list*, so that it matches any single character not in *list*. To include **]** in the list, make it the first character (after the ^ if needed); to include **-** in the list, make it the first or last; to include **^** put it after the first character.

The characters **$**, *****, **.**, **[**, and \ are normally not special within *list*. For example, **[\*]** matches either '\' or '*****', because the \ is not special here. However, strings like **[.ch.]**, **[=a=]**, and **[:space:]** are special within *list* and represent collating symbols, equivalence classes, and character classes, respectively, and **[** is therefore special within list when it is followed by **.**, **=**, or **:**. Also, when not in **POSIXLY_CORRECT** mode, special escapes like \n and \t are recognized within *list*. See Escapes for more information.

regexp1\\|*regexp2*

Matches either *regexp1* or *regexp2*. Use parentheses to use complex alternative regular expressions. The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. This is a GNU extension.

regexp1regexp2

Matches the concatenation of *regexp1* and *regexp2*. Concatenation binds more tightly than \\|, ^, and **$**, but less tightly than the other regular expression operators.

\digit

Matches the *digit*-th \\**(**...\\**)** parenthesized subexpression in the regular expression. This is called a back reference. Subexpressions are implicity numbered by counting occurrences of \**(** left-to-right.

\n

Matches the newline character.

\char	Matches *char*, where *char* is one of **$**, *****, **.**, **[**, \\, or **^**. Note that the only C-like backslash sequences that you can portably assume to be interpreted are **\n** and \\\\; in particular **\t** is not portable, and matches a '**t**' under most implementations of **sed**, rather than a tab character.

Note that the regular expression matcher is greedy, i.e., matches are attempted from left to right and, if two or more matches are possible starting at the same character, it selects the longest.

For example:

abcdef	Matches "**abcdef**".
a*b	Matches zero or more "**a**" characters, followed by a single "**b**". For example, "**b**" or "**aaaaaaab**".
a\?b	Matches "**b**" or "**ab**".
a\+b\+	Matches one or more "**a**" characters followed by one or more "**b**"s. "**ab**" is the shortest possible match, but other examples are "**aaaaab**", "**abbbbbb**", or "**aaaaaabbbbbbb**".
.*or **.**\+	Either of these expressions will match all of the characters in a non-empty string, but only **.*** will match the empty string.
^main.*(.*)	This matches a string starting with "**main**", followed by an opening and closing parenthesis. The "**n**", "**(**" and "**)**" need not be adjacent.
^#	This matches a string beginning with "**#**".
$	This matches a string ending with a single backslash. The regexp contains two backslashes for escaping.

\$	This matches a string consisting of a single dollar sign.
[a-zA-Z0-9]	In the C locale, this matches any ASCII letters or digits.
[^ *tab*]\+	(Here *tab* stands for a single tab character.) This matches a string of one or more characters, none of which is a space or a tab. Usually this means a word.
^\(.*\)\n\1$	This matches a string consisting of two equal substrings separated by a newline.
.\{9\}A$	This matches nine characters followed by an '**A**'.
^.\{15\}A	This matches the start of a string that contains 16 characters, the last of which is an '**A**'.

Often-Used Commands

If you use **sed** at all, you will probably want to know these commands.

#

(No addresses allowed with this command.) The # character begins a comment; the comment continues until the next newline.

If you are concerned about portability, be aware that some implementations of **sed** (which are not POSIX conformant) may only support a single one-line comment, and then only when the very first character of the script is a #.

Warning: if the first two characters of the **sed** script are **#n**, then the **-n** (no-autoprint) option is forced. If you want to put a comment in the first line of your script and that comment begins with the letter '**n**' and you do not want this behavior, then be sure to either use a capital '**N**', or place at least one space before the '**n**'.

q [exit-code]	This command only accepts a single address. Exit **sed** without processing any more commands or input. Note that the current pattern space is printed if auto-print is not disabled with the **-n** options. The ability to return an exit code from the **sed** script is a GNU **sed** extension.
d	Delete the pattern space; immediately start next cycle.
p	Print out the pattern space (to the standard output). This command is usually only used in conjunction with the **-n** command-line option.
n	If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then sed exits without processing any more commands.
{ *commands* **}**	A group of commands may be enclosed between **{** and **}** characters. This is particularly useful when you want a group of commands to be triggered by a single address (or address-range) match.

The s Command

The syntax of the **s** command (which stands for "substitute") is: '**s**/*regexp*/*replacement* /*flags*'. The / characters may be uniformly replaced by any other single character within any given **s** command. The / character (or whatever other character is used in its stead) can appear in the *regexp* or *replacement* only if it is preceded by a \ character.

The **s** command is probably the most important in **sed** and has a lot of different options. Its basic concept is simple: the **s** command attempts to match the pattern space against the supplied *regexp*; if the match is successful, then that portion of the pattern space which was matched is replaced with *replacement*.

The replacement can contain \n (*n* being a number from **1** to **9**, inclusive) references, which refer to the portion of the match which is contained between the *n*th **\\(** and its matching **\\)**. Also, the replacement can contain unescaped **&** characters which reference the whole matched portion of the pattern space. Finally, as a GNU **sed** extension, you can include a special sequence made of a backslash and one of the letters **L**, **l**, **U**, **u**, or **E**. The meaning is as follows:

\\L	Turn the replacement to lowercase until a **\\U** or **\\E** is found
\\l	Turn the next character to lowercase
\\U	Turn the replacement to uppercase until a **\\L** or **\\E** is found
\\u	Turn the next character to uppercase
\\E	Stop case conversion started by **\\L** or **\\U**

To include a literal \\, **&**, or newline in the final replacement, be sure to precede the desired \\, **&**, or newline in the replacement with a \\.

The **s** command can be followed by zero or more of the following flags:

g	Apply the replacement to all matches to the regexp, not just the first.
number	Only replace the *number*th match of the regexp. Note: the POSIX standard does not specify what should happen when you mix the **g** and *number* modifiers, and currently there is no widely agreed upon meaning across **sed** implementations. For GNU **sed**, the interaction is defined

	to be: ignore matches before the *number*th, and then match and replace all matches from the *number*th on.
p	If the substitution was made, then print the new pattern space. Note: when both the **p** and **e** options are specified, the relative ordering of the two produces very different results. In general, **ep** (evaluate then print) is what you want, but operating the other way round can be useful for debugging. For this reason, the current version of GNU **sed** interprets specially the presence of **p** options both before and after **e**, printing the pattern space before and after evaluation, while in general flags for the **s** command show their effect just once. This behavior, although documented, might change in future versions.
w *file*	If the substitution was made, then write out the result to the named *file*. As a GNU **sed** extension, two special values of *file* are supported: **/dev/stderr**, which writes the result to the standard error, and **/dev/stdout**, which writes to the standard output.
e	This command allows one to pipe input from a shell command into pattern space. If a substitution was made, the command that is found in pattern space is executed and pattern space is replaced with its output. A trailing newline is suppressed; results are undefined if the command to be executed contains a null character. This is a GNU **sed** extension.
I, **i**	The **I** modifier to regular-expression matching is a GNU extension which makes **sed** match regexp in a case-insensitive manner.
M, **m**	The **M** modifier to regular-expression matching is a GNU **sed** extension which causes ^ and **$** to match respectively (in addition to the normal behavior) the empty string after a newline, and the empty string before a newline. There are special character sequences (\\` and \\') which always match the beginning or the end of the buffer. **M** stands for multi-line.

Less Frequently-Used Commands

Though perhaps less frequently used than those in the previous section, some very small yet useful **sed** scripts can be built with these commands.

y/*source-chars*/*dest-chars*/

(The / characters may be uniformly replaced by any other single character within any given **y** command.)

Transliterate any characters in the pattern space which match any of the *source-chars* with the corresponding character in *dest-chars*.

Instances of the / (or whatever other character is used instead), \, or newlines can appear in the *source-chars* or *dest-chars* lists, provide that each instance is escaped by a \. The *source-chars* and *dest-chars* lists must contain the same number of characters (after de-escaping).

a\ *text*

As a GNU extension, this command accepts two addresses.

Queue the lines of text which follow this command (each but the last ending with a \, which are removed from the output) to be output at the end of the current cycle, or when the next input line is read.

Escape sequences in text are processed, so you should use \\ in text to print a single backslash.

As a GNU extension, if between the **a** and the newline there is other than a whitespace-\ sequence, then the text of this line, starting at the first non-whitespace character after the **a**, is taken as the first line of the text block. (This enables a simplification in scripting a one-line add.) This extension also works with the **i** and **c** commands.

i\ *text*

As a GNU extension, this command accepts two addresses.

	Immediately output the lines of text which follow this command (each but the last ending with a \\, which are removed from the output).
c\\ *text*	Delete the lines matching the address or address-range, and output the lines of *text* which follow this command (each but the last ending with a \\, which are removed from the output) in place of the last line (or in place of each line, if no addresses were specified). A new cycle is started after this command is done, since the pattern space will have been deleted.
=	As a GNU extension, this command accepts two addresses. Print out the current input line number (with a trailing newline).
l *n*	Print the pattern space in an unambiguous form: non-printable characters (and the \\ character) are printed in C-style escaped form; long lines are split, with a trailing \\ character to indicate the split; the end of each line is marked with a **$**. *n* specifies the desired line-wrap length; a length of **0** (zero) means to never wrap long lines. If omitted, the default as specified on the command line is used. The *n* parameter is a GNU **sed** extension.
r *filename*	As a GNU extension, this command accepts two addresses. Queue the contents of *filename* to be read and inserted into the output stream at the end of the current cycle, or when the next input line is read. Note that if *filename* cannot be read, it is treated as if it were an empty file, without any error indication. As a GNU **sed** extension, the special value **/dev/stdin** is supported for the file name, which reads the contents of the standard input.

w *filename*	Write the pattern space to *filename*. As a GNU **sed** extension, two special values of *filename* are supported: **/dev/stderr**, which writes the result to the standard error, and **/dev/stdout**, which writes to the standard output. The file will be created (or truncated) before the first input line is read; all **w** commands (including instances of the **w** flag on successful s commands) which refer to the same filename are output without closing and reopening the file.
D	If pattern space contains no newline, start a normal new cycle as if the **d** command was issued. Otherwise, delete text in the pattern space up to the first newline, and restart cycle with the resultant pattern space, without reading a new line of input.
N	Add a newline to the pattern space, then append the next line of input to the pattern space. If there is no more input then **sed** exits without processing any more commands.
P	Print out the portion of the pattern space up to the first newline.
h	Replace the contents of the hold space with the contents of the pattern space.
H	Append a newline to the contents of the hold space, and then append the contents of the pattern space to that of the hold space.
g	Replace the contents of the pattern space with the contents of the hold space.
G	Append a newline to the contents of the pattern space, and then append the contents of the hold space to that of the pattern space.

x	Exchange the contents of the hold and pattern spaces.

Commands for sed gurus

In most cases, use of these commands indicates that you are probably better off programming in something like awk or Perl. But occasionally one is committed to sticking with **sed**, and these commands can enable one to write quite convoluted scripts.

: *label*	[No addresses allowed with this command.] Specify the location of *label* for branch commands. In all other respects, a no-op (no operation performed).
b *label*	Unconditionally branch to *label*. The label may be omitted, in which case the next cycle is started.
t *label*	Branch to *label* only if there has been a successful substitution since the last input line was read or conditional branch was taken. The *label* may be omitted, in which case the next cycle is started.

Commands Specific to GNU sed

These commands are specific to GNU **sed**, so you must use them with care and only when you are sure that the script will not need to be ported. They allow you to check for GNU **sed** extensions or to do tasks that are required quite often, yet are unsupported by standard **sed**s.

e [*command*]	This command allows one to pipe input from a shell command into pattern space. Without parameters, the **e** command executes the command that is found in pattern space and replaces the pattern space with the output; a trailing newline is suppressed. If a parameter is specified, instead, the **e** command interprets it as a

command and sends its output to the output stream (like **r** does). The *command* can run across multiple lines, all but the last ending with a back-slash.

In both cases, the results are undefined if the command to be executed contains a null character.

F	Print out the file name of the current input file (with a trailing newline).
L *n*	This GNU **sed** extension fills and joins lines in pattern space to produce output lines of (at most) *n* characters, like fmt does; if *n* is omitted, the default as specified on the command line is used. This command is considered a failed experiment and unless there is enough request (which seems unlikely) will be removed in future versions.
Q [*exit-code*]	This command only accepts a single address. This command is the same as **q**, but will not print the contents of pattern space. Like **q**, it provides the ability to return an exit code to the caller. This command can be useful because the only alternative ways to accomplish this apparently trivial function are to use the **-n** option (which can unnecessarily complicate your script) or resorting to the following snippet, which wastes time by reading the whole file without any visible effect: :eat #Quit silently on the last line: $d #Read another line, silently: N #Overwrite pattern space each time to save memory: g b eat.
R *filename*	Queue a line of *filename* to be read and inserted into the output stream at the end of the current cycle, or when the next input line is read. Note that if *filename* cannot be read, or if its end is reached, no line is appended, without any error indication.

As with the **r** command, the special value **/dev/stdin** is supported for the file name, which reads a line from the standard input.

T *label* — Branch to *label* only if there have been no successful substitutions since the last input line was read or conditional branch was taken. The *label* may be omitted, in which case the next cycle is started.

v *version* — This command does nothing, but makes **sed** fail if GNU **sed** extensions are not supported, simply because other versions of **sed** do not implement it. In addition, you can specify the version of **sed** that your script requires, such as **4.0.5**. The default is **4.0** because that is the first version that implemented this command.

This command enables all GNU extensions even if **POSIXLY_CORRECT** is set in the environment.

W *filename* — Write to the given filename the portion of the pattern space up to the first newline. Everything said under the **w** command about file handling holds here too.

z — This command empties the content of pattern space. It is usually the same as '**s/.*//**', but is more efficient and works in the presence of invalid multibyte sequences in the input stream. POSIX mandates that such sequences are not matched by '**.**', so that there is no portable way to clear **sed**'s buffers in the middle of the script in most multibyte locales (including UTF-8 locales).

GNU Extensions for Escapes in Regular Expressions

Until now (in this document, anyway), we have only encountered escapes of the form '\^', for example, which tell **sed** not to interpret the circumflex (caret) as a special character, but rather to take it literally. For another example, '*' matches a single asterisk rather than zero or more backslashes.

This section introduces another kind of escape—that is, escapes that are applied to a character or sequence of characters that ordinarily are taken literally, and that **sed** replaces with a special character. This provides a way of encoding non-printable characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters in a **sed** script but when a script is being prepared in the shell or by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a	Produces or matches a bel character, that is an "alert" (ASCII 7).
\f	Produces or matches a form feed (ASCII 12).
\n	Produces or matches a newline (ASCII 10).
\r	Produces or matches a carriage return (ASCII 13).
\t	Produces or matches a horizontal tab (ASCII 9).
\v	Produces or matches a so called "vertical tab" (ASCII 11).
\c*x*	Produces or matches **Control-***x*, where *x* is any character. The precise effect of '**\c***x*' is as follows: if *x* is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus '**\cz**' becomes hex 1A, but '**\c{**' becomes hex 3B, while '**\c;**' becomes hex 7B.
\d*xxx*	Produces or matches a character whose decimal ASCII value is *xxx*.
\o*xxx*	Produces or matches a character whose octal ASCII value is *xxx*.

\x*xx*	Produces or matches a character whose hexadecimal ASCII value is *xx*.

‘**\b**’ (backspace) was omitted because of the conflict with the existing "word boundary" meaning.

Other escapes match a particular character class and are valid only in regular expressions:

\w	Matches any "word" character. A "word" character is any letter or digit or the underscore character.
\W	Matches any "non-word" character.
\b	Matches a word boundary; that is, it matches if the character to the left is a "word" character and the character to the right is a "non-word" character, or vice-versa.
\B	Matches everywhere but on a word boundary; that is it matches if the character to the left and the character to the right are either both "word" characters or both "non-word" characters.
\\`	Matches only at the start of pattern space. This is different from ^ in multi-line mode.
\\'	Matches only at the end of pattern space. This is different from **$** in multi-line mode.

Some Sample Scripts

Here are some **sed** scripts to guide you in the art of mastering **sed**.

Sample Script: Centering Lines

This script centers all lines of a file on 80 columns width. To change that width, the number in \{...\} must be replaced, and the number of added spaces also must be changed.

Note how the buffer commands are used to separate parts in the regular expressions to be matched—this is a common technique.

```
#!/usr/bin/sed -f

# Put 80 spaces in the buffer
1 {
  x
  s/^$/          /
  s/^.*$/&&&&&&&&/
  x
}

# del leading and trailing spaces
y/tab/ /
s/^ *//
s/ *$//

# add a newline and 80 spaces to end of line
G

# keep first 81 chars (80 + a newline)
s/^\(.\{81\}\).*$/\1/

# \2 matches half of the spaces, which are moved to the beg
s/^\(.*\)\n\(.*\)\2/\2\1/
```

Sample Script: Increment A Number

This script is one of a few that demonstrate how to do arithmetic in **sed**. This is indeed possible, but must be done manually.

To increment one number you just add 1 to last digit, replacing it by the following digit. There is one exception: when the digit is a nine the previous digits must be also incremented until you don't have a nine.

This solution is very clever and smart because it uses a single buffer; if you don't have this limitation, the algorithm used in Numbering Lines is faster. It works by replacing trailing nines with an underscore, then using multiple **s** commands to increment the last digit, and then again substituting underscores with zeros.

```
#!/usr/bin/sed -f

/[^0-9]/ d

# replace all leading 9s by _ (any other character except d
# be used)
:d
s/9\(_*\)$/_\1/
td

# incr last digit only.  The first line adds a most-signifi
# digit of 1 if we have to add a digit.
#
# The tn commands are not necessary, but make the thing
# faster

s/^\(_*\)$/1\1/; tn
s/8\(_*\)$/9\1/; tn
s/7\(_*\)$/8\1/; tn
s/6\(_*\)$/7\1/; tn
s/5\(_*\)$/6\1/; tn
```

```
s/4\(_*\)$/5\1/; tn
s/3\(_*\)$/4\1/; tn
s/2\(_*\)$/3\1/; tn
s/1\(_*\)$/2\1/; tn
s/0\(_*\)$/1\1/; tn

:n
y/_/0/
```

Sample Script: Rename Files To Lower Case

This is a pretty strange use of **sed**. We transform text, and transform it to be shell commands, then just feed them to shell. Don't worry, even worse hacks are done when using **sed**. Scripts have even been written converting the output of date into a bc program... So, stranger things have happened.

The main body of this is the **sed** script, which remaps the name from lower to upper (or vice-versa) and even checks out if the remapped name is the same as the original name. Note how the script is parameterized using shell variables and proper quoting.

```
#! /bin/sh
# rename files to lower/upper case...
#
# usage:
#    move-to-lower *
#    move-to-upper *
# or
#    move-to-lower -R .
#    move-to-upper -R .
#

help()
```

```
{
        cat << eof
Usage: $0 [-n] [-r] [-h] files...

-n      do nothing, only see what would be done
-R      recursive (use find)
-h      this message
files   files to remap to lower case

Examples:
       $0 -n *        (see if everything is ok, then...)
       $0 *

       $0 -R .

eof
}

apply_cmd='sh'
finder='echo "$@" | tr " " "\n"'
files_only=

while :
do
    case "$1" in
        -n) apply_cmd='cat' ;;
        -R) finder='find "$@" -type f';;
        -h) help ; exit 1 ;;
        *) break ;;
    esac
    shift
done
```

```
if [ -z "$1" ]; then
        echo Usage: $0 [-h] [-n] [-r] files...
        exit 1
fi

LOWER='abcdefghijklmnopqrstuvwxyz'
UPPER='ABCDEFGHIJKLMNOPQRSTUVWXYZ'

case `basename $0` in
        *upper*) TO=$UPPER; FROM=$LOWER ;;
        *)       FROM=$UPPER; TO=$LOWER ;;
esac

eval $finder | sed -n '

# remove all trailing slashes
s/\/*$//

# add ./ if there is no path, only a filename
/\//! s/^/.\//

# save path+filename
h

# remove path
s/.*\///

# do conversion only on filename
y/'$FROM'/'$TO'/

# now line contains original path+file, while
# hold space contains the new filename
x

```

```
# add converted file name to line, which now contains
# path/file-name\nconverted-file-name
G

# check if converted file name is equal to original file na
# if it is, do not print nothing
/^.*\/\(.*\)\n\1/b

# now, transform path/fromfile\n, into
# mv path/fromfile path/tofile and print it
s/^\(.*\/\)\(.*\)\n\(.*\)$/mv "\1\2" "\1\3"/p

' | $apply_cmd
```

Sample Script: Print bash Environment

This script strips the definition of the shell functions from the output of the set command in the Bourne-Again shell (bash).

```
#!/bin/bash

set | sed -n '
:x

# if no occurrence of '=()' print and load next line
/=()/! { p; b; }
/ () $/! { p; b; }

# possible start of functions section
# save the line in case this is a var like FOO="() "
h
```

```
# if the next line has a brace, we quit because
# nothing comes after functions
n
/^{/ q

# print the old line
x; p

# work on the new line now
x; bx
'
```

Sample Script: Reverse Characters Of Lines

This script can be used to reverse the position of characters in lines. The technique moves two characters at a time, hence it is faster than more intuitive implementations.

Note the **tx** command before the definition of the label. This is often needed to reset the flag that is tested by the **t** command.

```
#!/usr/bin/sed -f

/../! b

# Reverse a line.  Begin embedding the line between two new
s/^.*$/\
&\
/

# Move first character at the end.  The regexp matches unti
# there are zero or one characters between the markers
```

```
tx
:x
s/\(\n.\)\(.*\)\(.\n\)/\3\2\1/
tx

# Remove the newline markers
s/\n//g
```

Sample Script: Reverse Lines Of Files

This one begins a series of totally useless (yet interesting) scripts emulating various Unix commands. This, in particular, is a tac workalike.

Note that on implementations other than GNU **sed** this script might easily overflow internal buffers.

```
#!/usr/bin/sed -nf

# reverse all lines of input, i.e. first line became last,

# from the second line, the buffer (which contains all prev
# is *appended* to current line, so, the order will be reve
1! G

# on the last line we're done -- print everything
$ p

# store everything on the buffer again
h
```

Sample Script: Numbering Lines

This script replaces '**cat -n**'; in fact it formats its output exactly like GNU cat does.

Of course this is completely useless for two reasons: first, because somebody else did it in C (the **cat** command), and second, because the following Bourne-shell script could be used for the same purpose and would be much faster:

```
#! /bin/sh
sed -e "=" $@ | sed -e '
  s/^/      /
  N
  s/^ *\(......\)\n/\1  /
'
```

It uses **sed** to print the line number, then groups lines two by two using **N**. Of course, this script does not teach as much as the one presented below.

The algorithm used for incrementing uses both buffers, so the line is printed as soon as possible and then discarded. The number is split so that changing digits go in a buffer and unchanged ones go in the other; the changed digits are modified in a single step (using a **y** command). The line number for the next line is then composed and stored in the hold space, to be used in the next iteration.

```
#!/usr/bin/sed -nf

# Prime the pump on the first line
x
/^$/ s/^.*$/1/

# Add the correct line number before the pattern
G
```

```
h

# Format it and print it
s/^/      /
s/^ *\(......\)\n/\1  /p

# Get the line number from hold space; add a zero
# if we're going to add a digit on the next line
g
s/\n.*$//
/^9*$/ s/^/0/

# separate changing/unchanged digits with an x
s/.9*$/x&/

# keep changing digits in hold space
h
s/^.*x//
y/0123456789/1234567890/
x

# keep unchanged digits in pattern space
s/x.*$//

# compose the new number, remove the newline implicitly add
G
s/\n//
h
```

Sample Script: Numbering Non-Blank Lines

Emulating ‘**cat -b**’ is almost the same as ‘**cat -n**’: we only have to select which lines are to be numbered and which are not.

The part that is common to this script and the previous one is not commented to show how important it is to comment **sed** scripts properly...

```
#!/usr/bin/sed -nf

/^$/ {
  p
  b
}

# Same as cat -n from now
x
/^$/ s/^.*$/1/
G
h
s/^/      /
s/^ *\(......\)\n/\1  /p
x
s/\n.*$//
/^9*$/ s/^/0/
s/.9*$/x&/
h
s/^.*x//
y/0123456789/1234567890/
x
s/x.*$//
G
s/\n//
h
```

Sample Script: Counting Characters

This script shows another way to do arithmetic with **sed**. In this case we have to add possibly large numbers, so implementing this by successive increments would not be feasible (and possibly even more complicated to contrive than this script).

The approach is to map numbers to letters, kind of an abacus implemented with **sed**. ‘**a**’s are units, ‘**b**’s are tens and so on: we simply add the number of characters on the current line as units, and then propagate the carry to tens, hundreds, and so on.

As usual, running totals are kept in hold space.

On the last line, we convert the abacus form back to decimal. For the sake of variety, this is done with a loop rather than with some **80 s** commands: first we convert units, removing ‘**a**’s from the number; then we rotate letters so that tens become ‘**a**’s, and so on until no more letters remain.

```
#!/usr/bin/sed -nf

# Add n+1 a's to hold space (+1 is for the newline)
s/./a/g
H
x
s/\n/a/

# Do the carry.  The t's and b's are not necessary,
# but they do speed up the thing
t a
: a;  s/aaaaaaaaaa/b/g; t b; b done
: b;  s/bbbbbbbbbb/c/g; t c; b done
: c;  s/cccccccccc/d/g; t d; b done
: d;  s/dddddddddd/e/g; t e; b done
: e;  s/eeeeeeeeee/f/g; t f; b done
: f;  s/ffffffffff/g/g; t g; b done
: g;  s/gggggggggg/h/g; t h; b done
: h;  s/hhhhhhhhhh//g

: done
$! {
  h
  b
}

# On the last line, convert back to decimal

: loop
/a/! s/[b-h]*/&0/
s/aaaaaaaaa/9/
s/aaaaaaaa/8/
s/aaaaaaa/7/
s/aaaaaa/6/
s/aaaaa/5/
s/aaaa/4/
s/aaa/3/
s/aa/2/
s/a/1/

: next
y/bcdefgh/abcdefg/
/[a-h]/ b loop
p
```

Sample Script: Counting Words

This script is almost the same as the previous one, once each of the words on the line is converted to a single '**a**' (in the previous script each letter was changed to an '**a**').

It is interesting that real wc programs have optimized loops for '**wc -c**', so they are much slower at counting words rather than characters. This script's bottleneck, instead, is arithmetic, and hence the word-counting one is faster (it has to manage smaller numbers).

Again, the common parts are not commented to show the importance of commenting **sed** scripts.

```
#!/usr/bin/sed -nf

# Convert words to a's
s/[ tab][ tab]*/ /g
s/^/ /
s/ [^ ][^ ]*/a /g
s/ //g

# Append them to hold space
H
x
s/\n//

# From here on it is the same as in wc -c.
/aaaaaaaaaa/! bx;   s/aaaaaaaaaa/b/g
/bbbbbbbbbb/! bx;   s/bbbbbbbbbb/c/g
/cccccccccc/! bx;   s/cccccccccc/d/g
/dddddddddd/! bx;   s/dddddddddd/e/g
/eeeeeeeeee/! bx;   s/eeeeeeeeee/f/g
/ffffffffff/! bx;   s/ffffffffff/g/g
/gggggggggg/! bx;   s/gggggggggg/h/g
s/hhhhhhhhhh//g
:x
```

```
$! { h; b; }
:y
/a/! s/[b-h]*/&0/
s/aaaaaaaaa/9/
s/aaaaaaaa/8/
s/aaaaaaa/7/
s/aaaaaa/6/
s/aaaaa/5/
s/aaaa/4/
s/aaa/3/
s/aa/2/
s/a/1/
y/bcdefgh/abcdefg/
/[a-h]/ by
p
```

Sample Script: Counting Lines

No strange things are done now, because sed gives us '**wc -l**' functionality for free!!! (Restrictions apply. Limited to one per household. See official rules for details.) Here is the code:

```
#!/usr/bin/sed -nf
$=
```

Sample Script: Printing The First Lines

This script is probably the simplest useful **sed** script. It displays the first 10 lines of input; the number of displayed lines is right before the **q** command.

```
#!/usr/bin/sed -f
```

```
10q
```

Sample Script: Printing The Last Lines

Printing the last *n* lines rather than the first is more complex but indeed possible. *n* is encoded in the second line, before the bang ("**!**") character.

This script is similar to the **tac** script (above) in that it keeps the final output in the hold space and prints it at the end:

```
#!/usr/bin/sed -nf

1! {; H; g; }
1,10 !s/[^\n]*\n//
$p
h
```

Mainly, the scripts keeps a window of 10 lines and slides it by adding a line and deleting the oldest (the substitution command on the second line works like a **D** command but does not restart the loop).

The "sliding window" technique is a very powerful way to write efficient and complex **sed** scripts, because commands like **P** would require a lot of work if implemented manually.

To introduce the technique, which is fully demonstrated in the rest of this chapter and is based on the **N**, **P** and **D** commands, here is an implementation of tail using a simple "sliding window."

This looks complicated but in fact the working concept is the same as the last script: after we have kicked in the appropriate number of lines, however, we stop using the hold space to keep inter-line state, and instead use **N** and **D** to slide pattern space by one line:

```
#!/usr/bin/sed -f

1h
2,10 {; H; g; }
$q
1,9d
N
D
```

Note how the first, second and fourth line are inactive after the first ten lines of input. After that, all the script does is: exiting on the last line of input, appending the next input line to pattern space, and removing the first line.

Sample Script: Make Duplicate Lines Unique

This is an example of the art of using the **N**, **P** and **D** commands, probably the most difficult to master.

```
#!/usr/bin/sed -f
h

:b
# On the last line, print and exit
$b
N
/^\(.*\)\n\1$/ {
    # The two lines are identical.  Undo the effect of
    # the n command.
    g
    bb
}

# If the N command had added the last line, print and exit
$b

# The lines are different; print the first and go
# back working on the second.
P
D
```

As you can see, we mantain a 2-line window using **P** and **D**. This technique is often used in advanced **sed** scripts.

Sample Script: Print Duplicated Lines Of Input

This script prints only duplicated lines, like ‘**uniq -d**’.

```
#!/usr/bin/sed -nf

$b
N
/^\(.*\)\n\1$/ {
    # Print the first of the duplicated lines
    s/.*\n//
    p

    # Loop until we get a different line
    :b
    $b
    N
    /^\(.*\)\n\1$/ {
        s/.*\n//
        bb
    }
}

# The last line cannot be followed by duplicates
$b

# Found a different one.  Leave it alone in the pattern spa
# and go back to the top, hunting its duplicates
D
```

Sample Script: Remove All Duplicated Lines

This script prints only unique lines, like '**uniq -u**'.

```
#!/usr/bin/sed -f

# Search for a duplicate line --- until that, print what yo
$b
N
/^\(.*\)\n\1$/ ! {
    P
    D
}

:c
# Got two equal lines in pattern space.  At the
# end of the file we simply exit
$d
```

```
# Else, we keep reading lines with N until we
# find a different one
s/.*\n//
N
/^\(.*\)\n\1$/ {
    bc
}

# Remove the last instance of the duplicate line
# and go back to the top
D
```

Sample Script: Squeezing Blank Lines

As a final example, here are three scripts, of increasing complexity and speed, that implement the same function as '**cat -s**', that is squeezing blank lines.

The first leaves a blank line at the beginning and end if there are some already.

```
#!/usr/bin/sed -f

# on empty lines, join with next
# Note there is a star in the regexp
:x
/^\n*$/ {
N
bx
}

# now, squeeze all '\n', this can be also done by:
# s/^\(\n\)*/\1/
s/\n*/\
/
```

This one is a bit more complex and removes all empty lines at the beginning. It does leave a single blank line at end if one was there.

```
#!/usr/bin/sed -f

# delete all leading empty lines
1,/^./{
/./!d
}

# on an empty line we remove it and all the following
# empty lines, but one
:x
/./!{
N
s/^\n$//
tx
}
```

This removes leading and trailing blank lines. It is also the fastest. Note that loops are completely done with **n** and **b**, without relying on sed to restart the the script automatically at the end of a line.

```
#!/usr/bin/sed -nf

# delete all (leading) blanks
/./!d
```

```
# get here: so there is a non empty
:x
# print it
p
# get next
n
# got chars? print it again, etc...
/./bx

# no, don't have chars: got an empty line
:z
# get next, if last line we finish here so no trailing
# empty lines are written
n
# also empty? then ignore it, and get next... this will
# remove ALL empty lines
/./!bz

# all empty lines were deleted/ignored, but we have a non e
# what we want to do is to squeeze, insert a blank line art
i\

bx
```

GNU sed's Limitations (And Non-Limitations)

For those who want to write portable **sed** scripts, be aware that some implementations have been known to limit line lengths (for the pattern and hold spaces) to be no more than 4000 bytes. The POSIX standard specifies that conforming **sed** implementations shall support at least 8192 byte line lengths. GNU **sed** has no built-in limit on line length; as long as it can allocate more (virtual) memory, you can feed or construct lines as long as you like.

However, recursion is used to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of the buffer that can be processed by certain patterns.

About Extended Regular Expressions

The only difference between basic and extended regular expressions is in the behavior of a few characters: '**?**', '+', parentheses, and braces ('**{}**'). While basic regular expressions require these to be escaped if you want them to behave as special characters, when using extended regular expressions you must escape them if you want them to match a literal character.

For example:

abc?	becomes '**abc\?**' when using extended regular expressions. It matches the literal string '**abc?**'.
c\+	becomes '*c*+' when using extended regular expressions. It matches one or more '*c*'s.
a\{3,\}	becomes '**a{3,}**' when using extended regular expressions. It matches three or more '**a**'s.
\(abc\) \{2,3\}	becomes '**(abc){2,3}**' when using extended regular expressions. It matches either '**abcabc**' or '**abcabcabc**'.
\(abc*\)\1	becomes '**(abc*)\1**' when using extended regular expressions. Backreferences must still be escaped when using extended regular expressions.

sed examples

```
sed G myfile.txt > newfile.txt
```

Double-spaces the contents of file **myfile.txt**, and writes the output to the file **newfile.txt**.

```
sed = myfile.txt | sed 'N;s/\n/\. /'
```

Prefixes each line of **myfile.txt** with a line number, a period, and a space, and displays the output.

```
sed 's/test/example/g' myfile.txt > newfile.txt
```

Searches for the word "**test**" in **myfile.txt** and replaces every occurrence with the word "**example**".

```
sed -n '$=' myfile.txt
```

Counts the number of lines in **myfile.txt** and displays the results.

Related commands

awk — Interpreter for the AWK text processing programming language.
ed — A simple text editor.
grep — Filter text which matches a regular expression.
replace — A string-replacement utility.