Mastering Linux by Paul S. Wang

Appendix: Text Editing with `vi`

A text editor is used to enter programs, data, or text into new files, as well as to modify existing files. There are as many different text editors as there are computer systems. The **vi** editor is a full-terminal–window extension of **ex**, which is a line-oriented editor. The *vim* (vi improved) is standard on most Linux systems (see another appendix for an introduction to *vim*. Information here about how to use **vi** applies also to **vim**.
Information in the first sections will enable you to enter and edit programs comfortably. The remaining sections will make you an expert and you'll get any editing job done much faster.

Display Text-Editing Concepts

When you edit a file, the editor creates a text *buffer* for the temporary storage of the file you are editing. Your terminal or window displays the contents of the buffer as you edit. The text displayed is in your *current screen* or simply screen. The basic unit of a text file is the character. Consecutive characters form *words*, *lines*, *paragraphs*, and so on. A file can contain any number of lines. The screen can be moved forward and backward to view any part of the file depending on the commands you give the editor through the keyboard. Since both text and editor commands are entered on the keyboard, every text editor must have some convention for distinguishing between keystrokes to be executed as commands and keystrokes to be entered into the buffer as text. As you will see in this chapter, the **vi** convention involves defining different editor modes; **emacs** takes a different approach and uses CONTROL and ESCAPE characters to distinguish between commands and text.

Both editors display a *cursor* on the screen to indicate the position where editing will take place in the buffer. This is called the *current position*. If you give the command to delete a character, then the character at the current position, known as the *current character*, is deleted from the buffer. All insertions, deletions, and other modifications are performed within the buffer. No changes are made to the actual file unless and until you specifically request that the buffer be *saved* to the disk where the file is stored.

Screen editors provide commands to

- Move the window and move the cursor within the text file
- Insert text
- Delete text

- Substitute specified text with other text
- Search for a given string or pattern
- Cut and paste
- Perform file input and output

Getting Started with `vi`

To edit a file with **vi**, issue the Shell-level command

vi *file*

If the file exists, a copy of it will be read into the buffer, and the screen will display the beginning of the file. If the file does not exist, **vi** will create an empty buffer for you to insert and edit new text into. For a new file, the screen will display all empty lines, each of which is indicated by a tilde (`~`) character in the first column. The **vi** editor uses the tildes to indicate "lines beyond the end of the buffer." The tildes are merely a convention; the characters are not contained in the buffer.

Initially, **vi** is in the *command mode*. In this mode, **vi** considers all keystrokes as editing commands. To put text into the buffer, issue a command (for example, **i** or **o**) to change to the *insert mode* (see Section 2.4), then begin typing the text to be inserted. Text insertion is terminated by hitting ESC, which returns **vi** to the command mode. In other words, when you are not inserting text into the buffer, **vi** is always in command mode.

In the command mode, **vi** stands ready to receive and carry out your next command. The **vi** commands are short and easy to type: They are single-, double-, and multiple-character commands, which are carried out as soon as they are typed. Some **vi** commands are terminated by the special characters ESC or RETURN, and most **vi** commands can be preceded by an integer value, called a *repeat count*. The repeat count specifies the number of times a command is to be performed. We use the variable *n* to indicate this repeat count when presenting these commands. If *n* is omitted it is assumed to be 1. Typing errors in a **vi** command can be corrected using input editing (see Section 1.2).

The **vi** buffer window is always at least one line less than the total number of lines on your screen. The bottom line is called the *status line* and is used by **vi** to echo certain commands and to display messages. Any command beginning with a colon (`:`) as well as any search command beginning with a slash (`/`) (or `?`) is echoed on the status line. However, most other commands are not. Not being able to see what you type in the command mode can take some getting used to.

When you are finished with the editing session, you need to save the buffer and get out of **vi**. You type

ZZ (save buffer and exit)

If you wish to discard the buffer without updating the disk file, type

:q!RETURN (quit without saving buffer on disk)

instead of **ZZ**. The command **q** without the trailing **!** will quit only if the buffer has not been changed since the last save to disk. Be careful: It's easy to miss **q** and hit **w** instead. Unfortunately, the **vi** command

:w!RETURN (write buffer over file)

writes the file out even if there is a file by the same name on the disk; so instead of abandoning the unwanted buffer, **vi** wipes out a file!!!

Moving the Screen

In **vi** and other display editors, the *screen* makes a portion of the editor buffer visible. To edit a file, first you must move the screen to that part of the file where editing will take place. There are a few ways to do this: by line number, by search, or by *browsing*.

The buffer consists of lines of text numbered sequentially beginning with line 1. The command

*k***G** (goto line *k*)

will position the screen over the buffer so that the *k*th line is centered on the screen with the cursor placed at the beginning of the line. **G** provides a way to move the screen to any absolute position within the buffer; it also is possible to move the screen in a relative manner. To move forward and backward *n* lines from the *current line* (line with cursor), *n*RETURN and *n*– are used, respectively. The cursor is positioned over the first nonwhite character (a space, a tab, and so on) of the target line. Sometimes, pressing RETURN or **–** (minus sign) repeatedly is a convenient way to move a few lines forward or backward.

In many cases the line number of the target is not at your fingertips, but if you know an unambiguous substring of the text, a search command can be used to locate the target. The **vi** commands

/*text*RETURN (forward search)
?*text*RETURN (backward search)

search the buffer for the given *text* (these commands will be echoed on the status line). The forward search begins at the *current character* (at the cursor) and proceeds forward toward the end of buffer. If the pattern is found, the cursor will be positioned at the patterns first character. If the search reaches the end of buffer, it will *wrap around* to the beginning of the buffer until the

pattern is found or until the search returns to where it started. A backward search moves in the opposite direction, starting toward the beginning of the buffer and then wrapping around through the end until it returns to its starting point. If the pattern is not found, a message "Fail" will appear on the status line. You can locate multiple occurrences of a pattern after the first occurrence is found by typing

n (search for the next match)

which searches in the direction of the previous search command, or by typing

N (search for the next match)

which searches in the reverse direction of the previous search command.

The third way to position the screen is by browsing around until you find the section of the buffer you want. The commands

^D (scroll down)
^U (scroll up)

scroll the screen forward and backward, respectively. Smooth scroll is used if your terminal has this feature, so that you can read the lines as they are scrolling by. The commands

^F (forward)
^B (backward)

refresh the screen and give the next and previous screen, respectively. Two lines of text overlap the last screen to provide reading continuity.

Cursor Motion Commands

Once the screen has been positioned over the buffer location you wish to edit, the cursor must be positioned to the exact line, word, or character you wish to change. The **vi** editor has a full complement of commands to move the cursor. The following commands move the cursor to the first nonwhite character at the beginning of the target line.

H first line on the current screen
L last line on the screen
M middle line on the screen

In addition

h moves the cursor left one character on the current line
j moves the cursor down one line onto the same column position or the last character if the next line is not long enough
k moves the cursor up one line onto the same column position or the last character if the previous line is not long enough
l moves the cursor right one character on the current line
SPACE works the same as **l**

^	moves the cursor to the beginning of the current line
$	moves the cursor to the end of the current line
w	moves the cursor forward one word to the first character of the next word
e	moves the cursor to the end of the current word
b	moves the cursor backward one word to the first character that begins a word
t*c*	moves the cursor to the character just before the next occurrence of the character *c* in the current line (**T** for just after the previous occurrence of *c*)
f *c*	moves the cursor to the next (**F** for previous) occurrence of the character *c* in the current line
;	(semicolon) repeats the previous **f** or **F** command
,	(comma) repeats the previous **f** or **F** command in the other direction

The **f**, **;**, **F**, and **,** combinations are fast ways of moving the cursor in the current line. If your terminal has arrow keys, they will work the same as **h**, **j**, **k**, and **l**, although many prefer the letter keys because they are convenient for touch typing. Many people find it even easier to use SPACEBAR for **l**. To move the cursor to a specific column in the current line use

k| (cursor to column *k*)

The command | without a preceding number is equivalent to ^.

The **vi** editing commands also can move the cursor to sentences, paragraphs, and sections. To move the cursor backward or forward over a sentence, paragraph or section, use

(	moves the cursor left to the beginning of a sentence
)	moves the cursor right to the beginning of a sentence
{	moves the cursor left to the beginning of a paragraph
}	moves the cursor right to the beginning of a paragraph
[[	moves the cursor left to the beginning of a section
]]	moves the cursor right to the beginning of a section

These commands can be combined with commands such as delete to specify editing actions on words, sentences, paragraphs, and sections. For example,

d)

deletes everything from the current position up to the beginning of the next sentence.

Typing or Inserting Text

The **vi** editor uses mode to distinguish between keystrokes meant as commands and keystrokes meant as text. While the editor is in the insert mode, all keystrokes are considered text and are inserted into the buffer until you press ESC. Several commands put **vi** into the insert mode. The most frequently used command is

i*any number of*
lines of text to be inserted
terminated by the first escape character ESC

This command inserts the text just before the cursor (if you use **a** instead of **i**, the text will be inserted to the right of the cursor). After you press **i**, **vi** goes into the insert mode. From that point on, characters typed go into the buffer and are not considered to be commands. If, for example, you were to type **i** and then **ZZ**, the editor would insert `ZZ` into the buffer. If you type **ZZ** in the command mode, you would leave the editor.

Text can be inserted more than one line at a time, and lines can be separated by pressing RETURN. As you type, **vi** displays either what you type over the existing text on the screen or pushes the existing characters to the right as you type in new text. The exact method depends on your terminal. If the inserted text overwrites existing characters on the screen, do not be alarmed. These characters are still in the buffer. When you end your insertion by pressing ESC, the screen will display the inserted text correctly. Typing mistakes can be corrected in the usual manner, although you may choose to ignore the error and correct it later after returning to the command mode.

To insert one or more lines of text after the current line, type

o*anytext*ESC

This sequence opens up a blank line below the current line for the insertion. If an uppercase **O** is used, the lines are inserted above the current line. Also

I*anytext*ESC

and

A*anytext*ESC

insert text at the beginning (before first nonwhite character) and the end of the current line, respectively.

Inserting Special Characters

Many special characters cannot be entered into the buffer directly. For example, ESC terminates the insert mode, `^H` erases the previous character

typed, and the interrupt character (usually DELETE or `^C`) terminates insertion as well. RETURN separates lines but is not entered into the buffer as a character. To enter any of these characters into the buffer, you must precede them with `^V`which takes away the special meaning of the next character typed and forces **vi** to insert the character(s) literally: `^V` *escapes* or *quotes* the next character. Thus

```
i^V^H^VESC ESC
```

inserts the two characters `^H` and ESC. One instance where special characters are needed is in the `.login` file, where you may wish to redefine the Linux Shell-level erase and interrupt characters.

Erasing or Deleting Text

The **vi** editor provides many convenient ways to delete text from the buffer. To delete a few characters, the commands

*n***x** (delete next *n* characters)
*n***X** (delete previous *n* characters)

can be used. The value of *n* can be arbitrarily large, and an omitted *n* is assumed to be 1. The command

D (delete to end of line)

deletes from the cursor to the end of the line.

Joining two lines together is actually a deletion. To join the lines, you must delete the character at the end of the first line that marks the separation between it and the following line. To delete the line separation and join two lines in this manner, use

J (join next line)

The **vi** editor will not let two words run together; it will include a space between the last character of the first line and the first character of the second line.

One or several words can be deleted easily by

*n***dw** (delete next *n* words)
*n***db** (delete previous *n* words)

These commands erase the next and previous *n* words, respectively. Again, if *n* is omitted, it is assumed to be 1. Use

*n***dd** (delete *n* lines)

to delete the next *n* lines starting with the current line. Without *n* (*n* is 1), the current line is deleted. Depending on your terminal, **vi** may remove the deleted lines and close the gap by moving lines up from below, or it may use an `@` in front of an empty line to indicate a line deleted. These `@`'s are just markers and are not in the buffer. They will disappear when the screen is redrawn.

Sometimes a whole range of lines must be deleted. The command

`:`*i*,*j***d** (delete lines *i* to *j*)

deletes all lines from *i* to *j*, inclusive. The **vi** commands that begin with a colon (:) are actually **ex** commands being accessed by **vi**. Most of these commands use a range *i*, *j* where *i* and *j* are line numbers (addresses). The address *j* must not be less than *i*. In specifying line addresses, you may use

`$`	for the line at the end of the file
`.`	for the current line

Also allowed are such expressions as

`+4`	for current line plus 4
`-3`	for current line minus 3
`$-10`	for last line minus 10

and so on. For example, you would use

`:.,$d`

to delete from the current line to the end of the buffer. To delete the four previous lines and the current line (five lines all together), type

`:-4,.d`

Undo and Repeat

What if you make a mistake and delete something you did'nt mean to? The **vi** command

u (undo)

undoes the latest change made to the file. This command is very convenient and often lifesaving. (For example, what happens if you press **uu**?) The cut and-paste discussion (Section 2.10) has more on recovering deleted text.

Another convenience is the

. (repeat)

command, which repeats the latest change made to the buffer. For example, after giving the command 3**x**, you can move the cursor and press **.** to delete another three characters. To illustrate the use of this feature, let us consider *commenting out* code in a FORTRAN program. Suppose you want to insert the four characters `CC##` at the beginning of a number of consecutive lines. You position the cursor at the beginning of the first line, then press

I`CC##`ESC

From this point on, type the two characters

RETURN**.**

to comment out each successive line. RETURN moves the cursor to the beginning of the next line, and the period (**.**) repeats the last change, which is the desired insertion.

Making Small Changes

The **vi** editor includes several provisions for making small changes easily and efficiently.

r*char*	replaces the current character (at the cursor) by the given character
~ (tilde)	changes the case (upper to lower or lower to upper) of the current character
*n***cw***text*ESC	replaces the next *n* words by the given text. The extent of the text being replaced is marked on the screen by the cursor and a `$` as soon as *n***cw** is typed.
*n***cb***text*ESC	is the same as *n***cw** but replaces the previous *n* words
*n***s***text*ESC	substitutes the next *n* characters by the given text

The following are some useful combinations of commands.

xp	transposes the current character and the next character. This is really the command **x** followed by the command **p**.
easESC	pluralizes the current word. A combination of **e** and **a**.
ddp	interchanges the current line and the next line. A combination of **dd** and **p**.
dwwP	interchanges the current word and the next word
dwbP	interchanges the current word and the previous word

Text Objects and Operators

A text object is the portion of text defined by the current position in the buffer and by a cursor motion command. It is either a continuous *stream* of

characters or a section of consecutive lines that is bounded at one end by the current position and at the other by the place on which the cursor would appear after a cursor motion command were executed. For example, if the cursor is on the *m* of the word "men" in the following text:

```
Now is the time for
all good men to come
to the aid of their country.
```

then **2w** specifies "men to," **t**m specifies "men to co," **b** specifies "good," – the first two lines, and + the last two lines. You might think of this concept in terms of standing at one spot (the current position or current line) and tossing a stone (a cursor motion command). The area between where you are standing and where the stone lands is the text object you are interested in.

An *operator* is a **vi** command that operates on a text object, which is given to the operator as an argument. Two important operators are **d** (delete) and **c** (replace or change away). For example,

dG deletes the current line through end of file

The **d** is the operator and the **G** is the cursor motion command that specifies how far and in what direction to go from the current position to the other boundary of the text object. Here are a few more examples,

d$	deletes the current character through end of line
d)	deletes the current character to end of sentence
dt*char*	deletes the current character up to char in current line
c5w	changes away five words (use **b** in place of **w** for five words backward)
d/*text*ESC	deletes form the current character up to the *text* pattern

Another useful operator is the yank command **y**, which will be discussed in the next section.

Cut and Paste

One of the most useful operations an editor can provide is *cut and paste*. It reduces the task of rearranging major portions of a file to a matter of a few keystrokes. The **vi** editor provides auxiliary buffers that can be used for different kinds of cut-and-paste operations.

named buffers	a set of 26 buffers a-z
dtb	a single, unnamed, "deleted text buffer," (*dtb*), which always holds a copy of the latest deleted or changed away (**cw**, for instance) text
numbered buffers	the last nine blocks of deleted text are saved in a set of numbered buffers 1-9

The **vi** editor also provides commands to save a part of the main buffer in an auxiliary buffer and to copy the contents of an auxiliary buffer into the main buffer. This mechanism allows you to move text around easily in your file or between files.

Let's say you want to cut five lines from one place and move them to another place in the file. First move the cursor to the first of the five lines to be cut, then issue the command

5dd (delete five lines into *dtb*)

These deleted lines are held in the *dtb*. You then move the cursor to where you want to paste these five lines and use the *put* command

p or **P** (put *dtb* into buffer)

To insert a copy of the *dtb* into the main buffer. The exact position where the insertion takes place depends on the location of the cursor and whether the *dtb* contains whole lines or characters. If the *dtb* contains a number of whole lines (for example, our five lines saved by 5**dd**), the insert position is after (**p**) or before (**P**) the current line. If the *dtb* contains a sequence of characters (for example, text saved by 200**x**), the insertion is after or before the current character.

Copy into an Auxiliary Buffer: `yank`

To save text in a named buffer, you use the *yank* command **y**

"x**y***text-object* (copy object into buffer *x*)

where the first character is a double quotation mark. The *x* is any single-character buffer name (a-z). If the command **y** is not preceded by a buffer name, the *dtb* is used. Again, the *text object* to be yanked is specified by an appropriate cursor motion key sequence as explained in the last section. For example, to save the current line and the next nine lines in buffer **v**, you would type

"v**y**9RETURN

The **vi** editor will display the message

```
10 lines yanked
```

on the status line to confirm completion of the command. Note that the cursor and the lines of text stay exactly the way they were. Of course, you could use any cursor motion key sequence instead of 9RETURN to specify a text object. Similar to **y**, the command

"x**d***text-object*

deletes the specified text object into the named buffer *x*.

Paste: put

To copy text from a buffer *x* into the main buffer, type

`"`*x***p** (or `"`*x***P**)

and the content of buffer *x* will be inserted after (or before) the current line. If *x* is a numbered buffer *n*, the text inserted is the *n*th previously deleted text.

Direct Text Relocation: `move`

There is also a command to move lines of text directly from one place in the main buffer to another location:

`:`*i*`,`*j***m***k* (move lines *i* through *j* to *k*)

The range of lines *i* through *j*, inclusive, is moved after line *k*. For example,

```
:3,8m.
```

moves lines 3-8 after the current line. And

```
:.,.+5m$
```

moves the current line and the next five lines to the end of the file.

It is often useful to mark certain lines with symbolic names when rearranging text and to refer to these names instead of absolute line numbers. The command

m*x* (mark current position as *x*)

marks the current position with the single character name *x*. Any lowercase character can be used as a mark. To move the cursor to this line from anywhere in the buffer, you type

`'`*x* (go to line marked *x*)
`` ` ``*x* (go to character marked *x*)

Marked line addresses can be used in the same places as line numbers. For example,

`:'`*x*`,.`**m$**

move the text between the mark *x* and the current line (inclusive) to the end of file.

File-Related Commands

In simple applications of **vi**, the only necessary file manipulations are reading the file into the buffer initially and writing the buffer back out onto the disk at the end of the editing session. Files may, however, be accessed more flexibly than this. The main buffer can be written out to the file in mid-session. In fact, it is advisable to check your work occasionally by writing out the modified file as you progress. Otherwise, if the system crashed or you committed some dire error that erased the main buffer, you would lose whatever changes you had made and a great deal of time. The forms of the *write* command are

:w	writes back changes on disk
:w *file*	writes the buffer to *file*
:w! *file*	writes even if *file* exists
:*i*,*j***w** *file*	writes lines *i*-*j* to *file*

If *file* exists, **vi** will not overwrite it with the **:w** command, unless ! is added. It means "please carry out command as requested, I know what I am doing." The usage is a carryover from the **>!** type Shell-level commands.

It is often useful to read another file into the file being edited and combine the two files. You can use

:r *file* (read in *file*)

to read in the specified file and insert it after the current line.

If you wish to discard the current buffer and edit another file, you can type

:vi *file* (edit new *file*)

to replace the buffer by *file*. Whatever you have yanked from the old file will not be affected by this action and will remain in the auxiliary buffers. This command provides a way to cut and paste between files.

Text Substitution and Global Changes

Many times a file will require the same revision many times. For example, if you have written a letter about a man named John and discover that he spells his name "Jon," you will not relish having to fix each reference to his name individually. One of the most time-saving features provided by an editor such as **vi** is the ability to make *global* changes—to change each occurrence of a certain text pattern—with a single command. The more instances there are of the pattern, the greater will be the savings in time and effort. In this section we discuss several such global change commands. Complete, detailed descriptions of the global commands can be found in the **ex** reference manual. (Remember: The **vi** editor is an extension of the **ex** editor.)

Substitute

The command

:s/*pattern*/*text*/

is used to substitute *text* for the *first occurrence* of *pattern* in the current line. For example,

```
:s/summer/winter/
```

substitutes the first occurrence of `summer` with `winter` on the current line. The character `&` when used in the replacement text represents the pattern text. Thus,

```
:s/summer/each &/
```

substitutes `summer` with `each summer`. The substitute text can be empty, in which case the effect is deletion. If the substitute command is followed by **g**, that is

:s/*pattern*/*text*/**g**

then all occurrences in the current line are replaced.

The search and substitute operations can be combined into

:/*pattern1*/**s**/*pattern2*/*text*/**g**

which first searches for a line containing *pattern1* and then substitutes text for *pattern2* in that line. You can omit *pattern2* if it is the same as *pattern1*. Furthermore, the character `&` can be used in *text* to interpolate the pattern. For example,

```
:/Linux/s//& System V/
```

replaces the first occurrence of `Linux` with `Linux System V`.

The search patterns used can be more than simple strings. The way search patterns are specified is discussed in a later section.

Global Substitute

The global command **:g** allows you to specify a range of lines over which other **ex** commands, such as **s**, are executed. The format is

:*range***g**/*pattern*/**ex-***commands*

This means the *commands* are applied to all lines in the given *range* that contain the specified *pattern*. (If *range* is not given, the default range is all lines in the buffer.)

For example, to change the character string `LISP` to `Common LISP` throughout the file, you would use

`:g/LISP/s//Common LISP/g`

Note: If the trailing **g** is omitted, only the first occurrence on all lines will be replaced. Global substitute is powerful but dangerous. It does not give you a visual confirmation of each change. To remedy this, you may use an additional character **p** (after the trailing **g**); then each affected line is displayed on your terminal as the replacements are made. *Also beware*: Global changes, when issued incorrectly, can severely scramble the buffer. To guard against this, always write out the buffer to the file before doing a global replacement.

The **:g** can be applied to commands other than the substitute command. For example, to delete all lines containing the string `obsolete`, you would use

:g/`obsolete`/**d**

Indentation

Another useful global change is the indentation of text. The commands

n`<<`
n`>>`

indent and push out *n* lines starting at the current line. Each line is shifted by an amount set by the *shift width* option `sw` (normally eight spaces). Similarly, the commands

`<`*text-object*
`>`*text-object*

shift the entire text object, which must consist of whole lines. In other words, text objects of individual characters or words cannot be shifted in this way. For example,

`>G`

right-shifts from the current line to the last line in the buffer, whereas

`>L`

right-shifts from the current line to the last line on the screen.

Search Patterns

We have discussed the **vi** search command as a convenient way to move the cursor to specific locations in the file. The search command looks for a text string that matches a given pattern. The simplest pattern to find is a literal string of characters. For example,

```
/four score
```

searches forward for the literal pattern `four score`. If **?** is used instead of /, then a backward search is performed. A pattern extending beyond the end of one line into the next line cannot be specified in **vi**. This is a shortcoming of **vi** having evolved from the line-oriented **ex**.

Special Characters in Search Patterns

Special arrangements have been made for locating patterns at the beginning or end of lines. The character `^`(`$`) when used as the first (last) character of a pattern matches the beginning (end) of the line. Thus the string `four score` matches the pattern

```
^four score
```

only if it begins in column one on some line. Similarly, it matches

```
four score$
```

only when it is at the end of a line. If some blank space or nondisplaying characters follow, the match will fail, even though it may appear to you to be at the end of the line.

So far we have considered search patterns in which each character is specified exactly. In many situations, however, it is neither necessary nor desirable to specify the pattern literally and exactly. Let us look at a practical example: Consider editing a report that contains many items labeled sequentially by (1), (2), and so on. In revising the document, you need to add a few new items and renumber the labels. A search pattern can be specified as

```
/[1-9]
```

where the notation `[1-9]` matches any single character 1-9. With this notation, you can search using `/i([1-9]` the first time and then make the appropriate modification to the number. You then can repeat the search using the search repeat command **n**, change another number, search, and so on until all the changes have been made.

The **vi** editor offers a rich set of pattern notations (Table 0.1). They are referred to as regular expressions. To show the power of *regular expressions*, let us look at some specific matches

`[A-Z]`	matches any capitalized character
`\<[A-Z]`	matches any word that begins with a capital
`^##*`	matches one or more `#`'s starting in column one
`;;*$`	matches one or more ;s at the end of a line
`ing\>`	matches any word ending in `ing`

Quoting in Search Patterns

The use of special characters in any searching scheme inevitably leads to the question of how to search for a pattern that contains a special character. Let us say that you are editing a report and you want to search for `[9]`, which is a bibliographical reference used in the report. Because the pattern `[9]` matches the single character 9, you need to *quote* the `[` and `]` so that they represent themselves rather than taking on special meanings.

In **vi** the *quote character* \ (backslash) removes the special function of the character immediately following, forcing it to stand for itself. For example,

```
/\[9\]
```

Pattern	Meaning
x	A single character *x* with no special meaning matches that character.
x	Any character *x*, quoted by \, matches that character (exceptions: NEWLINE, <, >).
^	The character ^ matches the beginning of a line.
$	The character $ matches the end of a line.
.	The character . matches any single character.
[*string*]	A string of characters enclosed by square brackets matches any single character in *string*.
[*x*-*y*]	The pattern matches any single character from *x* to *y*. The notation is a shorthand for an explicit string.
[^*string*]	The pattern matches any single character not in *string*.
*pattern**	The pattern matches any repetition of *pattern* (including zero).
\<	The two-character notation matches the beginning of a word.
\>	The two-character notation matches the end of a word.

TABLE 0.1: **vi** Pattern Notations

searches for the literal `[9]`. Here are some other examples.

`/\.\.\.`	searches for ... (three dots)
`?\?!`	searches backward for `?!`
`/\/\*`	searches for `/*`
`/\\`	searches for \
`/[0-9A-z]`	searches for any of the indicated characters

Quoting a character that does not need quoting usually causes no harm.

Setting `vi` Options

The **vi** editor includes a set of options that the user can set to affect **vi** operation. These options help customize the editor for an individual user. There are three kinds of options:

- numeric options
- string options
- toggle (binary) options

Numeric and string options are set by

:set *option*=*value*

and *toggle options* are turn on or off by

:set *option* (turn *option* on)
:set `no`*option* (turn *option* off)

For example, `magic` is an option that is normally on. It allows for the special characters `.`, `[`, and `*` in search patterns. If you enter

:set `nomagic`

then these characters no longer will be treated specially in search patterns. To list options and their values for examination in **vi**, use the following commands:

:set lists option values different from the default
:set *option*`?` shows value of *option*
:set `all` lists all option settings

Several of the most useful options are discussed here.

`ic` The ignorecase option forces **vi** to ignore the difference between upper and lower case during searches. The default is `noic`.

`sm` The showmatch option bounces the cursor back momentarily to show you the opening bracket when the closing bracket is typed in the insert mode. This works for the commands **)** and **}**. If the matching bracket can't be found, **vi** beeps (if your terminal has the capability).

`nu` The nu option displays line numbers in front of the lines of text. The default value is `nonu`.

`ai` The autoindent option controls the automatic indentation of lines when text is inserted. When autoindent is set, a new line is started at the

same indentation as the previous line. This is especially useful when typing in programs. However, it is often the case that the supplied indentations are needed on some lines and not on others. To reject a supplied indentation, immediately after receiving it type: `^D` cancels one shift width (default eight spaces) of indentation. Suppose, for example, that after typing RETURN while inserting, autoindent supplies you with 16 spaces (or two tabs), and you only need eight spaces. You should type `^D` to move the cursor left eight spaces. The amount of left shift for each `^D` is itself an `sw` option. The default is `noai`.

`lisp` Setting the lisp option (default value is nolisp) facilitates editing LISP programs. The **vi** editor then reads LISP S-expressions as text objects. The commands **(** and **)** move the cursor to an S-expression. The commands **{** and **}** move over a balanced set of parentheses. The command `%` takes the cursor to the opposite balancing parenthesis. When lisp is set, the autoindent works differently; it supplies a consistent indentation level to align LISP expressions. One useful operation is positioning the cursor at the beginning of a LISP function and typing: `=%` This *pretty prints* the LISP structure by realigning the statements.

When a new **vi** job is initiated, all options have their default values. It often is desirable to set certain options whenever you use **vi** or **ex**. You can save options permanently by changing the Shell environment variable `EXINIT`.

`Vi` Macros

Occasionally, a sequence of editing commands is used repeatedly to make similar changes at different places in a file. If the sequence is long enough or used often enough, it may be convenient to assign a keystroke to save the sequence, which can then be used in its place. In **vi**, such named key sequences are called *macros*.

The map command establishes a named macro. The general form is

`:map` *key cmd-seq*

where *cmd-seq* is a string representing one or more **vi** commands. For example,

:map `v /Linux^V`ESC

establishes the character *v* as a macro to search for Linux in the buffer. When v is typed, it is as though the entire key sequence `/Linux`ESC is typed. Note the use of `^V` to quote ESC, so that it becomes part of the string and not taken as terminating the **map** command.

Consider editing a file where the word Linux is used frequently, sometimes in boldface and other times not. Let us say that you have decided to go through

the file and make Linux boldface in certain places. For LaTeX (Chapter 14) documents, boldface Linux is represented by

```
{\bf Linux}
```

in the text. Thus the editing task consists of searching for the pattern Linux and then for each occurrence, changing it to boldface if necessary. The **map** command is used to establish the macro `g`

:map `g i{\bf ^VESCea}^VESC`

Now the macro `g` changes any *word* to `{\bf` *word*`}`, assuming the cursor has been positioned on the first letter of *word*. With the preceding macros, an otherwise tedious editing task is reduced to typing `v`'s and `g`'s.

Mapped command macro names are restricted to a single keystroke, which may include a terminal function key. The command

:map `#`*n cmd-seq*

defines function key F*n* as a macro. Because `g`, `v`, and `V` are not already defined as **vi** commands, they are good choices for macro names. However, **vi** also allows you to save a command key sequence in a named buffer (`a-z`). Once the key sequence is stored in a named buffer, the command

@*x* (invoke macro in buffer *x*)

where *x* stands for the buffer name, will invoke the stored key sequence.

To cancel a macro use

:unmap *macro-name*

Input-mode Macros

In addition to making things easier in command mode, macros can also make text or program constructs easier to enter during input mode. This is done by the **map!** command in the form

:map! *macro-name macro-definition-string*

which defines an *input macro*.

Suppose you use the variable `BankAccount` a lot in developing a program, you can issue the **vi** command

:map! `` `B BankAccount ``

to establish the input macro `` `B ``. Now the character combination `` `B `` typed with normal speed during input mode will produce the full name desired. Thus, typing the input macro `` `B `` has the actual effect of entering its definition string.

During input mode, **vi** monitors each keystoke and checks if a character is the lead letter of an input macro. Thus, it is advisable to use an infrequent character to begin an input macro.

As another example, consider formatting a document in LaTeX. The sequence

```
\verb:word:
```

produces a constant-width font print out `word`. The input macro `\v` makes it easier to produce this construct

:map! `\v \verb::^V`ESC`hi`

To activate the input macro `\v`, the two characters must be typed together quickly. By typing slowly, you can avoid the effect of an input macro. For example, typing `\` and then `v` slowly in insert mode, the characters will be entered into the buffer and not taken as the `\v` macro. The command

:unmap! *macro-name*

cancels the given input macro. (This is a place to enter the *macro-name* slowly.) Settings of frequently used macros can be included on `EXINIT`.

Invoking the `vi` Editor

The routine usage of **vi** simply involves the command

vi *file*

However, this is not the only way to invoke **vi**. The usage synopsis is

vi `[-t` *tag*`] [-r] [+` *command*`] [-l] [-w`*n*`]` *file* `...`

The options and their meaning are explained.

`-l` This option is used to set up **vi** for editing a LISP source code file. When invoked with the `-l` option, **vi** automatically sets the options showmatch and lisp.

`-r` This option is used to recover after an editor or system crash, or to recover from failing to exit the editor before logging out. The command **vi** `-r` gives a complete listing of files saved that can be recovered. A typical output is

```
On Sun Jun 1 at 17:10 saved 4 lines of file
".exrc"
On Mon May 26, at 22:01 saved 28 lines of file
"community"
On Tue May 27 at 23:52 saved 67 lines of file
"hash.C"
```

to recover any of these saved files simply use:
vi `-r` *name*

`-w`*n* This option sets the *window size* to *n* lines. The default size at high speeds is 24, the size of most terminals. On dial-up lines, the default size is either 16 or eight lines, depending on the speed of the line.

`-t` *tag* This option is useful in maintaining a group of related source code or other text files. A *tag* is searched for in a file bearing the name tags in the current directory. The tags file is normally created by a command such as **ctags**. A tags file follows a simple format. It contains a number of lines, each with three fields separated by blanks or tabs: *tag-name filename* **ex**-*cursor-position-command*. The lines in a tags file are sorted. The effect of the command **vi** `-t` *tag*

is to search for a line in the tags file with *tag* in the first field. When found, the filename in the second field is loaded into **vi** for editing, and the cursor is moved to an initial position specified by the search command in the third field. For example, the file `/usr/lib/tags` contains system-wide tags information for Linux source code. The **vi** option tags contains the names of all tags files. The default setting is `tags=tags/usr/lib/tags`. A tag is searched for in sequence in the files given in the tags option.

`+`*command* This indicates that **vi** should begin by executing the specified *command*. For example, the argument `+50` places the cursor on line 50.

`Vi` Initialization: `EXINIT` and `.exrc`

The Shell environment variable `EXINIT` is used to initialize **vi** and **ex**, so that commands and option settings contained in `EXINIT` will be executed any time **vi** or **ex** is invoked.

For example, including the following line

```
setenv EXINIT 'set lisp ai sm ic'
```

in the `.login` file for *csh*, or the lines

```
EXINIT='set ai lisp sm ic'
export EXINIT
```

in the `.profile` file for *sh* users, sets the indicated options at program startup. The settings shown here are useful for editing LISP programs. The `EXINIT` can also be set to a sequence of **ex** commands (those accessed from vi with the `:` prefix) separated by the `|` character.

For example, the following can be put in your `.login` file

```
setenv EXINIT "se ai terse nowarn sm ic slow |\
                      map! \fo for( ; ; )"
```

In addition to this initialization mechanism, **vi** and **ex** also use the initialization file

```
.exrc
```

in a user's home directory. Commands contained in this file are processed as though they are typed interactively every time **vi** or **ex** is invoked. Option settings and often-used macros can be put in a `.exrc` file, an example of which is shown in Figure 1. Note, on some systems **vi** looks for `.exrc` in the current directory, or not at all.

The lines for `.exrc` file in Figure 1 sets the key `g` to italicize a word, `V` to boldface a word, and `v` to set a word in LaTeX *verbatim* mode. The symbol `^[` is the display representation of the nondisplayable character ESC. Similar lines can be placed in the `.vimrc` file.

FIGURE 1: Example `.exrc` file

```
map g i{\it ^[ea}^[
map V i{\bf ^[ea}^[
map v i\verb:^[ea:^[
```