A *vi* Primer

vi is a simple, visual editor. It is very fast, easy to use, and available on virtually every UNIX system. Although the set of commands is very cryptic, I learned it, as have thousands of others, which is proof that you can learn it too.

Notation

In the rest of these notes,

1. <CR> will denote the carriage return character obtained by pressing the *Enter* key on the keyboard and that we normally call the ***newline*** character.
2. <SP> will denote the space character, obtained by pressing the space bar. It will sometimes be necessary to represent this character as an underline "_" instead of <SP>.
3. <ESC> will represent the character obtained by pressing the escape key.
4. In general, angle brackets <…> around a word represents the character(s) described by the word, so for example, <positive integer> will mean any positive integer, and <char> will mean any single character.

Basics

1. Starting *vi*. That is easy. To edit a file named `myfile`, at the command prompt, type

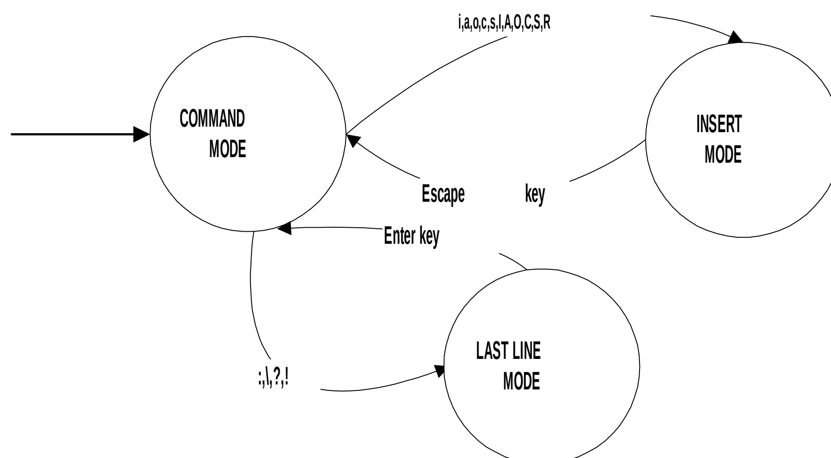
   ```
   vi myfile
   ```

2. *vi* is a 3-state finite automaton, with states

   COMMAND MODE

   INPUT MODE

   LAST-LINE MODE



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

When *vi* starts up, it is in **COMMAND MODE**. I.e., **COMMAND MODE** is the start state. The diagram shows the keystrokes that change *vi*'s state.

What you can do in each state and how you can change *vi*'s state is discussed below.

3. *vi* is *CASE*-SENSITIVE: everything you type must be in the correct case. If you read that "w" advances to the first character of the next word in the edited file (which it does) then you must use "w", not "W". The uppercase letter will do something else.
4. *vi* makes a copy of the file in a work buffer and places this in a temporary location (usually in `/tmp` or `/var/tmp`) often called the scratch directory in UNIX. On some UNIX systems, this is a world-readable file, in which case you should not edit documents you do not want people to see using *vi*. You can edit them on a non-timeshared, non-networked computer (your home computer) instead and upload them to the UNIX system. You can also use *vi* in a secure mode, in which it encrypts the work buffer. See the man page for *vi* for details – look up the -x option. First check whether the system is making the work buffers world-readable by running ls –l on the scratch directory. Note: if you do use the –x option for vi, no other program will be able to read the file, because vi will encrypt the saved file also. To decrypt the file, you will need to use the crypt command, as in:

```
crypt your_encryption_key < file_to_decrypt > cleartext_file
```

5. The cursor in *vi* is always ***on*** a character, ***not between*** characters as it is in Microsoft Word.
6. The ***start of a line*** is the leftmost NON-WHITE-SPACE character. Commands that move the cursor to the start of a line or modify the start of a line act on this first non-white-space character. If a line starts with leading blanks or tabs, these are not treated as part of the line when referring to the "start".
7. The ***end of a line*** is the last character before the <CR>, even if it is a white space character.
8. *vi* 's internal help command is the only way to get help. Type `:help`<CR> while in *vi*.

Overview of Command Mode

COMMAND MODE is the base camp for *vi*. It is like the campsite where you keep your tent. You go out on missions and return to camp each time, unless you decide to pack up and quit, in which case that is the last mission. So **COMMAND MODE** serves two purposes: (1) as a state in which to jump to other states, and (2) as a state in which to navigate, get text information, make global changes, and otherwise modify the document. In particular, the most important actions that you should know about are:

- Positioning the text insertion point, i.e., navigating,
- Displaying information such as where tabs and line feeds are,
- Redrawing the screen,
- Entering **INPUT MODE**,
- Entering **LAST-LINE MODE**,
- Substituting and deleting text,
- Cutting and pasting text.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Overview of Input Mode

The diagram shows that **INPUT MODE** is entered when one of several characters is typed in **COMMAND MODE**. The differences between what these different characters do will be discussed below. Regardless of how it is entered, once *vi* is in **INPUT MODE**, all text that you type becomes part of the document and **INPUT MODE** does not "know" how it was entered. The text is always placed to the immediate left of the cursor, i.e., the text insertion point (TIP) is to the left of the cursor. All printable text is inserted literally, including newlines. Control characters are interpreted, so you cannot insert them. To type control characters without their being interpreted first, type a Control-V (which I will write as ^V in these notes) followed by the character. For example, to insert a Control-C (^C) into a file in **INPUT MODE**, you would type ^V^C.

When you are finished entering text, you type the *escape-key* <ESC> to return to **COMMAND MODE**. (You *escape* to base camp.)

Overview of Last-Line Mode

From **COMMAND MODE**, one of the characters, :, \, ?, or ! puts *vi* into **LAST-LINE MODE**. These characters are commands. In **LAST-LINE MODE**, *vi* receives the input you type, followed by the Enter key (<CR>), and hands it to one of these commands. So remember the difference: **LAST-LINE MODE** is terminated by a <CR> and **INPUT MODE** is terminated by a <ESC>. The most important of these commands are used for

- Copying the work buffer to a file,
- Reading the contents of another file into the work buffer at the current cursor position,
- Making changes to a set of consecutive lines of the file,
- Setting line markers in a file,
- Yanking (copying to clipboard), copying sequences of lines to another location, deleting sequences of lines, or moving consecutive groups of lines from any part of the document to any other part,
- Quitting *vi*, saving or not saving the work, saving the work to the same or a different file,
- Searching (up or down) for lines that match a given pattern (extended regular expression).

Command Mode Operation

Navigation

vi treats the display as a two dimensional array of character positions. To move the cursor up, down, left, or right one position can be done with either the arrow keys on the keyboard, or with their character equivalents. The picture below illustrates. "k" is an upward movement for example, and "j" is downward.

```
  k
h   l    moves cursor one character up, right, down, or left as illustrated
  j
```

To move more than one character at a time, you have several options, because *vi* has many cursor movement options. These are summarized here. I have included many, but not all navigation commands. Movements based on the line structure of the text, are first. In some cases

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

I include more than one way to perform the operation. The arrow keys may not work on all systems, since it depends upon how the terminal has been configured. In all cases, the first key sequence listed is the one that is guaranteed to work.

If you are used to using an application like Microsoft Word then some of the concepts in vi require explanation. In Word, a paragraph is a sequence ending in a carriage return <CR>. In vi, a line ends in a <CR> and a paragraph ends in two consecutive ones, <CR><CR>. Also, vi defines a sentence as a sequence of words that ends in a sentence ending punctuation mark such as ".", "?", or "!".

Operation	Key Sequence(s)
Down one line	**j, Down Arrow, ^N**
Up one line	**k, Up Arrow, ^P**
Left one character	**h, Backspace, Left Arrow**
Right one character	**l, Space Bar, Right Arrow**
First non-white character on next line	**<CR>, +, ^N**
First non-white character on previous line	**-**
Goto line <line number>	**<line number>G**
Goto last line in file	**G**
Absolute left-most position on current line	**0 (zero)**
Left-most non-white character on current line	**^, Home Key**
Last character on the line	**$, End Key**
Go to character position on current line	**<position as a number>**

An integer replication factor can be prefixed to any command, not just navigation. For example, 3j means move down 3 lines. This integer is called a ***multiplier*** in *vi*.
The following are movements based on the word structure of the text.
.

Next word ignoring punctuation chars	**w**
Next word including punctuation chars	**W**
Previous word excluding punctuation chars	**b**
Previous word including punctuation chars	**B**
End of next word excluding punctuation chars	**e**
End of next word including punctuation chars	**E**

The definition of a word is somewhat complex. A word is a sequence of one or more word characters, but punctuation complicates it a little. If you want to skip across white space, use W. There are also commands to move across sentences and paragraphs:

Move to the start of the next sentence	**)**
Move to the start of the previous sentence	**(**
Move to the start of the next paragraph	**}**
Move to the start of the previous paragraph	**{**
Move right to the next matching character	**f<character>**
Move left to the previous matching character	**F<character>**

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Move right to position before matching character	**t<character>**
Move left to position after matching character	**T<character>**
Repeat previous f/F/t/T move	**;**
Repeat previous f/F/t/T move in reverse	**,**

Adding and Changing Text

Text is added to a document in **INPUT MODE**, which is entered from **COMMAND MODE** by typing one of the characters below. To return to **COMMAND MODE** from **INPUT MODE**, you type the <ESC> key. These are the most common and useful ways of adding text. There are others that I am not showing you.

Add Operations	Key Sequence
Append text to the right of the cursor	**a**
Append text at the end of the line	**A**
Insert text to left of the current cursor	**i**
Insert text to left of the first non-white character on the line	**I**
Add a new line below the current line	**o**
Add a new line above the current line	**O**

Change operations	
Replace text (replaces all text as you type.)	**R**
Change a word	**cw**
Change text through the end of the line	**c$**
Change text up to the next quote	**ct"**
Change the entire line	**cc**
Right shift the line	>
Left shift the line	<
Substitute input text for character(s)	**s/<pattern>/<pattern/<flags>**

This last operator, the substitute operator, is the single most important operator in vi. It is described in the section below entitled About the Substitute Operator.

If you want to replace a single character, you type **r<character>** and this will return to **COMMAND MODE**.

Deleting Text

Delete character to the right of the cursor	**x**
Delete character to the left of cursor	**X**
Delete current line	**dd**
Delete current word	**dw**
Delete to the end of the current word (does not delete delimiter)	**de**
Delete to end of sentence, paragraph, line, etc	**d), d}, d$ etc**.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

depending on symbol after d

Miscellaneous Useful Commands

Redraw the screen	**^L**
Change case of a letter	**~**
Join the current line with the next line	**J**

Copying and Pasting

When you delete text using the "d" operator or its variants, *vi* stores that text in a buffer. There are commands to paste the contents of the buffer to the current cursor position. There are also multiple buffers, but I will not discuss this here.

Paste the buffer contents to the right or below the cursor	**p**
Paste to the left or above the cursor	**P**

There are ways to copy text into the buffers without deleting it. This is called ***yanking*** in *vi*.

Copy word	**yw**
Copy to the end of line	**y$**
Copy line	**yy**

All of these can be preceded by a multiplier.

Last Line Mode Operation

LAST-LINE MODE is used for editing that affects a file globally, and for interacting with the file system and the operating system and shell. There are several ways to enter **LAST LINE MODE**. I start with the colon ":". When the ":" is typed, *vi* enters ***colon mode***. In colon mode, you can enter the following commands, all of which are followed by a <CR>, which returns *vi* to **COMMAND MODE**.

File Operations	Key Sequence
Save the buffer to the file opened by *vi*	**w**
Save a copy to the file named <file>	**w <file>**
Overwrite the file <file> if it already exists	**w! <file>**
Quit	**q**
Save and quit	**wq**
Quit without saving	**q!**
Read contents of <file> immediately after the current line	**r <file>**
Display name of current file	**f**

You can also create marks for lines in the file.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Make the character <char> represent the current line **k<char>**

In addition, in ***colon mode*** there are operators that can act on a range of lines. A range is of the form

<line expression>, <line expression>

where <line expression> is either a line number, a quote followed by a line mark, or a regular expression, described below. When an expression is ambiguous, meaning that it might match more than one line, it is taken to be the first of the matched lines. This can happen with regular expressions, since many lines might match an expression.

<u>Examples of Range Expressions</u>

Range	**Meaning**
1,5	lines 1 through 5
1,$	all lines in file
'x, $	lines from the one marked x to the last line
'a, 'b	lines from the one marked a to the one marked b
/aa*/,/bb*/	all lines from the first occurrence of a line matching pattern aa* to the first occurrence of a line matching bb*

In colon mode, you can use some of the operators already mentioned above, as well as some that can only be used in colon mode. The most powerful of colon mode operators is the substitute operator, *s*. To understand how to use it, you need to know about patterns. Therefore, I will delay discussing the substitute operator until after I present patterns to you. The operators that can be used in colon mode are:

delete	**d**
yank	**y**
right-shift	**>**
left-shift	**<**
substitute	**<range> s/target-pattern/replacement-pattern/flags**

<u>Examples</u>

:1,10d	delete the first ten lines
:6,$y	copy lines 6 to the end into a buffer
:1,$>	right shift the entire file

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

In addition, colon mode has the move and copy operators, *m* and *co*. These are a little tricky because the source range cannot overlap the target line. In other words, you cannot move lines 1 through 10 to line 5, or copy lines 1 through 10 to line 5.

Move the range of lines to the line <line>	<range-expression>**m**<line>
Copy the range of lines to the line <line>	<range-expression>**co**<line>

Searching

There is a way to search for patterns in the file using a pattern matching language much richer and more complex than the ones you will find in applications like Microsoft Word. The characters / and ? are pattern search operators. The / operator searches downward and wraps around to the top and the ? searches up and wraps around to the bottom. In short,

Search downward for pattern	**/<pattern>**
Search upward for pattern	**?<pattern>**
Repeat last search in the same direction	**n**
Repeat last search in the opposite direction	**N**

About Patterns

A pattern can be a simple string, or it can be a regular expression constructed by the rules described in the `regexp` man page. Type "`man regexp`" on the UNIX system to get the full story. This is only a brief synopsis of patterns. There are many more ways to construct them than are shown here.

1. Certain characters are ***special characters*** and have special meanings. These are period (.), asterisk (*) , left square bracket ([), backslash (\), caret (^), dollar-sign ($).
2. All other characters are one-character regular expressions.
3. The period (.) is a one-character regular expression that matches any character except NEWLINE.
4. The special characters can be matched by escaping them with a backslash: \$ matches $ and \[matches [for example.
5. Let p and q represent patterns (regular expressions). Then the following are also patterns

Pattern	Meaning
pq	matches any string that matches p followed by any string that matches q
p$	matches p only when anchored at the end of a line
^p	matches p only at the beginning of a line
\(p\)	matches p and saves the matching string in the next free register

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

[<list_of_chars>]	matches a single character in the list
[c1-c2]	matches a single character in the range c1 to c2 where c1 precedes c2 in ASCII ordering
[^<char_range>]	matches any single char except those in the specified range

Pattern	Meaning
c*	matches 0 or more consecutive c's
c\{n,m\}	match between n and m consecutive c's
c\{n,\}	match n or more c's
c\{,m\}	match up to m consecutive c's

This is just a start. I suggest you read the man page for regexp, since this is the most powerful part of *vi*, and these regular expressions also form the pattern matching language of *sed*, *awk*, and other tools.

About the Substitute Operator

The substitute operator, once again, is used with the syntax,

```
s/target-pattern/replacement-pattern/
```

or

```
s/target-pattern/replacement-pattern/flags
```

in **COMMAND MODE** and

```
<range expression> s/target-pattern/replacement-pattern/
```

or

```
<range expression> s/target-pattern/replacement-pattern/flags
```

in **LAST-LINE MODE** (meaning you typed a colon first.). If you omit `<range expression>` in **LAST-LINE MODE**, it applies the substitution only to the current line.

The substitute operator will search for the *first occurrence* in each line of the range or the current line only in **COMMAND MODE** and will replace the *longest matching occurrence* with the replacement pattern. If you want all occurrences on the line to be replaced, put a "g" where it says `flags` above. "g" means global.

Examples

Suppose the current line is

```
abcdeabcde   meeemeeemeee   12345123451234512345
```

Command	Resulting Line
`s/abcde/xxxxx/`	`xxxxxabcde   meeemeeemeee   12345123451234512345`
`s/abcde/xxxxx/g`	`xxxxxxxxxx   meeemeeemeee   12345123451234512345`

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

s/me*/X/	abcdeabcde	Xmeeemeee	12345123451234512345
s/me*/X/g	abcdeabcde	XXX	12345123451234512345
s/\(m.*\)<SP><SP>*/\1/	abcdeabcde	meeemeeemeee12345123451234512345	
s/\(.....\)\1//	meeemeeemeee	12345123451234512345	
s/\(.....\)\1//g	meeemeeemeee		

Comments

The first two are easy to follow. Adding the g flag makes the substitute apply to all occurrences of abcde. The third looks for a string consisting of 5 letters followed by the exact same 5 letters and deletes the first such occurrence. The fourth does this to all such strings on the line.

Useful or Enlightening Patterns

Some patterns arise often. Here are some I find most useful

[a-z][a-z]*	a lowercase word
[A-Z][a-z]*	a capitalized word
[a-z-][a-z-]*	a lowercase word with hyphens, like last-line

In the above example, the last hyphen is not part of a range; it is the literal hyphen character. UNIX regular expressions allow the hyphen to appear as the first or last character in the […] operator without having to be escaped. So the next one is equivalent.

[-a-z][-a-z]*	same as above. This is another way to do it
[]]	matches right square bracket . The [must be the first character after the opening [otherwise it must be escaped with a backslash as in the next example.
[\]]	same as above
[1-9][0-9]*	a positive integer
[<SP>][<SP>]*	white space (the <SP> is the white space character. See Notation paragraph above.)
[a-zA-Z_][0-9A-Za-z_]*	C++ identifier
\([a-z][a-z]*\)\1	words of the form ww, where w is a word.
^$	blank line
[^a-zA-Z0-9][^a-zA-Z0-9]*	sequences of characters other than letters or digits

Example 1

Suppose I have a file consisting of lines of the form

```
   Albert Sionov        asionov         shiva
      Andrzej Such         asuch          shiva
         Jacek Szulc          jszulc        hejira
   Khai Tran              ktran        hejira
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Karl Treen ktreen hejira

This is a list of first name, last name, user-name, host-name records. The amount of white space varies from one line to another.

Suppose I want to convert it to the form

asionov@shiva Albert Sionov
asuch@shiva Andrzej Such
jszulc@hejira Jacek Szulc
ktran@hejira Khai Tran
ktreen@hejira Karl Treen

in which the email address is separated from the first name, last name by a tab, and the first name and last name are separated by a space character. I can do this in several steps.
First, I start by deleting leading white space:

```
:1,$s/^ [ ]*//
```

The target pattern is a sequence of one or more space characters anchored to the start of the line. The replacement pattern is empty. The effect is to delete all leading white space:

Albert Sionov asionov shiva
Andrzej Such asuch shiva
Jacek Szulc jszulc hejira
Khai Tran ktran hejira
Karl Treen ktreen hejira

I then replace the white space between the username and the last word on the line by a "@". Since I want this to work for any last word on the line, I use the pattern [a-z][a-z]* to represent a sequence of one or more lowercase letters. To anchor to the end, I can use [a-z][a-z]*$. So I start to write:

```
:1,$s/ [ ]*[a-z][a-z]*$/
```

but how do I get the word that was matched to appear again. That is where the \(…\) brackets work. I tell *vi* to remember the matched string, which it places in the variable \1. I then use it in the replacement string:

```
:1,$s/ [ ]*\([a-z][a-z]*\)$/@\1/
```

The result is

Albert Sionov asionov@shiva
Andrzej Such asuch@shiva

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Jacek Szulc	jszulc@hejira
Khai Tran	ktran@hejira
Karl Treen	ktreen@hejira

Finally, I want to make the last word on the line the first word, and the first two words, the last two. I can do this by remembering the three words and re-ordering them.

```
:1,$s/\([a-zA-Z][a-zA-Z]*\) [ ]*\([a-zA-Z][a-zA-Z]*\) [ ]*\(.*\)$/\3 \1 \2/
```

This is in the form

```
:1,$s/\(pattern1\) \(pattern2\) [ ]*\(pattern3\)/ \3  \1  \2/
```

which saves the first match into \1, saves the second into \2 and the third into \3. The first and second patterns match words that have upper and lowercase letters. The third matches any string at the end of a line. The result is

asionov@shiva	Albert Sionov
asuch@shiva	Andrzej Such
jszulc@hejira	Jacek Szulc
ktran@hejira	Khai Tran
ktreen@hejira	Karl Treen

I could have done this transformation using a single substitute command, but it would have been a lot of typing, and one small mistake would have required retyping the whole line, so it is not worth the challenge. It may not look like this on your screen because the tab character settings determine how close the second column will be to the first.

Example 2

Suppose that you want to modify a file containing ordinary text consisting of sentences and paragraphs so that every sentence starts on a new line. Assume that all sentences end in either a period, question mark, or exclamation mark, and that a new sentence is separated from the previous sentence by one or more spaces, as is the convention. Then you need to replace the sequence consisting of the end-of-sentence mark, one or more spaces, first character of new sentence by the sequence consisting of the end-of-sentence mark, newline/carriage-return, first character of new sentence. Note that sentences that start paragraphs will not be changed if you do this because they already have a newline/carriage-return preceding them. The *vi* command to do this is

```
:1,$s/\([\.\?\!]\) [ ]*\(.\)/\1^V^M\2/g
```

Notes

1. When you type the Control-V (^V) it will not be visible. When you then type the Control-M, it will appear as a ^M on the command line.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

2. The g on the end is the global flag, which means that if there is more than one match on a line, it should apply the substitution to all matches, from left to right.
3. The characters ".", "?", and "!" must be escaped inside the […] operator.
4. The \1 and \2 are the strings that matched the first \(…\) and the second \(…\) respectively.

Example 3

Suppose that you want to change a C or C++ program so that every assignment operator has white space on either side of it. You are not sure whether they all do – some might and some might not. The following two vi substitution will do the trick.

```
:1,$s/\([^ =]\)=/\1 =/g
:1,$s/=\([^ =]\)/= \1/g
```

The first pattern is any character other than space or "=" followed by a "=". If there is a match, this two character sequence is replaced by the matched character then " =". The second pattern is the symmetric equivalent. I exclude the "=" because I do not want to modify equality comparisons. The "g" flag is just in case there are multiple assignments on a row.

Resources

http://www.eng.hawaii.edu/Tutor/vi.html

A web site with a good tutorial on vi. But be warned – they view vi as having only two states, not three.

http://www.download.com/Software-Online/3260-20_4-54451.html

This is the page from which you can download *Lemmy* as well as *WinXS*. Lemmy is a UNIX vi application that runs on almost all versions Windows. It lets you use almost all of the standard vi editing commands and also supports the traditional Windows cut and paste features. It is not exactly like vi, but close enough. It is worthwhile to download Lemmy and use it. When I write programs for UNIX on my Windows system at home, I write them in Lemmy and then upload them to the UNIX system via an FTP application.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.