VIVA vi !

deboor@cs.wisc.edu

as of 22mar01

TABLE OF CONTENTS

Enter **vi** via the (system-)command

```
vi <filename>⟨RET⟩
```

which makes the contents, if any, of <*filename*> available for editing.
Leave **vi** via the (**vi**-)command

```
ZZ
```

which saves the current text in <*filename*>.
If you want to quit, i.e., leave without changing the contents of <*filename*>, use instead

`:q`⟨RET⟩ or `:q!`⟨RET⟩

If you spend much time on a file, it is good to use

`:w`⟨RET⟩

occasionally to write the current version back into <*filename*> (just in case the machine goes down). Thus

`:wq`⟨RET⟩

has the same effect as `ZZ` (or `:x`⟨RET⟩) (except that the latter two *only* write if some change was made).
Ordinarily, the cursor is somewhere in the text of the file. However, if it is at the bottom of the screen after a colon, you are in the line-editor **ex** (onto which **vi** was built). You get into this editor temporarily, by typing a colon while in command mode (as was shown above already). But it is also possible to get into this editor permanently, perhaps accidentally, by the command `Q`. If that is not what you wanted, type

`vi`⟨RET⟩

and you are back in **vi** again.

MOST USEFUL COMMAND

```
u
```

undoes the last thing you did, while

```
U
```

restores the current line to its original state. This encourages experimentation!
Very useful command:

```
.
```

(a period) repeats the last change you made (but at the current position).
Obscure but very useful command:

⟨CON-v⟩

(made by pressing `v` while holding down the **control key**) causes the next keystroke, such as ⟨ESC⟩ or ⟨RET⟩, to be treated as text rather than a command. See SUBSTITUTIONS, MACROS.

ENTERING TEXT

There are two modes, **command** mode and **text** mode. The command mode is the basic mode, and you are in command mode when you start **vi**. You enter text mode only temporarily, in order to add text to the file, and then terminate text mode and return to command mode by pressing the **escape key**

⟨ESC⟩

This makes it possible to have many commands consist of one character only and makes it easy to string them together for more complex commands or macros. For *example*, the command `x` deletes the character at the cursor and positions the cursor on the next character, while the command `p` places the most recently deleted material just to the right of the cursor; hence the composite command `xp` interchanges two adjacent characters.

The fact that the basic mode in **vi** is the command mode has caused many people to avoid **vi** since people are used to editors that are always in text mode unless one uses some escape sequence to enter command mode temporarily. In such editors, anything you type is added into the file unless a special escape character is typed to indicate that the next item typed is to be taken as a command. However, this makes the issuing of commands a bit complicated, particularly if one wants to string commands together (into a macro).

In **vi**, you enter text mode only temporarily, for inserting, appending or otherwise changing or substituting text (also by opening the next line or doing any of this in capital). Further details on these commands `i`, `I`, `a`, `A`, `c`, `s`, `o`, `O` will follow.

While in text mode, you can wipe out the most recently typed

$$\left\{\begin{array}{l}\text{character}\\ \text{word}\\ \text{line}\end{array}\right\}\text{ by using }\left\{\begin{array}{l}\langle\text{BACKSPACE}\rangle\\ \langle\text{CON-w}\rangle\\ \langle\text{KILL CHARACTER}\rangle\end{array}\right\}.$$

Here and below, ⟨CON-$<l>$⟩ is typed by typing the character $<l>$ while holding down the **control key**.

To insert text before (append text after) the current position, use the command

```
i (a)
```

after which anything you type is interpreted as text (with some surprising results if you press keys other than the standard character keys) until you press

⟨ESC⟩

which gets you back into command mode again. The related command

```
I (A)
```

starts insertion at the beginning (end) of the current line. To start insertion at a new line (i.e., open a new line) below (above) the current one, use

```
o (O)
```

If you want automatic line-wrap-around (i.e., have the editor start a new line with the current word as soon as that word threatens to run beyond the right margin), issue the command

`:set  wm=`$<n>$

with $<n>$ a positive integer, e.g., 3. This sets the right margin to be $<n>$ columns from the right screen edge.

If you want an automatic check on just what `(` or `{` or `[` is being matched by the `)` or `}` or `]` as you type it, use

```
    :set sm
```

If you want to see just what other such options are available and what their current setting is, use the command

```
    :set all
```

Save yourself the trouble of typing such default setting commands all the time by putting them (*without* the initial colon `:`) into the file `.exrc` in your root directory. If there is no such file, create it. **vi** will execute all the commands in `.exrc` every time you use **vi**.

MOVES

The next character to the left, below, above, to the right, is reached with the aid of the arrow keys $\leftarrow, \downarrow, \uparrow, \rightarrow$ or by `h, j, k, l`. In principle, repeated use of these keys can get you anywhere, but there are many other moves available. These moves are not only convenient but are important for *modifying text* since many commands for modifying text are of the form <*action*> <*move*> to indicate that the action is to involve the text between the current position and the destination specified by <*move*>. E.g.,

```
    d$
```

indicates that the text from the current position to the end of the current line is to be deleted. The moves usually come in pairs, to be made forward (i.e., to the right and/or down) or backward (i.e., to the left and/or up), with the forward (backward) move usually (not) including the current position.

backward		**forward**
`` ` ` ``	position most recently jumped from	`` ` ` ``
`' '`	beginning of line most recently jumped from	`' '`
$<n>$`|`	column $<n>$ within line	$<n>$`|`
`0`	farthest character within line	`$`
`^`	first non-blank character in line	
`F`$<c>$	find nearest occurrence of character $<c>$ within line	`f`$<c>$
`T`$<c>$	up to nearest occurrence of character $<c>$ within line	`t`$<c>$
`b`	beginning of nearest word	`w`
`B`	Beginning of nearest “Word”	`W`
`b,B`	beginning/end of current word, “word”	`e,E`
`-`	beginning of nearest line	`+`
`(`	beginning/end of nearest “sentence”	`)`
`{`	nearest empty line	`}`
`[[`	beginning/end of nearest “section”	`]]`
`-`$<n>$⟨RET⟩	beginning of $<n>$th line	$<n>$⟨RET⟩

`<n>G`	Go to beginning of line numbered *<n>*	`<n>G`
`1G`	beginning of the farthest line	`G`
`H`	(High or Home) beginning of farthest line on screen (Low or Last)	`L`
`M`	beginning of Middle line on screen	`M`
`%`	matching parenthesis, brace or bracket	`%`
`?`*<pattern>*⟨RET⟩	nearest occurrence of *<pattern>*	`/`*<pattern>*⟨RET⟩
`?`⟨RET⟩	nearest occurrence of most recently used *<pattern>*	`/`⟨RET⟩

For *example*, if you wanted to change a *matching* pair of parentheses to brackets and the cursor is at the opening parenthesis, then the following would do the job:

```
%r]``r[
```

Neat, right?

In addition,

```
; (,)
```

repeats the last `F`,`f`,`T`,`t` command in the same (opposite) direction, and

```
n (N)
```

repeats the last `/`,`?` command in the same (opposite) direction. Also, the move

```
`<l>
```

gets you to the position marked earlier by the command

```
m<l>
```

while

```
´<l>
```

gets you to the beginning of the line containing that marked position.

PATTERN (or CONTEXT ADDRESS)

While all other moves in the table above are pretty clear, the last move mentioned, i.e., the move to the nearest occurrence of *<pattern>*, requires some explanation of *<pattern>*. This explanation will also be needed later, in the discussion of pattern substitution.

Offhand, a pattern is a string of characters (i.e., any string you could type), and, whenever you type `/`*<pattern>*`/` or `?`*<pattern>*`?`, **vi** searches forward or backward through the text to find a matching string of characters. However, some of the characters you can type have been declared by **vi** to be "magic". This means that they do not stand for themselves but are used to describe a pattern *feature*. For *example*, the pattern `abe` is just what you think it is, it would find a match in such words as `Rabelais` or `Kemosabe`. On the other hand, the pattern `ab*e` would also find a match in `faerie` or `jabber` since the character `*` is used to indicate zero or more occurrences of the character immediately preceding it. What do you do when you do want to match one of these magic characters? You precede it with the backslash `\`. In particular, the pattern `\\` matches a backslash.

The use of an extensive list of "magic" characters in the description of patterns is perhaps **vi**'s best, yet least understood, feature. Here is a more formal description.

A **pattern** (or **context address**, or **regular expression**) consists of characters (upper and lower letters, digits, and other typographical signs), and these match themselves, except for the **wild cards** or **magic** characters

```
        .  [  ]  *  ^  $  \  &   ~  ?  /
```

and these must be preceded by a `\` (backslash) if they are to match themselves (in a situation in which they could be considered magic).

`.`	(a period) matches any one character
`^`	(a caret) at beginning of pattern, matches beginning of the line
`$`	at end of pattern, matches end of the line
`\<`	matches the beginning of a word
`\>`	matches the end of a word
`[`<*string*>`]`	matches any one in the list <*string*>
`[^`<*string*>`]`	matches any one not in the list <*string*>
`*`	matches zero or more occurrences of the preceding character
`\(`<*pattern*>`\)`	indicates a pattern part

Here, a list <*string*> consists of one or more characters and may use the hyphen to indicate a whole group of characters; e.g., `c-f2-5AB-E` is an acceptable abbreviation for the list `cdef2345ABCDE`, and `A-z` stands for all letters great and small (and the characters `[\ ] ^ _ ``` ` ``` besides). This means that a hyphen must be the first item in such a list if it is to be part of the list.

For *example*, the next

$$\left\{\begin{array}{l}\text{empty}\\ \text{nonempty}\\ \text{blank}\end{array}\right\} \text{ line is indicated by } \left\{\begin{array}{c}\texttt{/\^{}\$/}\\ \texttt{/./}\\ \texttt{/\^{} *\$/}\end{array}\right\}, \quad \text{while } \texttt{/\^{}\textbackslash([\^{}\&]*\textbackslash)\&/}$$

picks out the next line with an ampersand in it and makes the text from the beginning of the line up to the ampersand available for subsequent SUBSTITUTION (see below).

Note that patterns are only found within a line. In particular, the <*newline*> character *cannot* be part of a pattern (except that, by finishing your pattern with `$`, you can insist that the match terminate with the end of the line). Also, the pattern matched is always *as long as possible.* E.g., the pattern `/(.*)/` will match the *first* open paren and the *last* closing paren in a line (and everything in between) even if these two parens don't match as far as the move `%` is concerned. To find an innermost matching pair of parens, you could use the pattern

```
        /([^()]*)/
```

MODIFYING TEXT

delete

To delete or expunge the current (previous) character, use

```
        x   (X)
```

Material consisting of more than one character is deleted by

```
d<something>
```

with <*something*> specifying the extent of the deletion.

```
d<move>
```

deletes from the current position to the point specified by the <*move*>, while

```
dd
```

deletes the current line. There is also the **ex** command `:<range>d` for deleting several lines; see RANGES below.

change and replace

The change command

```
c<move>
```

leaves you in text mode. The entire text between the current position and the destination of the <*move*> will be replaced by what you type until you press

⟨ESC⟩

The command

```
R
```

also leaves you in text mode but Replaces characters as you type until you press

⟨ESC⟩

Finally, the command

```
r<c>
```

replaces the current character by the character <*c*> and leaves you in command mode.

changing the case

The command

```
~
```

(tilde) changes the case of the current character, i.e., to upper (lower) case if it is lower (upper) case. See SUBSTITUTION below for selective change of case via a global substitution.

joining two lines

To Join two lines, put the cursor anywhere in the first line and use

```
J
```

MOVING AND COPYING TEXT

The most recently deleted thing can be placed before (after) the current position by

```
P (p)
```

Thus the composite command

```
xp  (Xp)
```

has the effect of transposing the current (previous) character with its neighbor to the right (the composite commands `xP` and `XP` do nothing), the composite command

```
dwBP
```

(issued when the cursor is at the beginning of a word) interchanges that word with the word *preceding* it on the line, while

```
ddp
```

transposes the current line with the line below it.

Actually, the 9 most recently deleted items are kept in **numbered buffers** named $1, 2, \ldots, 9$ and can be placed (at the present position) by

```
"<d>p
```

with *<d>* the appropriate digit.

There are additionally 26 **labeled buffers**, labeled by the 26 letters of the alphabet, into which a portion of text can be deleted by the command

```
"<l>d<something>
```

If the corresponding capital letter is used, i.e.,

```
"<L>d<something>
```

the material is *appended* to what is already in that buffer. The text so saved can be placed before (after) the current position by

```
"<l>P  ("<l>p)
```

Finally, it is possible to put text into buffers *without* deleting it by using the yank command instead of the delete. E.g.,

```
y0
```

yanks, i.e., copies, into the unnamed buffer, the text from the beginning of the current line up to (but not including) the present position.

Since the contents of the *labeled* buffers remain intact when starting to work on another file by the

```
:e  <filename>⟨RET⟩
```

command, this provides one mechanism for moving material from one file to another. If, after placing the buffer content in that file (followed by `:w`⟨RET⟩ to save the modified file), you want to return to the file from whence you came, use

```
:n#⟨RET⟩   or   :e#⟨RET⟩
```

REPETITION

Most commands can be repeated <*n*> times by prefixing the number <*n*>. E.g.,

`4yy`

yanks the current and 3 subsequent lines into the unnamed buffer.

REDRAW and REPOSITION

If incoming messages or a slightly misspecified terminal type have caused uncertainty about your current display, use

⟨CON-l⟩ or ⟨CON-r⟩

to redraw the screen. Use

⟨CON-b⟩ (⟨CON-u⟩ ⟨CON-d⟩ ⟨CON-f⟩)

to scroll -full (-half half full) screen. Reposition current line to top (middle bottom) of screen by

z⟨RET⟩ (z. z-)

LINE NAMES and RANGES

vi is built upon the line editor **ex**, and commands from the latter are available in **vi**. These are the commands that start with `:` (a colon), hence are also called **colon commands**, and finish with a ⟨RET⟩. We have already mentioned a few, namely `:w`, `:q`, `:e`, `:n` , and `:so`. Many **ex** commands, such as `:co` (copy), `:d` (delete), `:j` (join), `:l` (list), `:m` (move), `:p` (print), `:s` (substitute), and `:w` (write), are meant to operate on a group of lines, as specified by a **range**. For such commands, the **ex** command structure is

`:`<*range*><*action*>⟨RET⟩

For *example*,

`:`<*range*>`m`<*line*>⟨RET⟩   (`:`<*range*>`co`<*line*>⟨RET⟩)

moves (copies) the specified lines to the place right after the line named <*line*>. Here, <*range*> specifies a range of lines, usually in the following form

<*line*>,<*line*>   <*line*>;<*line*>

with the semicolon used in place of the comma when one wants the first line of the specified range to become the current line for the remainder of the command (including for the interpretation of the last line specification), and with <*line*> one of the following names for a line (with *n* denoting an unsigned integer):

n	the *n*th line of the file (default: the current line)
0	the (nonexisting) line before the first line of the file
`.`	the current line
`$`	the last line
<*line*>$\pm n$	the line whose number differs by $\pm n$ from that of the line named <*line*> (default: $n = 1$)
´<*l*>	the line (previously) marked <*l*> by the command `m`<*l*>

`''`	the line most recently moved from
`/`<*pattern*>`/`	the next line forward containing the pattern <*pattern*>
`?`<*pattern*>`?`	the next line backward containing the pattern <*pattern*>

For *example*, `$-3` is the fourth line from the bottom of the file, while `?default?+1` is the line below the nearest line above the current one with the word `default` in it. The range `/The/,+5` has the first line below the current line containing `The` as its first line, and the line 5 lines down from the current one as its last line, while the range `/The/;+5` contains six lines, starting with the first line down from the current one containing `The` and including the five lines following it. As a more complicated *example*, if the most recent command was `{}` (i.e., the move to the empty line preceding the current paragraph followed by a move to the empty line following the current paragraph), then the range `'',.` describes all the lines in the current paragraph (including those two empty lines), while the range `'',/The/` describes all the lines from the beginning of the paragraph just outlined to the first line below the paragraph that contains the string `The`.

Use

`:`<*line*>`=`⟨RET⟩

to get a temporary printout (on the bottom of the screen) of the number of the line specified by <*line*>. In particular, `:.=`⟨RET⟩ gets the line number of the current line, as do `:f`⟨RET⟩ and ⟨CON-G⟩.

The range specification <*line*> (consisting of just one line) specifies the range consisting of just that one line.

As indicated in the above table, the **default value** for <*line*> is the current line. E.g., `+` is the line below the current line, `:=`⟨RET⟩ gets the line number of the current line, while `,+` is the range consisting of the current line and the line following it, and `,` is the same as the default value for <*range*>, namely the current line (except for the `:w` command for which the default range is the entire file, i.e., the range `1,$` (which may be abbreviated `%`)).

Finally, it is possible to narrow down the lines within a given range to be acted upon by using **ex**'s full command structure

`:`<*range*1>`g/`<*pattern*>`/`<*range*2><*action*>⟨RET⟩

Here, the default value for <*range*1> is the entire file, each of the lines within <*range*1> containing <*pattern*> becomes the current line in turn, and the default for <*range*2> is the current line. This **global** specification provides remarkable flexibility. For *example*, the command

`:/{/;/}/j`⟨RET⟩

will locate the next line containing a left brace, then combine it with all succeeding lines until it has adjoined a line containing a right brace. The global version

`:g/{/;/}/j`⟨RET⟩

of this command has the following effect: Each line containing a left brace becomes in turn the current line, the range `,}` therefore specifies all the lines from that current line to the next line containing a right brace, and the specified <*action*> `j`⟨RET⟩ is, as before, to join all these lines. In other words, the joining of such groups of lines now takes place globally, in the entire file. The modified global version

```
:1,/CHAPTER 2/-1g/{/;/}/j⟨RET⟩
```

will join such blocks of lines only if they occur before the first line containing the string `CHAPTER 2`.

As another *example*, suppose that you have a file containing blocks of information (e.g., a list of references, or of addresses) separated by a single empty line and you want to move, in each block, the third line of the block to the top of the block. This is accomplished by the following simple command

```
:g/^$/+3m.⟨RET⟩
```

Using `v` (for Latin's *vel*=other??) or `g!` instead of `g` specifies all lines that do *not* contain the given <*pattern*>. For *example*, the command

```
:v/^ *$/s/$/XXX/⟨RET⟩
```

affixes to the end of every nonblank line the string `XXX`.

If you are uncertain just what your <*range*> specifies, print it all (on your screen) with the command

```
:<range>p⟨RET⟩
```

to have a look at it, then use ⟨RET⟩ to get rid of this (temporary) printout.

COPYING MATERIAL FROM/TO ANOTHER FILE

A second way to move material from/to another file uses the `:read` and `:write` commands.

```
:<range>w  <filename>⟨RET⟩     ( :<range>w >><filename>⟨RET⟩ )
```

writes (appends) the specified range of lines to the specified file, while

```
:r  <filename>⟨RET⟩
```

reads the specified file into the current file after the current line.

(GLOBAL) SUBSTITUTION

vi lets you change text in very sophisticated and handy ways, by replacing one pattern by another. This mechanism is very powerful, because of the magic characters allowed in the description of patterns (see PATTERN above).

The commands

```
:<range>s/<oldpattern>/<newpattern>/⟨RET⟩

:<range>s;<oldpattern>;<newpattern>;⟨RET⟩
```

substitute the <*newpattern*> for the first occurrence of the <*oldpattern*> found in each line within the specified <*range*> (with the current line the default range, and the pattern used in <*range*> the default for <*oldpattern*>). The second version is convenient when a slash (but not a semicolon) is part of the pattern(s). For *example*, the command

```
:%s/ .*$/⟨RET⟩
```

will truncate every line in the file at the first blank, while the command

```
:/\<onto\>.*/s//into⟨RET⟩
```

will find the next line containing the word 'onto' and replace the text there from 'onto' to the end of the line by 'into'; note that the final `/` or `;` is optional. The command

```
:g/^[^ ]/s/^/copy ⟨RET⟩
```

will insert 'copy ' at the beginning of every line that does not begin with a blank. If *all* occurrences in a line are to be replaced, follow the above command with a `g` (for **global**); if you want a say in which occurrences are to be replaced, follow the command with a `c` (for **conditional**). Thus, the command (which needs the `/` after the second pattern)

```
:.,$s/copy /delete /gc⟨RET⟩
```

will look at all lines from the current one to the last one in the file, show you each occurrence of 'copy ' and wait for a `y`⟨RET⟩ to change it to 'delete '; responding with anything other than `y`⟨RET⟩ will leave that occurrence of 'copy ' unchanged. (A more hands-on procedure with the same effect is to type `/copy `⟨RET⟩ (which will get you to the first character of the next occurrence of 'copy '), type `cedelete`⟨ESC⟩(to change that word to 'delete'), then type `n` (to get you to the next occurrence of 'copy'), and, if it is also to be changed, type `.` (period), and so on.)

Part or all of the material matched by <*oldpattern*> can be used in <*newpattern*>, as follows. `&` stands for the entire material matched by <*oldpattern*>, while `\`<*n*> stands for the <*n*>th part, i.e., the part between the <*n*>th `\(,\)` pair. For *example*, if you have a file produced by a directory command, with lines of the type

```
ALIASES.CS;1 4/6 16-MAY-1988 12:20 [DEBOOR] (RWED,RWED,RE,RE)
ALWTEX.COM;1 7/9 21-DEC-1987 11:04 [DEBOOR] (RWED,RWED,RE,)
....
```

and you want to make from it a command file that moves all .com files to the subdirectory [mrc.deboor.command], you could use the following two commands:

```
:v/\.COM;/d⟨RET⟩
```

(which deletes all lines not containing the string '`.COM;`'), and

```
:%s/^\([^;]*\)\(;[^ ]*\) .*$/rename \1\2 [mrc.deboor.command]\1/⟨RET⟩
```

which leaves you with the file

```
rename ALWTEX.COM;1 [mrc.deboor.command]ALWTEX.COM
....
```

As another *example*, if, in a TEX file, you would like to have all `$$` appear on a separate line (to make it easier to find all 'displays'), then you might try the command

```
:%s/$$/⟨CON-v⟩⟨RET⟩&⟨CON-v⟩⟨RET⟩/g⟨RET⟩
```

Note how each ⟨RET⟩ here is preceded by a ⟨CON-v⟩, to tell the input routine that the ⟨RET⟩ is *not* an actual ⟨RET⟩ but is part of the pattern. Now, this command is faulty on two counts: (1) the last character of the first pattern is `$`, hence it will match an end-of-the-line rather than a `$`, unless we precede it with a backslash; (2) the command will put a newline character in front of the `$$` even if `$$` starts a line, and will follow up `$$` with a newline character even if `$$` is already at the end of a line. Therefore, it is better to use the following *two* commands:

```
:%s/$$\(.\)/$$⟨CON-v⟩⟨RET⟩\1/⟨RET⟩        :%s/\(.\)$\$/\1⟨CON-v⟩⟨RET⟩$$/⟨RET⟩
```

which inserts a newline character only *between* a `$$` and some actual character.

As another *example of a control character within a pattern*, here is a quick way to generate a sequence of lines containing the statements `\input file1`, `\input file2`, ..., `\input file9`: On an empty line, type the string `123456789`; then, with the cursor at that line, issue the command

```
:s/[1-9]/⟨CON-v⟩⟨RET⟩\\input file&/g⟨RET⟩
```

vi is not very good with column-oriented work, but can manage, as in the following *example*, in which a file of lines of the form

```
....
% bspline- Display a B-spline.
% bsplidem- B-spline demo.
....
```

is to be adjusted, by the insertion of blanks, to have the dash appear in column 12. The following two-step process (in which there are exactly 11 dots) will accomplish this:

```
:%s/-/       -/⟨RET⟩            :%s/^\(...........\)[^-]*/\1/⟨RET⟩
```

It is also possible to **change the case** during replacement. Thus, `\l\2` (that's an 'ell', not a 'one') changes the first character in pattern part 2 to lower case, while `\L\2` changes all characters in pattern part 2 to lower case. `\u`, `\U` do the same job for upper case. For *example*, if a substandard `ftp` command has transmitted some files from a VMS system to a unix system by giving them upper-case names equipped with version numbers, you could use `ls -l` to list them all in a file, such as this one

```
-rw-r--r-- 1 deboor 2077 Mar 13 08:57 EMPTYTIT.TEX;2
-rw-r--r-- 1 deboor 408 Mar 13 08:57 MODEST.2;1
....
```

and convert the file, into the shell script (note the careful handling of the semicolon!)

```
mv EMPTYTIT.TEX\;2 emptytit.tex
mv MODEST.2\;1 modest.2
....
```

suitable for a name change, by the substitution

```
:%s/^.* \([^\;]*\)\(\;[0-9]*\)$/mv \1\\\2 \L\1/⟨RET⟩
```

As another *example*, consider a file containing names, all in capital, such as

```
..., L. SCHUMAKER, B. SMITH, ...
```

which you would like to convert to the form

```
..., L. Schumaker, B. Smith, ...
```

For this, recall that `\<` matches the beginning of a word and that `&` stands for everything matched. Thus, a suitable one-shot substitution is the following (which picks up separately the first letter and the rest of the letters in each word)

```
:%s/\<\(.\)\([A-Z]*\)/\u\1\L\2/g⟨RET⟩
```

while a more obvious two-step substitution is

```
:%s/^.*$/\L&/⟨RET⟩        :%s/\<./\u&/g⟨RET⟩
```

If you are uncertain about a substitution, try it first as a MACRO (see below). Also, I have found that, for complicated substitutions, it is best to say aloud what is intended as I type the pattern descriptions. For example, in typing the long substitution command two displays above, I would sing out the following while typing:

`:` (do)
`%` (on all lines)
`s` (substitute)
`/` (for the pattern)
`\<` (that starts at the beginning of a word)
`\(.\)` (pick up and remember the first letter)
`\([A-Z]*\)` (pick up and remember capital letters, as many as you can)
`/` (the following replacement pattern)
`\u\1` (uppercase the single letter remembered)
`\L\2` (then all lowercase for the rest of the letters remembered)
`/` (that's it)
`g` (and do this as many times as possible on the line)
⟨RET⟩ (now do it)

If you want to interrupt a substitution, type

⟨CON-c⟩

To repeat the most recent substitution command, use

```
&
```

MACROS

When you find yourself repeatedly typing in the same string of keystrokes (or if you are about to do some complicated substitutions), consider typing them once in text mode and then either picking up the entire string with the mouse and, with a click of the (correct) mouse button, executing it, or else deleting or yanking the string into some buffer $<l>$, and then executing it with the command

`@`$<l>$

(Note that **vi**'s structure (one-letter commands and buffer labels, etc.) makes it possible to concatenate commands without the use of separators.)

dealing with control characters as part of a macro

The only tricky thing about this concerns keystrokes like ⟨ESC⟩ or ⟨RET⟩ that terminate text mode or otherwise cause things to happen that you really only want to happen when the string is actually executed. For *example*, consider making up such a command string (perhaps useful when working with TEX) that will check each opening brace for its corresponding closing brace and stop when it cannot find one, as will the command string

`/{/`⟨RET⟩`%%@v`

provided it sits in buffer v, hence could call itself. To make certain that you don't start a new line while typing in this string, type, instead of ⟨RET⟩,

⟨CON-v⟩⟨CON-m⟩

Each ⟨CON-v⟩ tells the text input routine that the character following it is to be taken literally, i.e., is not to be acted upon, and ⟨CON-m⟩ is certain to be the <*newline*>-character (while ⟨RET⟩ might not be). Thus, the keystroke string actually typed in for this example, at the beginning of a clean line, say, would be

`/{/`⟨CON-v⟩⟨CON-m⟩`%%@v`

(terminating text mode with ⟨ESC⟩, of course), after which the command

`0"vd$`

deletes it into buffer v, ready for use.

This loading can be itself put into a command string, as is convenient when storing this command string together with others in a file, to be activated by loading them into various buffers. If the intent is to load the above command into the buffer v, say, then yank the line

`i/{/`⟨CON-v⟩⟨CON-m⟩`%%@v`⟨CON-v⟩⟨ESC⟩`0"vd$`

into a buffer, <*l*> say, find an empty line and execute

`@`<*l*>

The desired command string will appear, then will be whisked into the buffer v. After that, the command

`@v`

will start hunting for an unmatched opening brace (the most frequent nonobvious error in a TEX file). Make sure the nowrapscan option is set to avoid an infinite loop. (Actually, to find a nonmatching brace (or other 'fence'), it is fastest to go to the end (beginning) of the file, type a closing (opening) brace and hit `%` to look for a match.)

As another *example*, **vi** is not very good for column-oriented work, but some things can be made easier with the aid of buffers. To find the next line containing at least 73 characters, you could of course look for the pattern

```
/^.........................................................................
```

containing exactly 73 dots. That pattern is hard to type correctly. It is easier to type first

`o/`⟨ESC⟩`73i.`⟨ESC⟩`a/`⟨ESC⟩

to get those 73 dots, prefixed and postfixed with a slash, then put the whole line into the a-buffer via

`0"ad$`

then execute it via

`@a`

assigning a macro to a key

It is also possible to **assign** (or `map`) a command string **to any key** <*k*> of the keyboard, by the command

    :map <k> <string>⟨RET⟩

Here, command keystrokes to be included in <*string*> must be prefixed by ⟨CON-v⟩ to make certain that they are not acted upon rightaway. For *example*, the IBM-AT keyboard has a ⟨DEL⟩-key which is not recognized as such in **vi**. After typing in the command string

    :map ⟨CON-v⟩⟨DEL⟩ x⟨RET⟩

the ⟨DEL⟩-key will work exactly as the x-key. In fact, it is worthwhile to include such key definitions in the `.exrc` file (in which case one actually types in the line

    map ⟨CON-v⟩⟨CON-v⟩⟨CON-v⟩⟨DEL⟩ x

to insure that the map operation will map the ⟨DEL⟩-key and not the string `1x`) to make sure that they are done automatically every time you enter **vi**.

My `.exrc` file, for *example*, contains the line

    map v 081lBi⟨CON-v⟩⟨CON-m⟩⟨CON-v⟩⟨ESC⟩+f -J

which, together with the `J` command, is useful for reformating part of a text file that has acquired overly long or very short lines during the editing process.

trying out a macro

As you make up more complicated command strings to be assigned to keys, it pays to put the entire key-mapping command into a buffer <*l*> first and execute that buffer via

    @<l>

If the key works, fine. If not, type `u` to undo any possible damage done, then you merely have to *edit* the contents of the buffer (by placing its contents onto an empty line, editing that line, then yanking the result back into that buffer) and don't have to start from scratch.

recurrent use of a macro

It is even possible to get around the fact that it is illegal in **vi** for a key mapping to include the key itself, yet that is exactly what you would like to do if a complicated action is to be carried out repeatedly.

As a simple *example*, assume that you have a TEX-file containing a description of **vi**, in which you have neglected to terminate the colon commands with a final ⟨RET⟩. You know that all these commands are entered into the file in the form `{\tt:`<*details*>`}`, where, in <*details*>, pretty much anything might happen. In fact, such a statement might extend over more than one line in the TEX-file. All you want to do is to insert the string `\ret` in front of the closing brace *that matches the opening brace* to the left of that `\tt:` . The thing to do is to find the next occurrence of `{\tt:`, (or perhaps of `{\tt *:` in order to catch also the case when there is a blank between `\tt` and the colon), go to the matching closing brace by the command `%`, insert that string `\ret`, and do the whole thing over again. But that can get pretty boring, particularly if the file is long. Instead, map the whole action to some key, e.g., the key `g`, as follows:

```
:map g /{\\tt *:/⟨CON-v⟩⟨CON-m⟩%i\ret⟨CON-v⟩⟨ESC⟩v⟨RET⟩
```

thus invoking the key `v` as the final part of the action, and define the key `v` simply to give the action of `g`, i.e.,

```
:map v g⟨RET⟩
```

Now go to the top of your file, make certain that `nowrapscan` is set (by issuing the command

```
:set nowrapscan⟨RET⟩
```

if need be; it's already set if **vi** refuses this `:set` command), simply type `g` (or `v`), and watch the performance of your sorcerer's apprentice, for it is fun to watch.

If `wrapscan` is on, this will go on forever, with unhappy results. In that case, use ⟨CON-c⟩ to stop the action, and follow it immediately with the undo command `u`, set `nowrapscan`, go to the top of the file, and hit `g` again.

Here is another *example*, of use in tables with long lines. The eye usually can tell easily whether a line is first, second, in the middle, second-last, or last; all other positions are harder to track. For that reason, it is a good idea to have a blank line after every five lines in a table. Assuming that the file contains just the table and an initial blank line, one would, with the cursor on that blank line, use `g`, with the definitions

```
:map g :+5⟨CON-v⟩⟨CON-m⟩A⟨CON-v⟩⟨CON-m⟩⟨CON-v⟩⟨ESC⟩v⟨RET⟩
```

and

```
:map v g⟨RET⟩
```

Another *example* concerns overlong lines. Suppose you want to make certain that all lines in your file have at most 72 characters. Then you would like to break all lines longer than that into shorter pieces, preferably placing the break at the beginnng of a 'word'. To make your and my life easier, I assume that you have just looked for a line with at least 73 characters in it (as discussed earlier), hence, at this point, the command `n` will look for the next line with at least 73 characters in it. So, define `g` to be the following:

```
:map g n73⟨CON-v⟩⟨CON-v⟩|Bi⟨CON-v⟩⟨CON-m⟩⟨CON-v⟩⟨ESC⟩-v⟨RET⟩
```

and define its companion, `v`, by

```
:map v g⟨RET⟩
```

With `nowrapscan` set, go to the top of the file and type `g` or `v` and watch the fun. - I have noticed that, with some versions of **vi**, `n` cannot be used in this way. For those, you would have to start the definition of `g` with the explicit pattern search, i.e., replace the `n` by

```
/^........................................................................./
```

followed by ⟨CON-v⟩⟨ESC⟩. Ugh!

untypable symbols

Another *example* for the use of the keymap concerns **replacement of untypable symbols**: If the file contains a funny symbol (e.g., an unknown control or escape) that you can't seem to type and so you want to replace it throughout by <*string*>, do the following. Go to the beginning of a new line and type

```
mai:%s//<string>/g⟨CON-v⟩⟨RET⟩⟨ESC⟩
```

Then put the cursor at one of those funny symbols and type

```
xP`a0f/p
```

If the funny symbol is a control character (i.e., starts with a caret), then you would have to insert in front of it ⟨CON-v⟩⟨CON-v⟩. In any case, follow this up with

```
0"ad$@a
```

to finish the job. By the way, if you want to make visible all characters (including end-of-line, tabs, and other control-characters) in a <*range*> of lines, use

`:`<*range*> `l`

abbreviations

Another way to avoid the repeated typing of the same thing is provided by the use of **abbreviations**. The command

`:ab` <*string*1> <*string*2>⟨RET⟩

will cause any occurrence of <*string*1> bounded by blanks to be replaced by <*string*2> (within the same blanks) as soon as it appears. Note that <*string*1> mustn't contain any blanks! You might stick permanently useful abbreviations into `.exrc` (but choose <*string*1> carefully!).

use of scripts

When a substitution is to be applied to several files, and/or several similar substitutions are to be applied even to just one file, it pays to generate first a **script file** of **ex** commands, and then use the system command

`ex -` <*filename*> `<` <*scriptfile*>

If the script is to be applied to several files, simply generate a file of such command lines, then execute it (in `DOS`, it would have to be a batch file, in `UNIX`, it would have to be executed by a `csh` or `sh` or `source` command, depending on what shell you are using).

The script file should contain only **ex** commands, but without the initial colon (`:`). The final command in the script file should be `x` to save the changed file. Note that the process will break off if some command cannot be carried out. E.g., `%s/alpha/beta/` will cause a stop if no line contains the string `alpha`. Therefore, if global substitutions are to be carried out, it is better to restrict the substitution to lines that contain the pattern to be replaced, i.e., to use substitutions in the form

`g/`<*pattern*1>`/s//`<*pattern*2>`/`

which causes (the first occurrence of) <*pattern*1> to be replaced by <*pattern*2>.

Here is an example that deals with a particularly tough substitution problem, the replacement of **tabs** by the correct number of blanks (to make certain that verbatim typesetters will get all the column alignments right). **vi** is not very good at column-oriented work, and the real difficulty occurs when tabs are used for column alignment of material (rather than just proper indentation of program text). To handle this with **vi**, let's assume that tabbing gets you to the next multiple of 4, i.e., that `tabstop=4`. Then you want to replace a tab in column 1 by four blanks, a tab in column 2 by three blanks, a tab in column 3 by two blanks, a tab in column 4 by one blank, a tab in column 5 by 4 blanks, etc. In other words, the following script (extended long enough to cover all tabs in the file) will do the job:

```
g/^ /s//    /
g/^\(.\)    /s//\1   /
g/^\(..\)   /s//\1  /
g/^\(...\)  /s//\1 /
g/^\(....\) /s//\1    /
```
etc
```
x
```

In this script, the first group of blanks in each line is actually generated by a tab. To make the tabs (and some other control characters) visible, you could `:set list`, and now the script file would look like this (in **vi**), showing the tab as `^I` and the endline as `$`:

```
g/^^I/s//    /$
g/^\(.\)^I/s//\1   /$
g/^\(..\)^I/s//\1  /$
g/^\(...\)^I/s//\1 /$
g/^\(....\)^I/s//\1    /$
```
etc
```
x$
```

SHELL COMMANDS

The **ex** command `:!` makes it possible to execute system commands from inside **vi**, and this is quite useful at times.

For *example*, in a `UNIX` environment, the command

`:`<*range*>`!fmt -w 60`⟨RET⟩

will reformat each paragraph within the given range to have up to 60 characters per line (breaking a line only at the end of a word), a very handy command when editing has produced lines of very uneven lengths. Of course, all the other `fmt` options are available, too, including that of no options, in which case lines of default length are produced.

As another (`UNIX`) *example*, the command

`:.!date`⟨RET⟩

replaces the current line with a line containing the current date (down to a second).

As another *example*, if you have made some changes in a file and then decide that it would be good to keep around the original file as it was before you started working on it, then (assuming that you did no `:w` since you started on the file), the command

`:!cp` <*filename*> <*old–file*>⟨RET⟩

will preserve the file <*filename*> as it was in the file <*old–file*>, to be sure.

As another *example*, if you can't quite remember the name of the file you are about to read into the present one (and you are not working on a window system but in `unix`), then the command

`:!ls`⟨RET⟩

will display on your screen a list of all the files in the current directory.

As another *example*, the command

```
:<range>!sort⟨RET⟩
```

will, in effect, sort the lines specified by <*range*>. For *example*, if you want to retain all the items in a list that are *not* in a certain sublist, start with the file containing that sublist, one item per line, and mark the end of each line by some special symbol, e.g., by the command `:%s/$/*/`, then bring the full list (having one item per line) into the file and do the two commands

```
:%!sort            :g/\*/,.d
```

leaving you with the desired list of items not in the sublist.

You can even pipe the output from one shell command into a second command. E.g., the command

```
:<range>!sort|uniq⟨RET⟩
```

will remove all duplicate lines from that range. If, e.g., one wanted an alphabetized list of all words in a text file spelled entirely with capital letters (perhaps because these are names of variables in some oldfashioned fortran program), one could copy the entire file to a temporary file and, in that temporary file, use the commands

```
:g/^ *$/d⟨RET⟩

:%s/^\([A-Z][A-Z]*\) /⟨CON-V⟩⟨RET⟩\1⟨CON-V⟩⟨RET⟩/g⟨RET⟩

:%s/ \([A-Z][A-Z]*\)\$/ ⟨CON-V⟩⟨RET⟩\1⟨CON-V⟩⟨RET⟩/g⟨RET⟩

:%s/ \([A-Z][A-Z]*\) /⟨CON-V⟩⟨RET⟩\1⟨CON-V⟩⟨RET⟩/g⟨RET⟩

:v/^*[A-Z]* *$/d⟨RET⟩

:%!sort|uniq⟨RET⟩
```

The first command removes all blank lines. The next three isolate all of the desired words on separate lines, and the one following removes all other lines. The final one does the clean-up job: the first command sorts the lines and the second takes advantage of the sort to remove all duplicates.

Here is a more elaborate use of the `!sort` command. Suppose that you have generated a file of addresses for use with TEX. Each address uses typically several lines and has the general structure indicated by the following *example*:

```
\def\bee{H. L. Bee
167 N. Prospect Ave.
Madison WI 53705
%tel (608)-555-1212
%email {\tt hellbee@aol.com}
}
```

i.e., it starts with a line starting with `\def\`<*last − name*>, and ends with a line that contains just a right brace and nothing else. You would like to put this list in alphabetical order. Now, the command

```
:%!sort⟨RET⟩
```

will, indeed, sort all the lines in the file, but you want to sort the addresses, not the individual lines. This means that you must first convert each address into one line, sort, and then break those lines apart again. To make this possible, you would first append some unusual string, e.g., `XXX`, to each line in an address but the last one; then, for each address, join all lines in that address; then sort all lines, and, in the final step, replace all `XXX` by a <*newline*>, i.e., by `^M`.

Here is the whole sequence of commands:

```
:g/^\\def/,/^}$/-1s/$/XXX/⟨RET⟩
:g/^\\def/,/^}$/j!⟨RET⟩
:%!sort⟨RET⟩
:%s/XXX/⟨CON-v⟩⟨CON-m⟩/g⟨RET⟩
```

The exclamation point `!` after the joining command `j` ensures that no blanks are inserted at the point of the join. Also, we want to get rid of *every* `XXX` in each line, hence the final `g` in the last command.

What other text editor would have let you accomplish this so quickly?

REFERENCES

There are various simple tutorials available on the web; see, e.g.,

`http://www.cs.wisc.edu/~deboor/vi.tut`

A good summary of **vi** is contained in *Vi Command & Function Reference* by Alan P. W. Hewett. There is also a two-page list of commands for easy reference.

Online references to many other sources of information about **vi** are available at

`http://www.thomer.com/thomer/vi/vi.html`

including a link to Bill Joy's very own *Introduction to Display Editing with* **vi**:

`http://docs.FreeBSD.org/44doc/usd/12.vi/paper.html`

Another, quite independent, detailed account of **vi** is to be found in the PC/VI manual. The best documentation I have seen, though (unfortunately only now, in August 1993), is 'Learning the vi editor' by Linda Lamb, in the *NUTSHELL* series of O'Reilly & Associates, Inc. Look for any of these if you are interested in a **complete description** of **vi**.

email complaints and suggestions to `deboor@cs.wisc.edu`