Introduction to vi

Stephen Ison
si202@cam.ac.uk

5 September 2006

Contents

Introduction

This is a course on the basic use of the vi text editor. As it is taught on PWF Linux the actual editor used will be a vi clone called vim (vi Improved). It's basic command set is the same as vi, but it offers more of the modern features people have come to expect from a text editor. Note that some of the features covered in this course might not be available on a basic version of vi.

Course outline

This course is intended to give a basic grounding in the use of vi.

What is vi?

vi is a text editor, it is not a word processor [1].

Some other definitions:

- vi (pronounce: “vee eye”, not “six”, not “vye”) is an editor. An editor is a program to edit files.
- An extremely powerful Unix editor with the personality of a junk yard dog. Much-beloved by many Unix aficionados.
- A popular and common Unix text editor.
- A standard (visual) text editor (with a somewhat awkward command-line interface) for machines running Unix.
- A screen editor crufted together by Bill Joy for an early {BSD} release.

vi is actually only half the story, it's the visual mode of the vi/ex editor (ex being a line editor), for more on ex see the history section at the end of these course notes.

Why should I use vi?

One major advantage vi has over other editors is that you will find it installed with any version of Unix you use.

Because of its standardization through Unix, the vi command set is pretty much identical among all versions of vi. This means that a vi user can walk up to any Unix machine and immediately begin editing files. This is particularly important for systems administrators who often find themselves on disabled machines without most of the usual tools.

vi or vi clones are available for lots of different operating systems.

1 **word processor**

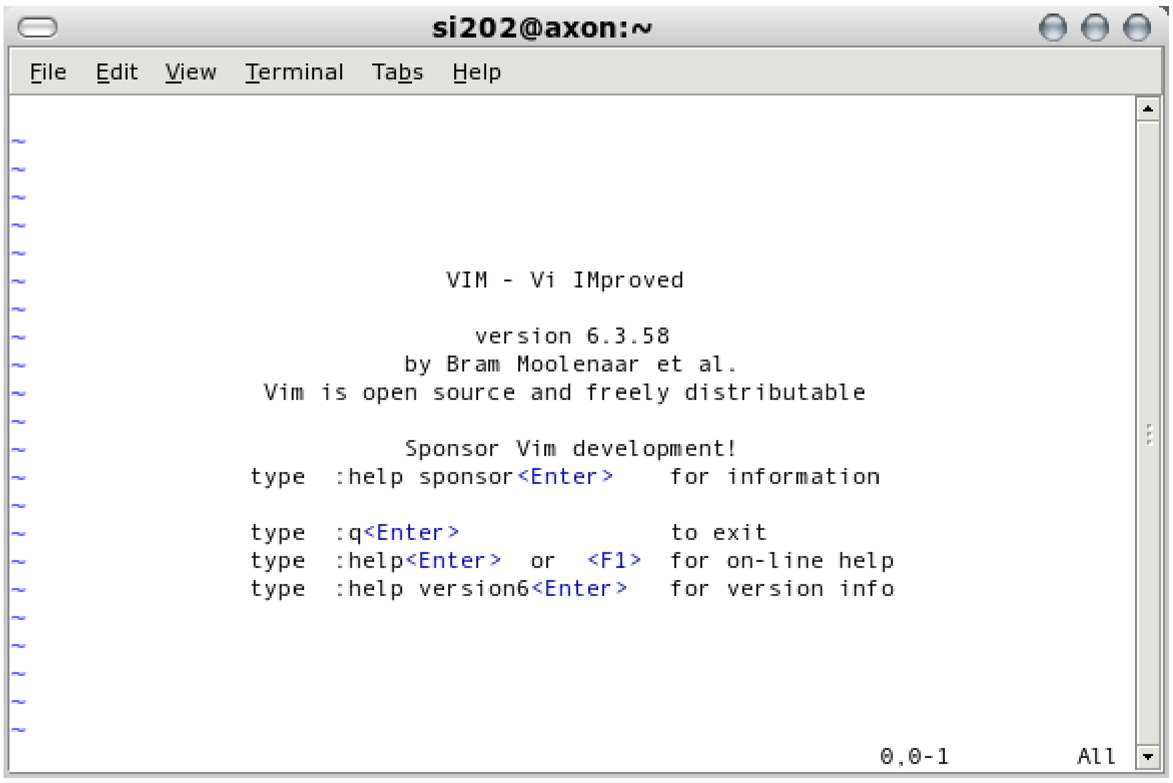
A word processor is computer software used to compose, format, edit and print documents.

Launching vi

At the shell prompt issue the command:

```
$ vi
```

This launches vi with an empty buffer (it does not load a file):



```
~
~
~
~
~
~                        VIM - Vi IMproved
~
~                         version 6.3.58
~                     by Bram Moolenaar et al.
~           Vim is open source and freely distributable
~
~                      Sponsor Vim development!
~          type  :help sponsor<Enter>    for information
~
~          type  :q<Enter>               to exit
~          type  :help<Enter>  or  <F1>  for on-line help
~          type  :help version6<Enter>   for version info
~
~
~
~
~
                                                0,0-1         All
```

Quitting vi

There are several ways of leaving vi, depending on whether you want to save changes to the original file, a different file or leave vi without saving any changes.
Here are a few options:

```
:q <return>
```

exits the vi program if no changes have been made to the file being edited.

```
:wq <return>
```

Writes the file to disk, with any changes you have made and then exits the vi program.

```
:x <return>
```

This only writes to the file if it has been modified.

```
:wq filename <return>
```

Writes to a new file called filename, without changing the original, and then exit the vi program.

```
:q! <return>
```

Discards all your changes (no files updated on disk), and then exit the vi program.

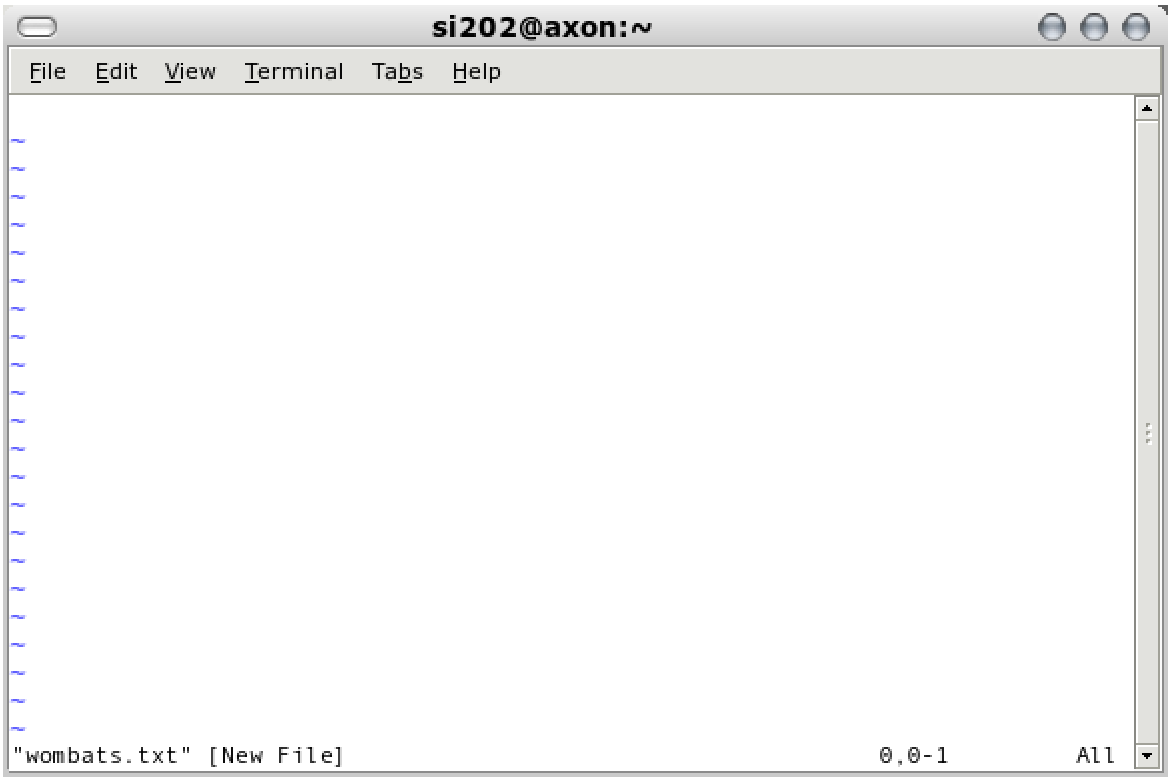
Loading a file

To start vi and load a file:

```
$ vi <filename>
```

This command launches vi with a copy of the file `<filename>` loaded if it exists, if not, the editor starts with a blank screen.

The bottom line of the editor window is the "status line", this displays information from vi, on the right two numbers separated by a comma show the current cursor position (line number followed by character number) and the portion of the file being viewed, this can be all, top, bottom or a percentage. On the left we are reminded that the file (in this case called “wombats.txt”) is a new file. The status line is also used for entering global commands or search commands. The remaining lines are your view of the file being edited. A tilde character "~" on it's own signifies that the line is empty, if a line is wider than the screen, it wraps around onto additional lines:



```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"wombats.txt" [New File]                                  0,0-1          All
```

Note that we have made no changes to `<filename>` by loading a copy of it into vi, we can still exit vi and leave `<filename>` unchanged.

Try loading `exercise_1.txt`.

Note: all vi commands are case sensitive.

vi modes

vi has two different modes. Command mode, and insert (text entry) mode:

1. **Command mode**
   Command mode is the initial state encountered when vi is invoked from the Unix shell. It is in this state that vi receives instructions on what actions are required, such as moving around the document, copy and paste operations, joining sections of text and undoing command actions.

In command mode everything that you type is interpreted as a command. Some commands are a single keystroke, while other commands require more information, such as the number of lines to apply a command to. Command mode is used to perform powerful editing functions, as well as such actions as writing the text buffer to a file. To enter command mode from insert mode, press the escape key `<ESC>`.

2. **Insert mode**
   In insert mode all characters typed are printed on the screen and become part of the current document. editing in this mode is restricted to insertion and type over. To enter insert mode from command mode, press i (for insert mode), a (for append mode), or o (for open mode).

The escape key

The escape key is your friend, if you begin to issue a command and change your mind, press `<ESC>`, if you like, press it a few times. `<ESC>` always takes you back to command mode, pressing `<ESC>` when you are already in command mode does no harm to your document, although your machine may beep at you.

Basic navigation

Moving around a file in vi does not need special cursor arrow keys, instead `h`, `j`, `k` and `l` can be used to control cursor movement. Note that these are lower-case only.

- `h` - moves the cursor one character to the left
- `j` - moves the cursor down one line
- `k` - moves the cursor up one line
- `l` - moves the cursor one character to the right

If your terminal type has been correctly specified and your keyboard has them, the cursor arrow keys should also work. However, this is not always the case, so even if you normally choose to use the arrow keys you should at least be aware of `h`, `j`, `k` and `l`. Many touch typists also prefer `h`, `j`, `k` and `l` as they don't have to move their fingers from the centre of the keyboard.

It is also possible to move around a file in bigger chunks:

By word:

- `w` - moves the cursor one forward one word
- `b` - moves it back one word
- `e` - moves to the end of the current word

By line:

- `^` - moves the cursor to the beginning (column 0) of the current line
- `$` - moves to the end of the current line
- `G` - can be used with a line number to jump to any line in the document, `1G` moves the cursor to the first line, `2G` to the second and `26G` to line 26.

By sentence:

- `(` - moves to the beginning of the previous sentence
- `)` - moves to the end of the next sentence

By paragraph:

- `{` - moves to the beginning of the previous paragraph
- `}` - moves to the end of the next paragraph

By screenful:

- `<ctrl>f` - moves forward one screenful
- `<ctrl>b` - moves back one screenful
- `<ctrl>d` - moves down half a screenful
- `<ctrl>u` - moves up half a screenful

- `H` - (high) moves the cursor to the top of the screen
- `M` - (mid) moves the cursor to the middle of the screen
- `L` - (low) move the cursor to the bottom of the screen

The whole document:

- `1G` - moves the cursor to the first line of the document
- `24G` – moves the cursor to line 24 of the document
- `G` - (with no preceding number) moves the cursor to the last line of the document.

`Do exercise 1.`

Combining commands and modifiers

vi commands operate on text objects (characters, words and lines etc), remembering how to combine a few commands and object types makes complex editing much easier. For example, commands can often be modified with repeat modifiers to make the command operate on multiple objects:

As we saw in the list of navigation commands above:

```
w
```

moves the cursor forward one word, so if we add a repeat modifier (3):

```
3w
```

moves the cursor forward three words (the command operates on the next three word objects).

```
b
```

moves back one word, so:

```
5b
```

moves back five words.

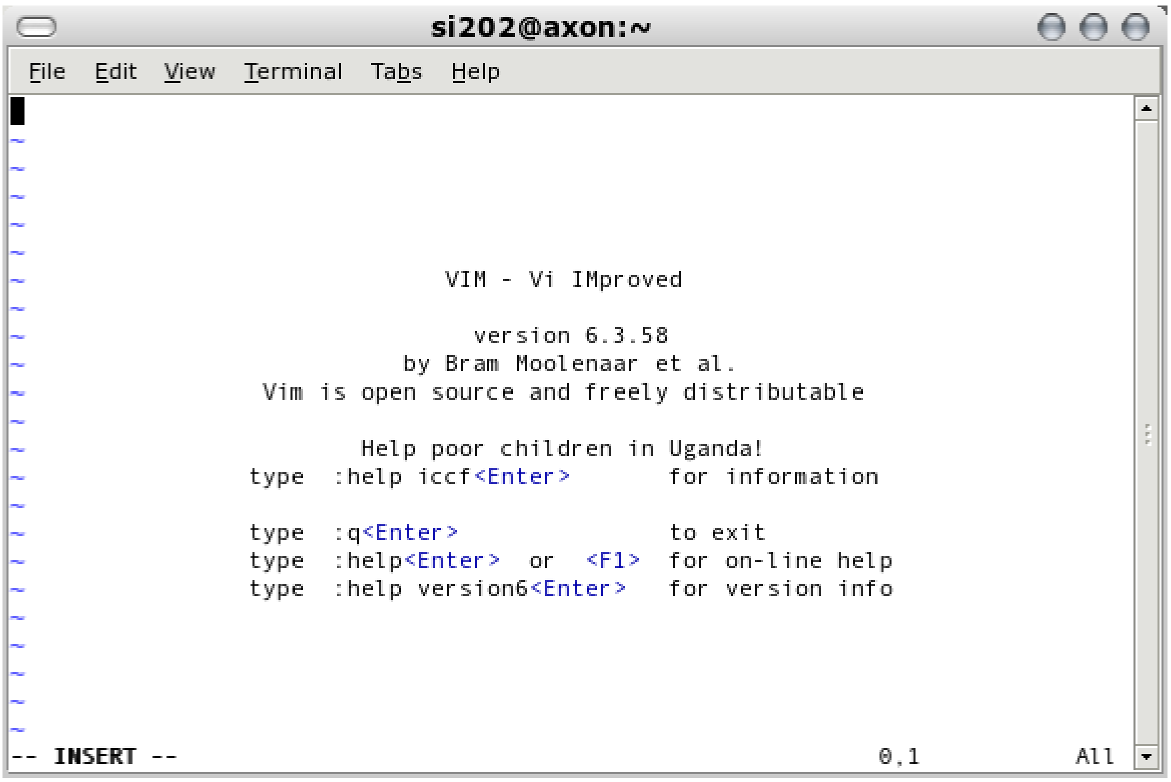
Inserting, replacing and deleting text

If you are used to a word processor there is a major difference in text input with vi. Word processors combine text editing with formatting functions, when you reach the end of a line a word processor automatically inserts a line break for you.

In vi the line wraps around with no line break. This enables you to see all of your text on the screen, but if you try to print this text, it will run off the end of the page.

Inserting text

To insert text we need to enter insert mode. This is done by pressing the i key, you will notice that `-- INSERT --` has appeared in the bottom left corner of the screen:



```
~
~
~
~
~
~                        VIM - Vi IMproved
~
~                         version 6.3.58
~                     by Bram Moolenaar et al.
~            Vim is open source and freely distributable
~
~                   Help poor children in Uganda!
~           type  :help iccf<Enter>       for information
~
~           type  :q<Enter>               to exit
~           type  :help<Enter>  or  <F1>  for on-line help
~           type  :help version6<Enter>   for version info
~
~
~
~
~
-- INSERT --                                          0,1           All
```

When you have finished inserting text and want to return to command mode press `<ESC>`.

Here are some other options for entering insert mode:

- `A` – appends text to the end of the current line
- `s` - replaces the character under the cursor with any amount of new text
- `o` - opens up a new line below the line the cursor is currently on

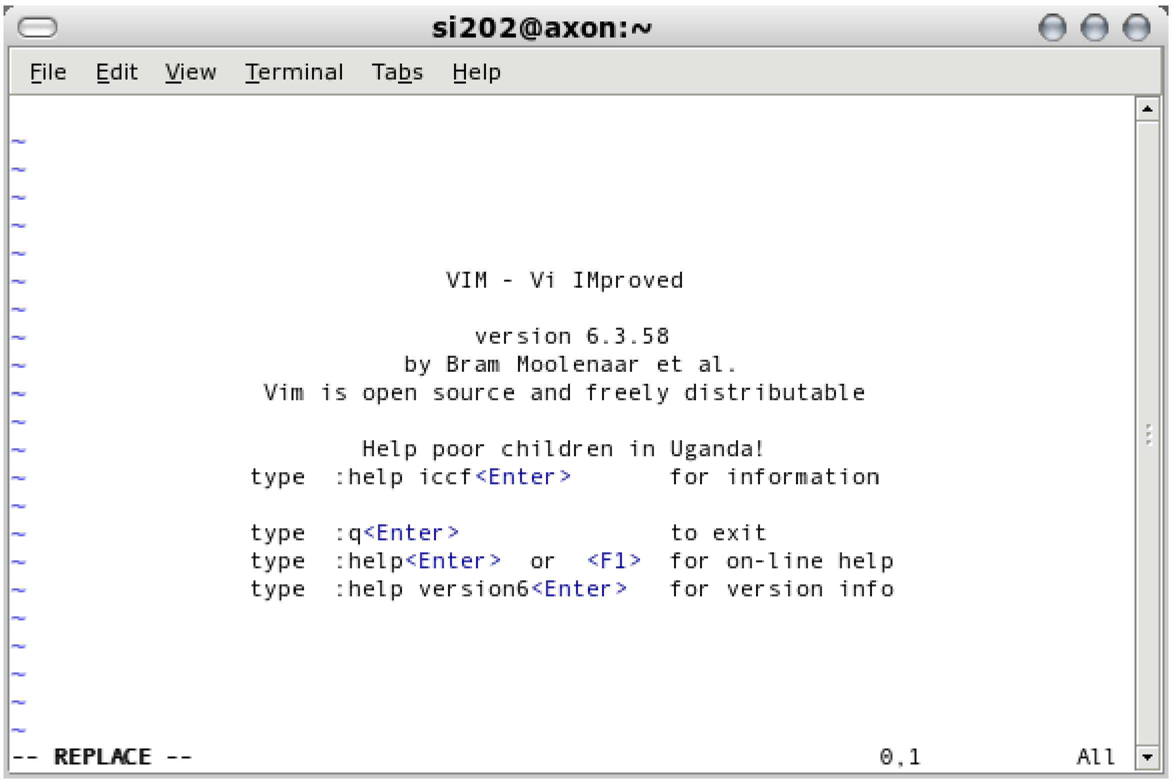
Replacing text

One character at a time

Replacing text also happens in insert mode, with:

- r – replace a single character
- R – replace multiple characters

Note that the `--INSERT--` reminder in the bottom left corner of your vi window changes to `--REPLACE--`:



```
~
~
~
~
~
~                       VIM - Vi IMproved
~
~                        version 6.3.58
~                   by Bram Moolenaar et al.
~            Vim is open source and freely distributable
~
~                   Help poor children in Uganda!
~           type  :help iccf<Enter>       for information
~
~           type  :q<Enter>               to exit
~           type  :help<Enter>  or  <F1>  for on-line help
~           type  :help version6<Enter>   for version info
~
~
~
~
~
-- REPLACE --                                          0,1           All
```

Larger chunks of text

The command:

```
cw
```

Changes a word from the current cursor position. It can also take a repeat modifier to operate on more than one word:

```
c3w
```

changes the next three words from the current cursor position.

The lower-case `“c”` operator can also be used in conjunction with modifiers to operate on larger chunks of text:

- `cw` – change one word from the current cursor position
- `c3w` – change the next three words from the current cursor position
- `c$` - change text from the current cursor position to the end of the current line (the same as C)
- `c)` – change until the end of the current sentence
- `c}` – change until the end of the current paragraph

```
Do exercise 2.
```

Deleting text

Deleting single characters

The command:

```
x
```

deletes the character currently under the cursor, it can be modified to delete multiple characters in the same way as the navigation commands we saw earlier:

```
3x
```

deletes the character currently under the cursor and the following two.

```
10x
```

deletes the character currently under the cursor and the following nine.

Deleting larger chunks

The lower-case "`d`" operator deletes text objects and can also be used in conjunction with modifiers to delete multiple characters, words, and lines. The scope must be specified after the delete operator:

- `dw` - delete one word forwards
- `db` - delete one word backwards
- `d$` - delete from the current cursor position to the end of the line (the same as D)
- `d0` - delete from the current cursor position to the beginning of the line
- `dG` - delete from the current line to the end of the file
- `d)` - delete one sentence forwards
- `d(` - delete one sentence backwards
- `dd` - deletes the current line
- `3dd` – deletes 3 lines, starting with the current line

It's worth noting that if the cursor is positioned halfway through a word `dw` will delete the remainder of the word, `d$` would delete the remainder of the word and all text to the end of the line etc.

Undo

vi has an undo function that can cause confusion. Undoing the last command action is easy:

```
u
```

However, when upper-case:

```
U
```

Undoes all changes on the current line as long as the cursor has not moved off the line since making the edits!

```
Do exercise 3.
```

Cut and Paste (yank and put)

To copy text in vi, position the cursor at the beginning of the required section of text then use the yank command, this puts the text into a temporary unnamed buffer, then reposition the cursor at the point you want the copied text placed use use the put command.

The yank command can be modified in much the same way as other vi commands:

- `yw` - yank one word forward
- `yb` - yank one word backwards
- `y$` - yank from the current cursor position to the end of the line
- `y0` - yank from the current cursor position to the beginning of the line
- `yG` - yank from the current cursor position to the end of the file
- `y)` - yank from the current cursor position to the start of the sentence
- `y(` - yank from the current cursor position to the end of the sentence
- `yy` - yank one line
- `20yy` – yank 20 lines

Relocating the Copy with the Put Command

The put command is used to place the contents of the unnamed buffer back into the file being edited. Returning whole lines into the text is handled differently to words and sentences. If the text contained in the unnamed buffer forms a line segment or is a scope which partially spans more than one line, it will be placed within the current line after the cursor if you use lower-case '`p`', or before the cursor if you use upper-case '`P`'. On the other hand, whole lines are returned to the file from the unnamed buffer without changing the current line. The lower-case '`p`' places the line or lines below the current line and the upper-case '`P`' places them above the current line.

Yank and put has the ability to insert the copy repeatedly within the same file. The format for this action is yank, relocate cursor, put, relocate cursor, put, etc. until all desired copies have been placed.

A section of text that has been deleted with `dd`, `dw` etc also ends up in an unnamed buffer and can be pasted.

```
Do exercise 4.
```

Searching

You can search your document for a pattern with the following commands:

- `/<pattern>` searches forwards
- `?<pattern>` searches backwards
- `n` repeat the last search for the next occurrence in the same direction
- `N` repeat the last search for the next occurrence in the opposite direction

You can also use regular expressions in searches. Regular expressions are extremely powerful and really beyond the scope of this course, however I think it's worth mentioning a little about the basics.

A regular expression is a description of a set of characters, the description is a mixture of text intermixed with special characters, for example, we saw earlier (in the basic navigation section) that in vi “^” and “$” represent the beginning and the end of a line, so the search:

```
/^Only this text$
```

matches lines that contain the text “Only this text” and nothing else, so:

```
/^$
```

the beginning of a line, followed immediately by the end of a line, matches empty lines.

A dot “`.`” matches any single character, and an asterisk “`*`” matches zero or more occurrences of the preceding character, so:

```
/r.t
```

matches rot and rat but not root.

```
/r..t
```

would match root.

```
/r.*t
```

would match root, but also roooooooooot.

`Do exercise 5.`

Repeating the last action

In command mode a full stop “.” repeats the last action.
For example, if you append the word “wombat” to a line, then press `<ESC>` (to leave insert mode), move to another line and press “.” the word “wombat” would be appended again.
This can be very useful for applying the same edit to multiple lines:

```
A wombat <ESC>
<return>.
<return>.
```

A wisdom of wombats! (a wisdom being the collective noun for wombats according to the San Diego Zoo).

Search and replace (substitution)

Manual substitution

Let's say we want to replace some occurrences (but not all) of “foo” with “bar”, start with a search for “foo”:

```
/foo
```

then, change word (`cw`) enter “bar”, and leave insert mode by pressing `<ESC>`:

```
cw bar <ESC>
```

Now if you press “`n`” (repeat the last search forwards) the cursor will jump to the next occurrence “foo”, if you want to replace it with “bar” then press “.” (repeat last edit), you can then repeat this throughout your file:

```
n.n.n.n.
```

and so on.

Global substitution

The manual substitution technique described above is fine for a few edits, but what about a large text file where hundreds of edits are needed?

Recall that “^” in a regular expression matches the beginning of a line and “$” matches the end?, in vi “%” matches the whole file, so to change every occurrence of the word “foo” to “bar”, in command mode:

```
:%s/foo/bar/g
```

Lets look at this command in more detail:

- `%` - The percent sign tells vi that we want this edit to apply to the whole file.
- `s` - The “`s`” makes this command a substitution.
- `/` - The first slash tells vi that what follows is the string we want to search for.
- `foo` – The search string.
- `/` - The second slash denotes the end of the search expression and the beginning of the replacement string.
- `bar` – The replacement string.
- `/` - The third slash marks the end of the replacement string.
- `g` – The “`g`” makes this a global edit (it applies to every occurrence of the string on current line, without the “`g`” the substitution would only be made for the first occurrence on the current line).

If you want the option to confirm each substitution add “`c`” to the command above:

```
:%s/foo/bar/gc
```

These commands are actually ex commands. vi is build upon another editor called ex, ex edits files one line at a time, you may also find this referred to as line mode. To avoid confusion we will not go into too much detail about ex, if you are interested see the history section at the end of these course notes.

`Do exercise 6.`

Syntax highlighting

Programmers often find it invaluable to have a text editor colourize their code according to the syntax of the language they are using, helping them to spot syntax errors. Syntax highlighting is available in vim and can be enabled with:

```
:syntax on
```

and disabled with:

```
:syntax off
```

If you don't like the colour scheme chosen by vim, try choosing another, the options available to you can be found in the <path to vim installation>/colors directory. On my machine this was /usr/share/vim/vim63/colors.

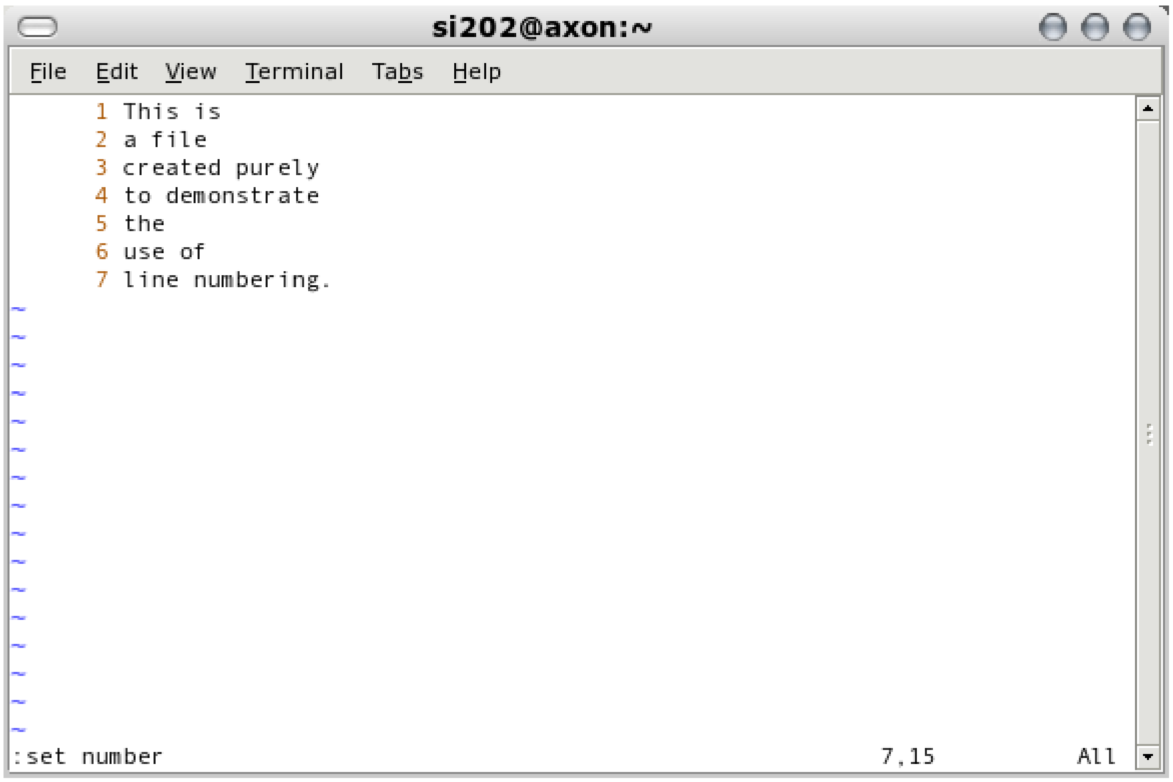
To switch to another colour scheme:

```
:colorscheme <name>
```

Working with line numbers

Line numbers are very useful when working with sections of text. In command mode line numbering can be turned on with:

```
:set number
```



```
      1 This is
      2 a file
      3 created purely
      4 to demonstrate
      5 the
      6 use of
      7 line numbering.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:set number                                          7,15          All
```

and off with:

```
:set nonumber
```

This can be particularly useful for deleting sections of text, or using substitutions that only affect part of your text. For example if you want to delete lines 2 through to 10:

```
:2,10d
```

If you want to substitute “foo” for “bar”, but only on lines 1 to 10:

```
:1,10s/foo/bar/g
```

Both of these commands need to be entered in command mode. You don't need line numbers showing for these commands to work, but it might make it easier to make sure your edit does what you think it should!

`Do exercises 7 and 8.`

Using text markers

You can mark text in vi with:

- `mx` - marks the current cursor position with x (x can be any character)
- `'x` – (apostrophe x) moves the cursor to the first character of the line marked by x
- `` `x `` – (backquote x) moves the cursor to the character marked by x

Markers can also be used for manipulating text in the same way as line numbers:

To substitute foo for bar between markers a and b:

```
:'a,'bs/foo/bar/g
```

Markers are only set during the current vi session. They are not stored in the file. To display a list of current marks use:
`:marks`

Named buffers

vi provides 26 named buffers (a-z) for storing text. You can yank text into a named buffer with:

```
“<name of buffer><command>
```

- `“ayy` – yanks the current line into named buffer a
- `“f7yy` – yanks the next seven lines into named buffer f
- `“zy)` – yanks from the current cursor position to the end of the sentence into named buffer z

You can can append text to the contents of a named buffer by using the upper case buffer name:

- `“Zyy` – appends the current line to the contents for named buffer z
- `“Ay3w` – appends the next three words to buffer a

Once you have text stored in a named buffer use can use the paste command to retrieve it:

- `“ap` – paste the contents of named buffer a after the cursor
- `“fP` – paste the contents of named buffer f before the cursor

There is no way to put part of a buffer into the text – it's the whole buffer contents, or nothing at all.

Macros

A named buffer can also act as a macro if it contains a vi command. To execute the command contained in a named buffer use `@<buffername>` For example if buffer a contains the text “10dd” the command:

```
@a
```

would delete the next 10 lines of text from the cursor position.

Mapping macros with the map command

If you find yourself using a command sequence frequently you can save having to enter the command each time by mapping the command to an unused key using the map command:

```
:map K :s/the/haddock/
```

Now if I need to replace the next occurrence of the word “the” with “haddock” I can enter:

```
K <return>
```

and vi does the substitution for me.

Before mapping commands to keys it is essential to know which keys you can use:

Unused letters:
g, K, q, V and v

Unused control characters:
^A, ^K, ^O, ^W and ^X

Unused symbols:
_, *, \ and =

Abbreviations

You can define abbreviations that vi will automatically expand for you whenever you type the abbreviation in insert mode. An abbreviation is defined with the `:ab` command:

```
:ab uoc University of Cambridge
```

Now when in insert mode if I enter “`uoc`” vi will expand this to the full text “`University of Cambridge`”.

Abbreviations are expanded when you enter a non-alphanumeric character, these include punctuation, spaces, a carridge return and `<ESC>`.

When choosing characters to represent an abbreviation be careful to choose characters that don't normally occur in the text you are typing.

To list currently defined abbreviations:

```
:ab
```

To disable an abbreviation use the `:unab` command:

```
:unab uoc
```

Abbreviations and infinite loops

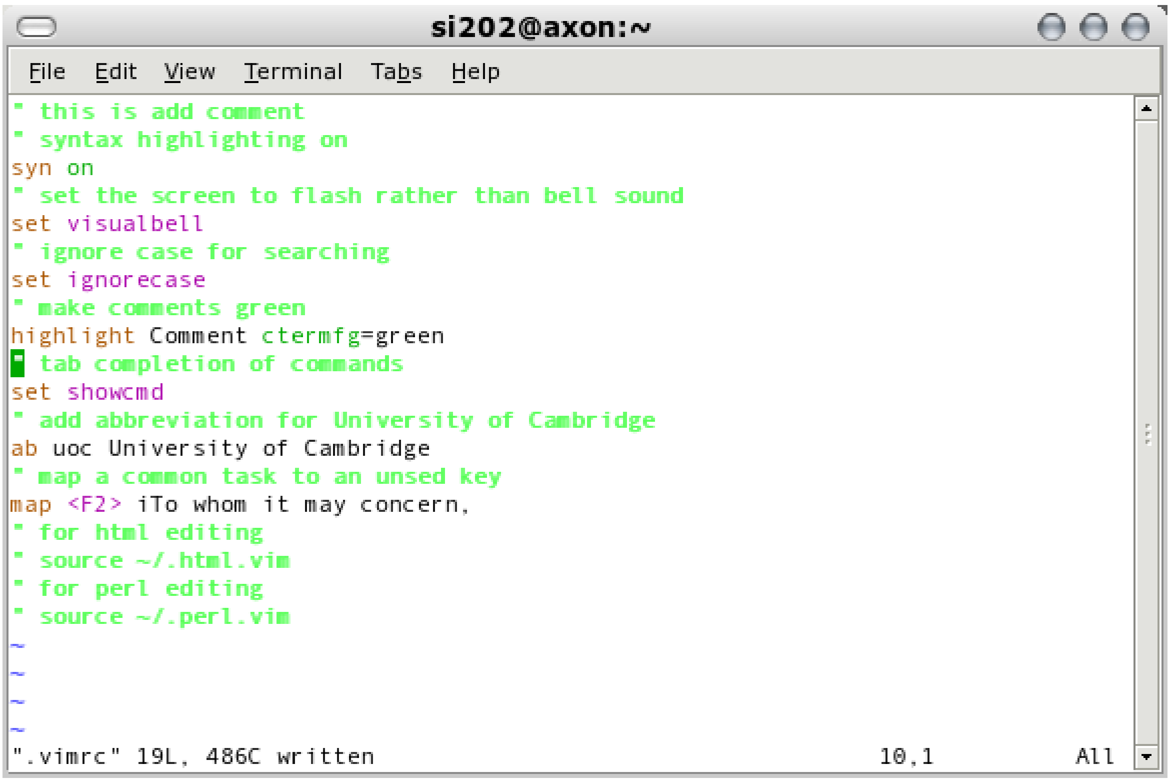
If when defining an abbreviation the characters that end the abbreviation are the same characters that I use to define it, vi may try to expand this repeatedly. For example:

```
:ab UOC University of Cambridge UOC
```

If I now enter “UOC” vi expands this to “University of Cambridge UOC”. It then sees “UOC” again and does the expansion a second time, then a third and so on. If this happens you will probably get an error, but it does depend on the version of vi you are using. I tried this trick on vim version 6.2.263 and it behaved very nicely, expanding “UOC” once only.

Using configuration files

When vi is started it looks for a configuration file in your home directory, for vim the file is .vimrc. Here is an example:



```
" this is add comment
" syntax highlighting on
syn on
" set the screen to flash rather than bell sound
set visualbell
" ignore case for searching
set ignorecase
" make comments green
highlight Comment ctermfg=green
" tab completion of commands
set showcmd
" add abbreviation for University of Cambridge
ab uoc University of Cambridge
" map a common task to an unsed key
map <F2> iTo whom it may concern,
" for html editing
" source ~/.html.vim
" for perl editing
" source ~/.perl.vim
~
~
~
~
".vimrc" 19L, 486C written                                     10,1          All
```

Lines starting with double quotes '”' are comments.

The last four lines “source ~/.html.vim” and “source ~/.perl.vim” (commented out here) read other vim configuration files which may have key mappings and abbreviations that make html and perl editing easier, this does not have to be in your .vimrc, the same command can be entered in command mode. This makes it easy to have lots of vim configuration files for specific tasks that you only source when you need to.

Useful tricks

To launch vi and begin editing <filename> at line 134:

```
vi +134 <filename>
```

To launch vi and edit multiple files:

```
vi <file1> <file2>
```

(you can skip to the next file to be edited with `:n`)

Split the current vi session in two and edit <filename> in the new session:

```
:split filename
```

(`<ctrl>ww` switches sessions)

Change case of character under cursor:

```
~
```

Join the current line and the next line:

```
J
```

Upper case the first letter of all lines:

```
:%s/^.*/\u&
```

Remove null lines in the file:

```
:g/^$/d
```

To replace all end-of-line characters with tabs:

```
:%s/\n/\t/g
```

To replace all tabs with linefeeds:

```
:%s/\t/^M/g
```

(to get ^M do: `<ctrl>v <ctrl>M`)

Reverse the order of all lines in a file:

```
:g/^.*/m0
```

Change text until:

```
ct<delimiter>
```

e.g.: position the cursor after a single quote, `ct'` will change text until the next single quote.

Change text until next occurrence of “foo”:

```
c/foo
```

List all occurrences of the word “foo” with line numbers:

```
:g/foo/#
```

Read output from a Unix shell command into current text:

```
:r !<command>
```

`Do exercises 9, 10, 11, 12 and 13.`

vim tutor mode

vim is installed on PWF Linux and has a tutor mode. This is launched by opening a terminal and typing:

```
vimtutor <return>
```

Recap and conclusion

Hopefully this course will have helped you learn the basics of vi. It does take time to get used to but can be an amazingly lightweight and efficient editor once that time has been invested.

A little history

The original text editor for Unix was ed. Written by Ken Thompson sometime around 1970 for UNICS (UNiplexed Information and Computing Service, later renamed Unix, although no-one seems to remember who by or when) on a PDP-11 owned by Bell Telephone Laboratories.

By modern standards a line editor is not a very user friendly piece of software. It doesn't display the file being edited unless you tell it to, and even then it does so one line at a time. The reason for this is that when ed was developed, a “state of the art” I/O device was a hard copy terminal. These machines printed their output on paper, so not only was refreshing the display unnecessary (as the user could look back through their edits on the paper output) but would also have been very wasteful.

In 1974 Bill Joy (author of vi) arrived at UC Berkeley, and during a visit from George Coulouris of Queen Mary College London was shown em (editor for mortals) that Coulouris had been working on.

Coulouris gave Joy a copy of the source code for em and after mixing some of the features of ed and em, Joy had an editor he called En. Eventually En became ex.

ex is the line editor which underlies vi (in fact vi is the visual mode or screen mode of ex).

ex commands are used in vi for making large scale edits. When using a line editor line numbers are used to identify the part of a file to apply a command to. So the command:

```
:1,5s/foo/bar/g
```

substitutes “foo” for “bar” on lines 1 to 5. In vi these commands are known as ex commands.

You can enter ex mode from within vi with:

```
Q
```

and

```
vi
```

from within ex mode drops you back into vi (visual mode).

You can also open a file in ex mode from a shell prompt with:

```
$ ex <filename>
```