Learning the vi editor/Print version

Aus Wikibooks
< Learning the vi editor

Contents

1 Learning the vi editor

Authors ⊞

```
This book aims to teach you how to use the vi
editor, common to many Unix and Unix-like operating systems.
~
~
~
~
~
"Learning_the_vi_editor" [New file].
```

1.1 Other sources of information:

- http://www.texteditors.org
- http://thomer.com/vi/vi.html
- Functions of VI editor (http://blog.eukhost.com/2006/10/14/fuctions-of-vi-editorlinux)

2 Getting acquainted

2.1 Introduction

2.1.1 Overview

vi is a powerful editor that is ubiquitous amongst Unix and Unix-like operating systems, but is available on many other operating systems, even on MS-DOS, Windows and the Macintosh. If not the original vi, there is usually at least a good clone available that runs on your system. Even if you use another editor you must have a passing knowledge of vi as an administrator. Sometimes vi is the only editor available when your computer crashes leaving a minimal system for you to repair.

vi, pronounced like 'vee eye', was originally written by Bill Joy for BSD Unix in Berkeley in 1976 and became quickly part of many vendor-specific versions of the (at that time) original AT&T Unix. It was later directly added to AT&T's System V Unix, too. Bill Joy later went on to co-found Sun Microsystems (http://www.sun.com) , and became the company's Chief Scientist at that time. vi stands for *visual* and was an enormous improvement of the classic Unix editor called **ed**. **ed** is a line-editor. If you are still familiar with MS-DOS, then you may know the MS-DOS **edlin** editor. ed is similar, although more powerful than edlin, which doesn't mean much.

vi also has a line-mode, called **ex**. In fact, one can argue that the program is indeed two editors in one, one editor called vi, another called ex. It is possible to switch between line and visual mode during editing. It is also possible to choose the mode during startup. However, pure usage of ex is rare. The visual mode is the prevailing mode.

Although vi stands for *visual*, classic vi is mainly operated via the character keys, and not via the mouse or the cursor keys. Once you are used to this, it becomes extremely convenient, because there is less movement of the hands to the cursor keys or mouse involved.

vi also served as a kind of incubator for Unix's terminal control capabilities. Because of vi's need to control the terminal and the many different types of terminals at that time, the *termcap* (terminal-capabilities) database was introduced (later replaced with the more flexible *terminfo* database). vi's internal high-level screen control library was later separated, and became *curses* - the Unix standard library for CRT screen handling.

2.1.2 Conventions

<c>

A single character, such as 'a' or '1'.

<ESC>, <Ctrl-[>
: Indicates that the Escape (Esc) key on your keyboard should be pressed, which is identical to Control and '['.

<CR>
: Indicates that the Return (Enter) key should be pressed.

<TAB>
: Indicates that the Tabulator key should be pressed

<Ctrl-x>, <C-x>
: Indicates that the Control key and the 'x' key should be pressed simultaneously. 'x' can be almost any other key on your keyboard.

<Shift-x>, <S-x>, <X>
: Indicates that the Shift key and the 'x' key should be pressed simultaneously

<Meta-x>, <M-x>
: Indicates that the Meta or Alt key and the 'x' key should be pressed simultaneously.

:quit, **:q**
: An Ex command. started with <:>, followed by the command and ends with <CR>. For many Ex commands there is a long form (**:quit**) and a short form (**:q**).

/*pattern*/, ?*pattern*?
: A Search pattern. Search pattern in vi are regular expressions.

:*range***s**/*search*/*replace*/*options*, **:global** /*pattern*/ **delete**
: A Search pattern combined with an Ex command.

All commands in *vi* are case sensitive.

unix-command(section)

Sometimes references to Unix commands are used in this book. On first occurrence such a name of a command is written in the typical Unix style. This style consists of the command's name followed by the section of the manual pages in which the command description can be found, in brackets. E.g. **sed**(1) refers to Unix's **sed** command which is usually documented in section 1 of the Unix manual pages (sed is the Unix stream editor; a tool for manipulating text without user interaction).

2.2 Getting vi if you don't have it already

If you're running a Unix system, or a Unix-like system (for simplicity from now on we will refer to both as a "Unix system"), such as a BSD or Linux distribution, or even Mac OS X, you're sure to have vi or one of its variants on your system.

If you're running Windows, you can get a version of vi called "vim" (http://www.vim.org) or "elvis" (http://elvis.the-little-red-haired-girl.org/whatiselvis/) . If you're on an older Mac OS (pre-OS X) system, you can get MacVim Classic here (http://macvim.swdev.org/MacClassic/) .

2.2.1 Noted vi variants

As mentioned, vi has a number of variants. They have been created because vi was only available on rather expensive Unix operating systems. Although vi itself, as well as nvi was created in Berkeley for the free BSD Unix variant, usage of BSD Unix required an original AT&T Unix license (this has later changed, see below). Original vi, for example, used code from AT&T's **ed***(1)* editor.

Over time, BSD replaced many of the original AT&T code up to the point where today there is no such code anymore in BSD, and an original Unix license is no longer needed. As part of the effort to replace all AT&T code in BSD, Keith Bostic undertook the work to create a clone of vi that was free of AT&T code, called nvi. nvi then became BSD's standard vi instead of the original vi. Another vi clone is Elvis, which was written by Steve Kirkendal.

Over time, nvi was enhanced – for example, supporting multiple windows – but originally it was not supposed to be an enhancement, 'just' a pure clone.

BSD's original vi (with the ed code inside) lives on as the vi which is distributed with System V Unix, because AT&T decided a long time ago to take it from BSD and add it to the official Unix. Of course AT&T didn't have a problem with an AT&T Unix license, so they probably never replaced the ed code inside the original vi.

Yet, some find nvi still to be too minimal, and so vim was born. **vim** (vi-i*m*proved), is another effort to extend vi's capabilities. Unlike nvi, vim goes even further to extend vi's capabilities. However some find that vim is often too much. vim comes in two variants, a text-only version, and a GUI version, the latter is called **gvim**.

Other vi clones are the already mentioned **elvis** and **stevie**. These clones were born in the CP/M and home computer area to bring the editor to these platforms, too. Of course, they were later ported to MS-DOS and Windows. These days, however, **vim** seems to be the prevailing vi-clone on free/open platforms and proprietary platforms as well.

You should choose the version you feel most comfortable with – if you have an editor you feel displeased with, it will affect your productivity.

2.3 Getting around vi

2.3.1 Starting the editor

If you are running a Unix system, you can start up vi by typing

```
vi<CR>
```

at the command line. If you are running X, with a desktop like GNOME, KDE, CDE/Motif or OpenLook you may have a launcher button handy to start the editor - if you have such a setup, you can just click the icon.

If you are running Windows or DOS with elvis, you can start up the Windows editor by double-clicking "winelvis.exe", or in DOS, you can start the editor by typing in "elvis" at the command line.

You will be greeted with a screen similar to:

```
~
~
~
~
~
~
"No File"
```

2.3.2 Quitting the editor

To quit for now, press the Escape key (the editor should beep), then enter the three characters **:q!** and press Return:

```
<ESC>:q!<CR>
```

Just before you type the final <CR> the screen will look similar to

```
~
~
~
~
~
~
:q!
```

:q! is the short form of **:quit!** which quits the editor.

You should be dropped back to your operating system (or, rather, the shell from where you started).

There are other ways to quit, e.g. pressing <Z><Z> (<Shift-z><Shift-z>) will save any unsaved work and quit the editor. Typing **:wq** will always save, even if there are no unsaved changes, and then quit the editor. **:x** will write if there are no unsaved changes, and it will quit. **:wq** and **:x** requires that you had previously provided a file name, so it will not work for the above simple example. Typing **:q** will quit if there have been no changes made; if changes have been made, vi will print a warning similar to "No write since last change".

2.3.3 Don't worry

Many first time vi users stop at this point, and never touch vi again. If you tried to enter some text after you started, you will most likely have been greeted with a series of beeps and rather erratic behavior. Don't worry. This is perfectly normal for vi, and the editor is not broken. You will soon see why this is normal vi behaviour.

2.4 Continue

Now that you know how to start the editor and quit it, let's move on to getting things done in `vi`: see Learning vi:Basic tasks

3 Basic tasks

Now that we know how to invoke the editor and quit it, we can get acquainted with how to *use* the editor.

Alternatively, you can use the ViM tutor which comes with many modern vim distributions. It contains, essentially the same information as the text below. You can invoke the tutor by entering **vimtutor** at your shell.

3.1 vi is a *modal* editor

The vi editor can do two things:

- accept a command, such as deleting a line
- accept text, written by you

In the vi editor, each of these tasks is achieved by putting the editor into a particular mode of operation (normally just called a *mode*). When you wish to give vi a command, you enter *command mode*, and when you want to enter text, you enter *insert mode*. We'll cover how to do this below.

It is important to set the correct mode before you begin writing, but this is simple to do. When you first start vi, it is automatically in command mode.

3.2 Entering text

Entering text is the most basic task an editor can do! From command mode (in which the editor starts), press `i` to enter *insert mode*, and you can begin typing. You can use the backspace key to correct mistakes you make. If you make a mistake after a few sentences, leave these errors for now, we will look at correcting them later. To leave insert mode once you're done typing, and return to command mode, press the Escape key on your keyboard (or type Control-[).

3.2.1 Exercise

Let's have an exercise:

1. Start the editor.
2. Enter insert mode.
3. Type some text
4. Return to command mode.
5. Quit the editor.

3.2.2 Solution

1. We can start the editor as in the previous section.
2. Since the editor starts in command mode, we must press the <i> key before we can start typing.

3. You can make some text up yourself!
4. Press the <Escape> key.
5. If you want to quit the editor, you need to be in command mode. Since pressing Escape puts you in command mode, you can just use the method in the previous section to exit: type **:q!**

3.3 Command mode

Command mode allows you to perform many useful tasks within vi.

3.3.1 Moving around

Say you have been writing for some time, and have forgotten something. Pressing <Backspace>, *erasing* previous work is not the best solution! We would like to move around the document freely, moving the cursor.

We can move around in the editor by first entering command mode, and then using the <h>, <j>, <k>, and <l> keys.

Note
: your arrow keys may be set up to work, and you can use them if you like, but for proficiency and for advanced work later, you should learn to use the letter keys.

- The <h> key, in command mode, moves the cursor one character **left**.
- The <j> key, in command mode, moves the cursor one character **down**.
- The <k> key, in command mode, moves the cursor one character **up**.
- The <l> key, in command mode, moves the cursor one character **right**.

If you have trouble remembering this, keep in mind that <h> is leftmost, the letter <j>goes down below the line, the letter <k>pokes up above the line, and the <l> key is rightmost. (J also resembles an arrow pointing downward, if you squint a bit.)

After you have moved the cursor using those keys, you can enter insert mode again by pressing <i>. When you do this, you insert text at the cursor, inserting text between the character to the **left** of the cursor and the **current** position of the cursor. Let's practice this in an exercise.

3.3.1.1 Exercise

You can repeat this exercise with your own sentences. Make sure you are proficient doing this before you continue.

1. Start the editor.
2. Enter the text: "The quick fox jumps over the dog"
3. Insert the word "brown" between "quick" and "fox".
4. Insert the world "lazy" between "the" and "dog".
5. Quit the editor.

3.3.1.2 Solution

1. Use the method in the previous section.
2. Press <i>, then enter `The quick fox jumps over the dog` normally.
3. Press <Escape>, then press <h> until the cursor is at the letter "f" of "fox". Press <i>, and then type "brown ".
4. Press <Escape>, then press <l> until the cursor is at the letter "d". Press <i>, and then type "lazy ".
5. Press <Escape> again, then type **:quit!**.

3.3.2 More on movement

Using `h`, `j`, `k`, and `l` is ok, but vi understands more than rows and columns. These are some commands that move by text objects:

- `w` moves forward to the beginning of the next word.
- `b` moves backwards to the beginning of the previous word.
- `(` and `)` move by sentences, either backward or forward.
- `{` and `}` move by paragraphs.

3.3.3 Deleting things

If you have made a mistake after a few lines, for instance, pressing Backspace until you have erased the mistake and starting again isn't always the best solution. We need a method of deleting mistakes that happen in the normal course of editing.

vi allows you several methods of deleting text, based on how much you want to remove. Now that you are familiar with moving around, once you've moved the cursor to where your error is:

- the `x` key deletes *one* character
- pressing `dw` deletes *one* word.
- pressing `dd` deletes *one* line

3.3.3.1 Exercise

From now on, we will omit the steps for you to start and quit the editor - you should be already familiar with those.

1. Enter the following text: `Sad I been here, I wouldnt ever ever leave.`
2. Change the word "Sad" to "Had".
3. Add an apostrophe after "wouldn".
4. Delete the extra "ever".
5. Delete the line.

3.3.3.2 Solution

1. Type the text normally. (You should already be familiar with entering insert mode and leaving it.)
2. Enter command mode, use `h` to get to the start of the line, and then press `x` to delete the S. Press `i` to insert the H, then leave insert mode by pressing Escape.
3. Now position the cursor on the `t`, and press `i` to insert the " `'` ". Leave insert mode.
4. Position the cursor over the first "e" in the word "ever" (choose whichever one you like). Type `dw` to delete the word.
5. Type `dd` to remove the entire line.

4 Making your work easier

Currently, you should by now know the rudiments of using `vi`. However, to really make `vi` work for you, it may be helpful to know the following to make your work easier for you.

4.1 More on commands

Say you are editing a document, and you wish to delete ten lines - as of now, the only way to do this is to enter `dd` ten times. Or if you want to delete seven characters exactly - you would have to enter `x` seven times. There must be a better way!

4.1.1 Repetition

Fortunately, vi lets you augment most of the commands in case you want to repeat that command a certain number of times. This is done by typing in the number of times you want that command repeated, followed by the command.

So, if you want to delete ten lines, you would type `10dd`. Or if you want to delete seven characters, you would type `7x`.

You can also repeat the last action done by typing `.` (this is a single period keystroke), the single-repeat operation over the location you want to repeat the previous operation.

So if you wanted to repeat the deletion the ten lines in the previous example, you could repeatedly press `.` to perform this operation over and over again.

4.1.1.1 Exercise

1. Type the sentence `Good morning Doctor, how are you today?`. Delete "Good morning".

2. Now using the single-repeat operation delete "how are".

4.1.2 Motion

vi allows you greater flexibility over motion as well. There are a few commands to allow you to quickly jump around your document, such as :

- `0` moves to the immediate beginning of the line
- `$` moves to the immediate end of the line
- ^ moves to the first non-whitespace character of the line

^ acts in the following way, if the line was

```
          hello how are you
```

and your cursor is on the u, if you would enter `^`, the cursor would be upon the h.

Furthermore, the `/` command allows you to jump directly to some pattern in the file. For example, if you're looking for the next occurrence of the word "pomegranate" in your text, if you hit `/`, then type in pomegranate (you need not enter insert mode) and hit enter, the cursor will jump to the next occurrence of the word, if it exists. If you want to search backwards, you would perform the same procedure, but use the `?` command. To repeat either search, enter `//`, `??`, or alternatively, type `/` or `?` and hit Enter.

4.1.2.1 Commands and motion

We know now that vi lets you enter a number to specify how many times to do something. Consider this example now: you want to delete everything after a certain point on a line - you could enter `dw` for each word from the cursor position to the end of the line, or hold down `x`, but these are cumbersome examples. vi thankfully lets you do something much faster.

With certain commands, vi allows you to specify a position, using the methods in the previous sections. The position is specified after the command. For example, to delete up to the end of the line, you would enter `d$`.

Other examples:

- `d/;` will delete until the next semicolon (This is helpful in languages like C and perl that use semicolons to finish statements).
- `d2}` to delete the next two paragraphs.
- `d4b` to delete the previous four words (alternatively, you could enter `4b4dw`).

5 Advanced tasks

5.1 Copying and Pasting

Copying and pasting tasks are done with three keys, <y> (for "yank"), <d> (for "delete"), and <p> (for "paste"). In general, you type <y> or <d> to tell vi that you're at the position where you want to start yanking or deleting some text. Then you need to tell vi where to stop, using cursor movement or other commands.

5.1.1 A Word

To delete a single word, move your cursor to the first letter, then type <d><w>. To yank a single word, move your cursor to the first letter, then type <y><w>.

5.1.1.1 Other Methods

Move to the character past the last letter and type <d> <b>.

To delete a word like "can't", which has an apostrophe, move to the first character and type <d><W>. Note the capital W. This tells vi to go all the way to the first whitespace character after the word.

Likewise, try dB.

5.1.2 A Line

To delete a single line, type <d><d>.

5.1.3 Other Amounts

One of the great things about vi is that it lets you select a bunch of text without having to move your hand to your mouse.

Type <m><a>. This will mark the current position that your cursor is at. You can go back to this position anytime you want from now on by typing <\`><a>. (\`a means "move to the character that has been marked as **a**")

Now move to some other position. Type <d><\`><a>. This will delete everything from the current position to the position you marked as **a**.

5.1.3.1 To the end or beginning of a line

<d><$> or <d><^>

5.1.3.2 To the end or beginning of the file

<d><G> or <d><1><G>

5.1.3.3 To the next occurrence of a pattern

<d>/myPattern

This is particularly useful when editing HTML files with d/<

5.2 Adjusting the Screen

vi, as a visual screen-oriented editor has a number of useful commands to redraw or adjust the screen in case you find yourself somewhere where you don't want to be.

If you run in a Unix shell, it is possible that some background process writes to the same terminal. This will disturb vi's screen layout. In order to force vi to redraw the complete screen, press <Ctrl-L> or <Ctrl-R>. Both commands do the same.

If you want to adjust what is currently displayed, then the <z> command is rather useful. It's a kind of Swiss army knife, and has a rather complex syntax:

```
[/pattern/][m]z[n](<CR>|.|-)
```

([...] denotes optional items, (...|...) denotes alternatives)

Before we explain the syntax in detail, here are some common applications of the command:

Scroll the screen so the current line becomes the middle line of the screen. The cursor remains on that line:

```
<z><.>
```

Scroll the screen so the current line becomes the top line on the screen:

```
<z><CR>
```

Scroll the screen, so the current line becomes the bottom line of the screen

```
<z><->
```

If a `/pattern/` or a number *m* is given the cursor is moved further after the adjustment. `/pattern/` indicates to move the cursor to the first match of that pattern. *m* indicates to move the cursor to the *m*th line on the screen. So, for example,

```
/while/z<CR>
```

would first scroll the screen so the current line becomes the top line on the screen, and then move the cursor to the first 'while' in the text from that position on.

The number *n* is a rather obscure parameter. If provided, it tells vi to behave as if the screen is just *n* lines high. The result is that only *n* number of lines are adjusted, and the rest of the screen is either ignored or cleared, presumably useful on slow terminals to avoid redrawing the screen unneccessarily.

6 Details

This section describes some of the details of the vi program itself (such as command line features), and other advanced vi features for aspiring vi power users.

6.1 Command line invocation

Different vi clones of course have different ways of starting the program (invocation). Usually, however, command-line versions of vi share a common basic set of command line options. These following command line options and flags are typically available. In addition, vi can be started under different names. Depending on the name used to start vi, it may either behave slightly different or load a different vi clone.

The common command line options and flags are

-

or

-s
: Suppress. All interactive user feedback is suppressed (not written to the terminal). This allows to pipe editing commands through the editor, and use it as a kind of stream editor. There are probably better streaming editor tools on Unix, like **sed**(1), **awk**(1), or **Perl**(n).

  Note, "-" is a common Unix notation to indicate standard input. It has been chosen as an alternative to `-s` by the vi authors to provide a familiar look when piping commands. It does not realy mean 'read from standard input' since vi does that anyhow.

-C
: En*c*ryption. vi prompts the user for a key (a kind of password), and uses this key to encrypt its files before writing. It also uses this key to decrypt any file opened with vi. This feature is not supported by many clones, and the encryption algorithm is a very weak one (it is based on a 256-element one-rotor algorithm). The algorithm is easy to crack. It is compatible with the Unix **crypt**(1) command. See also `-x`.

-l
: (lower-case letter L) Change some default settings so they are more useful for editing LISP source code.

-L
: (upper case letter L) Lists all files which have been saved during a crash. See `-r`, too.

-r *filename*
: Recover the file *filename* after a crash. Use `-L` to get a list of files which can be recovered.

-R
: Readonly. Files can only be viewed, not written.

-S
: Tags are not sorted. When a *tag* file is used, this flag tells vi that the tag file is not sorted, therefore vi will use a slower algorithm to look up tags. See `-t`, too.

-t *tag*
: Edit (open) that file which contains the given *tag*. This of course requires that a tag file (called *tags*) is available.

-v
: Start in visual mode. Only useful if the editor is started under the name **ex** and not **vi**.

-V
: Verbose. Commands read via standard input are echoed to standard error. This is useful for debugging when the editor is used as a streaming editor.

-w*number*
: Window size. Set the editor's number of lines to *number*. vi behaves as if the

terminal has only *number* number of lines. This was used in the old days to speed up things when connecting via a slow terminal or modem line.

-x
: Encryption. Similar to `-C`. The difference is that vi tries to guess if a file that is opened needs decryption or not. `-C` on the other hand always runs the decryption when a file is opened.

+*command*

or

-c *command*
: Execute the command *command* before allowing the user to enter own commands. The most common usage is to use this to position the editor at some specific line in a file. E.g.

```
vi +10 list.txt
```

will open the file `list.txt` and position the cursor at line 10. Another common usage is to specify a pattern:

```
vi +/END script.awk
```

This will open the file `script.awk` and position the cursor at the first occurance of the pattern 'END'.

As already mentioned, vi can be started using different names (all may not be available depending on the particular clone):

vi
: The usual way to start vi.

view
: vi starts in read-only mode.

vedit
: A few settings are changed to better suit beginners: *magic* is cleared, *showmode* and *novice* are set, and *report* is set to 1.

ex -v
: Same as just typing `vi`

6.2 Commands: Objects & Operators

6.2.1 General

Until now, this tutorial has just talked about commands, and that commands can be used in conjunction with things like word counts. E.g. `d2w` has been explained as the operator delete applied to two words. Note the `2w` part. You have learned that this part specifies to which text the operator should apply. And indeed, the `2w` part specifies to which objects of the text (words, lines, characters etc.) the operator is supposed to be applied. And you have seen that the same object specifiers can be used with all types of operators - as long as the combination makes sense.

vi commands in fact follow a general schema. Commands are made up from operators and objects:

```
[[times] operator] [[number] object]
```

This means the *operator* should be executed *times* on *number* of *object*s. Almost all parts are optional. Also, some operators don't take objects at all. This operator/operation syntax is vi's heart. It is why people either love or hate vi. People love it, because it is such a simple schema. Once one knows the few operators (not more than ten), and a few of the objects one can be very productive in vi. People who hate vi simply can't get this schema, and the fact that there is a difference between command and insert mode, into their heads.

6.2.2 Objects

We told you that things like the `w` command moves one word. We actually cheated a little bit when telling you this. There is no such thing as a `w` command. `w` is an object specification, not a command. The object specification was given without an explicit operator like `d`. In such a case vi uses the implicit default operator. And that operator is *move*.

Whenever you use an object specification without an operator, the operator *move* will be used. Therefore, object specifiers degrade to move commands. The following is a list and summary of all object specifier. Logically, you can use them in conjunction with operators, or to move around if used stand-alone. You have seen a few of them already:

6.2.2.1 Paragraph, Section, Sentence Objects

}
: Everything until next paragraph end.

{
: Everything until previous paragraph end.

]]
: [Everything until next section end.]

[[
: [Everything until previous section end.]

)
: Everything until next sentence end.

(
: Everything until previous sentence end.

6.2.2.2 Line Objects

[*number*]G
: Everything until line *number*. If *number* is ommited, last (not first) line in file. The first line can be addressed as `1G` instead.

[*number*]H
: *number* of lines after the first line currently on screen. If number is not given, the first line on the screen.

[*number*]L
: *number* of lines before the last line currently on screen. If number is not given, the last line on the screen.

M
: The middle line of the screen.

j
: One line down from current line.

k
: One line up from current line.

_
: (underscore) The current line as a whole.

6.2.2.3 Positions within Lines

0
: (Digit 0). Backward to first column of line. Same as `1|` (not `0|`).

^
: Backward to first non-whitespace character.

$
: Forward to end of line.

[*number*]|
: Column *number* of the current line. If number is not given, column 1 is used.

t*char*
: Before the next appearance of character *char* on the current line.

T*char*
: Backwards after the next appearance of character *char* on the current line.

f*char*
: Next appearance of character *char* on the current line.

F*char*
: Previous appearance of character *char* on the current line.

;
: Repetition of the last `t`, `T`, `f`, or `F` command.

,
: Repetition of the last `t`, `T`, `f`, or `F` command, but in opposite direction.

+

or

<CR>
: To the first non-whitespace character on the next line.

-
: To first non-whitespace character on the previous line.

6.2.2.4 Word Objects

w
: Forward to next begin of a word.

e
: Forward to next end of a word.

b
: Backwards to next begin of a word.

6.2.2.5 Character Object

h

or

<BS>
: Left character.

l

or

<SPACE>
: (lower-case letter L or space) Right character.

6.2.2.6 Pattern Matching Objects

/*pattern*
: Forward to the beginning of the first match of pattern *pattern*.

?*pattern*
: Backwards to the beginning of the first match of pattern *pattern*.

n
: Repeat the last / or ?.

N
: Repeat the last / or ? in opposite direction.

%

To next matching (, {, or [.

6.2.3 Operators

The previously listed objects can be used as arguments to operators. If no operator is given, the default *move* operator is used. The number of operators in vi is surprisingly small - ten in total. Here is a list of the operators:

needs better descriptions, a few of them are separately described later in this module

6.2.3.1 Operators taking Objects

c
: change - change the addressed objects. In fact, the text is replaced by what is typed in.

d
: delete - delete the addressed objects. The deleted text is placed in the undo buffer.

y
: yank - copy the text of the addressed objects into the buffer.

<
: shift left - object arguments can only be objects which address lines Indenting and Shifting.

>
: shift right - object arguments can only be objects which address lines Indenting and Shifting.

!
: *bang* filter-through - filter lines through an external program. Objects can only be objects addressing lines Filtering (stub).

6.2.3.2 Operators not taking Objects

r, s,

x
: Delete character. Use the *d* operator for deleting other objects than characters.

~
: Flip case of character at cursor position. An uppercase letter becomes its lowercase equivalent, and a lowercase letter becomes its uppercase equivalent.

6.2.3.3 Special Operator Forms

There are two special forms when typing an operator:

1. Typing the character in upper-case, instead of lower-case. E.g. `Y` instead of `y` , and
2. doubling the character. E.g. `yy` instead of `y`.

6.3 'Strange' lines on the screen

vi was written at a time when terminal or modem connections were slow. Therefor, vi used several optimisation techniques to limit the need for redrawing the whole screen. In such cases, vi used to display lines beginning with a special marker. Modern vi's seldom have the need for such performance optimizations any more, but they still have the habit to display such lines.

There are two special markers used:

```
~line
```

A leading '~' indicates that the line is past the end of the file (non-existant). This can be observed, for example, when vi is started on a new or empty file.

```
@line
```

The line is only on the screen, not in the file. This happens for deleted lines. If *wrap* is enabled (the default), this also happens for lines that are too long to show on the screen all at once.

6.4 Indenting and shifting

vi supports auto-indentation of text lines and also provides command for manual indentation. This is usefull when editing program source code. It is a common convention in many programming languages to use indentation to increase readability of the source code.

6.4.1 Options

The option `shiftwidth` (sw) determines how much space is used for indentation. E.g.

```
<ESC>:set shiftwidth 4<CR>
```

or

```
<ESC>:set sw 4<CR>
```

tells vi to use four spaces for indentation.

The option `[no]autoindent` (ai) tells vi to use auto identation or not. Auto indentation is turned on by

```
<ESC>:set autoindent<CR>
```

or

```
<ESC>:set ai<CR>
```

And it is turned off by

```
<ESC>:set noautoindent<CR>
```

or

```
<ESC>:set noai<CR>
```

6.4.2 Command Mode

Shifting lines is done with the `<` and `>` commands. `<` moves the text one `shiftwidth`to the left (outdenting), while `>` moves the text one `shiftwidth` to the right (indenting). The number of lines which can be affected are specified in vi's typical way. However, only *objects* which identify lines, and not *objects* which identify words or individual characters can be used.

E.g.

```
>G
```

moves all lines from the current line until the end of the file to the right.

Or

```
<}
```

moves all lines from the current line until the end of the paragraph to the left. Of course, the shift commands can be used in conjunction with `%`, which indicates the next opening bracket. E.g. to shift the lines encompassing the current cursor position up to the first line with a matching (, {, or [to the left one would type:

```
<%
```

Like with all commands it is also possible to specify a line count:

[*number*]<<

or

<[*number*]<
: Moves *number* of lines, starting at the current line, one `shiftwidth` to the left

(outdenting). If *number* is not given, 1 is assumed - this leads to the shifting of the current line to the left.

[*number*]>>

or

>[*number*]>
: Moves *number* of lines, starting at the current line, one `shiftwidth` to the right (indenting). If *number* is not given, 1 is assumed - this leads to the shifting of the current line to the right.

The < and > commands can also be used with a marker. In this case, the reference to the marker is placed between the two characters of the command:

<'m<
: Shifts the lines from the marker *m* up and including the current line to the left.

>'m>
: Shifts the lines from the marker *m* up and including the current line to the right.

6.4.3 Insert Mode

^t
: Moves *shiftwidth* to the right. Note, it is a common mistake to use the <TAB> key instead of `^t`. <TAB> inserts a Ctrl-I character and moves to the next multiple of *tabstop*, and not to *shiftwidth*. So <TAB> only works if *tabstop* and *shiftwidth* are set to the same value.

: Since it is not a good idea to set *tapstop* to anything else than 8, <TAB> can only be used instead of `^t` for indenting when *shiftwidth* is also set to 8.

^d
: In autoindent mode, backtabs one *shiftwidth*. E.g. if autoindent is on, and one wants to enter the follwing text:

```
if(true) {
    printf("done"); // start sw indent
    return;
} // bracket moved back to the left
```

one would type

```
if(true) {<CR>
^tprintf("done"); // start sw indent<CR>
return;<CR>
^d} // bracket moved back to the left<CR>
```

There are some special variants of `^d`, too:

^^d
: (the letter ^ followed by Ctrl-D). When this is typed first on a new line, all

autoindent is killed (the insertion point is moved to the beginning of the line). Autoindent is then continued on the next line.

E.g. to enter the following text when using autoindenting

```
        an indented paragraph
        another line in the indented paragraph
.F roff formating commands have to start at column one with a '.'
        more text in the indented paragraph
```

one would type

```
^tan indented paragraph<CR>
another line in the indented paragraph<CR>
^^d.F roff formating commands have to start at column one with a '.'<CR>
more text in the indented paragraph<CR>
```

0^d
: (the digit *0* followed by Ctrl-D). Kills all autoindent (moves cursor to the beginning of the line), and leaves autoindent off, until text is once manually indented (using `^t`).

E.g. to enter the following text when using autoindenting

```
     INTEGER FUNCTION FAC(N)
     FAC = 1
     DO 100 I = 2, N
          FAC = I * FAC
C
C PROVIDE LABEL TO END LOOP
C A HINT FOR THOSE GRASSHOPPERS: THIS IS FORTRAN CODE :-)
C
100  CONTINUE
     RETURN
     END
```

one would type

```
<ESC>:set sw=5<CR>
o^tINTEGER FUNCTION FAC(N)<CR>
FAC = 1<CR>
DO 100 I = 2, N<CR>
^tFAC = I * FAC<CR>
0^dC<CR>
C PROVIDE LABEL TO END LOOP<CR>
C A HINT FOR THOSE GRASSHOPPERS: THIS IS FORTRAN CODE :-)<CR>
C<CR>
100  CONTINUE<CR>
^tRETURN<CR>
END<CR>
```

6.5 Modelines

Modelines are a fancy but dangerous vi feature. Therefore, they are usually turned off by default in every self-respecting vi version and may have only partial support or no support at all. Some clones intentionally only support selected commands in modelines to avoid the security problems.

Modelines are lines in text files which are specially interpreted by vi when such a text file is opened. When the `modeline` (`ml`) (in some version of vi also called `modelines`) option is turned on (e.g. in the users `.exrc` file), vi scans the first and last five lines of each opened file for text of the form

```
unrelated text vi:command: more unrelated text
```

or

```
unrelated text ex:command: more unrelated text
```

Each *command* from such lines is taken and executed as it would have been typed by the user. Any text in front of the modeline-marker (`vi:` or `ex:`) or behind the closing `:` is ignored for the modeline interpretation. This can be used to place modelines in comments if they are used in some programming source code.

Here is an example Java source code file. It contains a modeline on the second and third line, in a Java comment:

```
/*
 * vi:set sw=4 ai:
 * vi:set showmatch:
 */

package gnu.freesoftware;
public class Interpreter {
    public Interpreter() ...
         ...
```

When modelines are turned on, and this file is opened, `shiftwidth` (`sw`) is set to 4, `autoindent` (`ai`) is turned on, and the `showmatch` (`sm`) option is turned on, too. There is no particular reason why two `set` commands on two modelines are used other than to demonstrate that all modeline commands found in the first and last five lines are executed, and not just the first.

Modelines can be used to play some practical jokes. E.g., a file with the modeline

```
vi:q!:
```

immediately closes the editor and makes it impossible to edit the file as long as modelines are turned on.

Modelines get outright dangerous if they mess with system files. E.g., if the super user (administrator) of a Unix system has modelines turned on, and is tricked into opening a file with the following modeline, the important Unix password file is overwritten with the contents of the opened file:

```
vi:2,$w! /etc/passwd:
root:A shiny new root password:0:0:System Administrator:/:/bin/sh
anotheruser:Another shiny new password:1:0:Just another user:/home/anoth
```

Therefore modelines should only be turned on in a controlled environment. This is sad, since in principle it is a nice idea that files are able to provide the editor with a configuration best suited to edit that file.

There are some other problems with modelines. Classic vi versions always set a file's status to *modified* if they find a modeline, even if no editing in the file has taken place. This forces the user to leave the editor with `:q!` instead of just `:q`. If instead `ZZ` is used to leave, the file is written. This causes tools like **make** to think the file has changed if it in fact hasn't.

6.6 *.exrc* Configuration File

This module is a stub. You can help Wikibooks by fixing it. For a start:

.exrc files are files containing vi (and ex) configuration data. The format of the data in such a file is that of ex commands, without the leading ':' (column). Typically, .exrc files are used to load some default mappings (map and map! ex commands) or define particular defaults. E.g. the following .exrc file would set autoindent and the shiftwidth when vi is started:

```
set ai
set sw=4
```

Normally, a .exrc file is placed in the user's home directory. Since the file name starts with a '.', the file is hidden under Unix-like operating systems. It is possible to place .exrc files in other directories, too. Vi can read the .exrc file in the current directory from which it is started. However, this feature is considered a security risk and turned off by default. It is considerd a risk, because similar jokes can be played with .exrc files as with what has been described for modelines. The .exrc file in a user's home directory is considered save, because on a correctly configured Unix system only the particular user should have write access to it.

There are three important things which should be observed when working with a classic vi and .exrc files:

1. .exrc files must not contain empty lines. Classic vi chokes on these lines with all kinds of cryptic error messages.
2. There is no official way to place a comment in .exrc files. However, since the beginning of time the following hack is used and is known to work: A line which starts with a " (quotation character) is ignored by vi.
3. Classic vi is very picky about map and map! commands. Definitions which by all means should work can trigger strange error messages. This is due to classic vi's limited parser and interpreter for such definitions. Spliting a map or map1 command in several smaller ones can sometimes help.

Many clones have relaxed these rules by allowing empty lines in an .exrc file, and by officially specifying the " as the comment character. Also, good clones should have no problem with map or map! specifications.

```
"
" This is a comment in an .exrc file
" A .exrc file must not contain empty lines, so
" comment lines need to be used to separate entries
"
set sm
set sw=8
"
set wm=8
"
" map 'g' to go to begin of file
map g 1G
" rcs check-out (/co) and check-in (/ci)
map /co :w! %.co.bak^M:!co -l %^M:e!
map /ci :w^M:!ci -u %^M:e!^M
"
" Abbreviations
ab Lx Linux
```

6.7 Tags

6.7.1 Overview

Vi can use so called *tag files* (or *tags*) to allow for quick navigation (jump) to "interesting" information in a set of files. The most common usage for this is to navigate within source code files. E.g. to jump from the usage of a certain function to the function's definition, possibly in another file.

The mechanism is relatively simple. You tell vi to go to a particular tag. vi looks up the file in which the tag can be found, opens that file and jumps to the location of the tag in that file. In order to find the file and position of a tag, vi consults a *tag file*. A *tag file* contains an index of tags. A tag is an item (e.g. some programming language object) for which such an index entry can be found in a tag file. When vi is asked to jump to a particular tag, vi looks up the index entry for that tag, and uses the information to jump to the particular item.

In order to use this feature one first has to create a tag file, or a set of tag files, containing entries for all potentially interesting items. These tag file or files then need to be made known to vi - if the default file name is not used. This can e.g. be done by having appropriate commands in an `.exrc` file.

Modern IDEs provide similar navigation features, but without the need to build a tag file separately. IDEs build the necessary index on-the-fly or use fast brute-force full-text search algorithms. The need for the extra step of creating a tag file for vi is annoying by modern standards. Still, vi's tag file system works and is usable.

6.7.2 Tag File Format, Creation & ctags(1)

The creation of a tag file typically requires to use a tool which analyses the input text files (e.g. programming source code) and generates entries for each item of interest found in the input text file. The most common tool is called **ctags**(1) and is a standard Unix program. Several vi clones come with own versions of ctags, sometimes called differently.

ctags knows the syntax of a number of programming languages and generates index information for items like function names, and macro definitions.

In case ctags is not available, or the available version of ctags does not support the programming language in use it is also possible to generate tag files with text processing tools like **awk**(1), **sed**(1) or **perl**(n) and some clever scipts, because tag files are ASCII files.

Typically an entry in a tag file looks like

```
tag-name<TAB>file-name<TAB>ex-command
```

tag-name
: The name of the item. E.g. a function name or macro name.

file-name
: The name of the file in which the *tag-name* item can be found

ex-command
: An ex editor command indicating how to locate the item in the file. This can be any ex command. But two types of ex commands make the most sense:

1. In the simple form *ex-command* is a line number, which is indeed a valid ex command.
2. Usually, however, it is a better idea to use a search pattern like `/`*tag-name*`/`. This provides some flexibility if the file is edited later. It reduces the number of times the tag file has to be re-build, because something moved inside a file. ctags also mostly generates pattern search commands and not line numbers.

Typically vi clones allow for some extensions of this format. Check the particular documentation.

A tag file should be sorted in alphabetic order to speed up operation. If this can't be done, vi's -S command line option can be used.

It is usually not a good idea to generate tag files by manually running ctags or an own tool. Instead the building of tag files is usually better integrated into the software build system. For Unix this means using *Makefiles*. Typically, the **make**(1s) targets for generating tag files are called *tags*, because that's the name of the to be created tag file:

```
# Makefile snipet
SRCS = ... # all source code
tags: $(SRCS)
        ctags -dt $(SRCS)
```

6.7.3 Ex Commands

By default, vi looks in a file called `tags` for any tags. This file name can be changed with the following ex command. In fact, more than one file name can be specified. They are all loaded to find tags. The command is maybe best placed in a project-specific `.exrc` file.

:set tags=*filename*[\ *filename ...]<CR>*
: Set name of files which contain tag information. The syntag of the command varies a little bit from vi to vi version if more than one tag *filename* is supposed to be provided. Filenames have either to be separated by "`\ `" (backslash space) or "`;`" (semicolon).

Naviation to tags can be done via the following ex command. There is also a vi command to do this.

:ta *tag-name*<CR>
: or

:tag *tag-name*<CR>
: Look up the *tag-name* in the tags file(s), open the file named in the index entry and execute the *ex-command* from the index entry. This effectively positions the user at the file and position where the symbol *tag-name* is defined. The command also remembers the current file and position on the tag stack.

6.7.4 Vi Commands

Navigation to tags can be done via the following vi command:

^]
: Take the *tag-name* at the cursor position, look it up in the tag file(s) and navigate to it, similar to the `:ta` ex command. The command also remembers the current file and position on the tag stack.

The following command uses the tag stack to go back to the previous position. Older vi's don't have it implemented:

^T
: Get the previous position and file from the tag stack and return to it. The data is removed from the file.

6.7.5 Command Line

Vi can also be started with a tag name instead of a file name. See the -t command line option.

6.8 Shell escape

While one is working in vi there might arise a need to run another operating system command. In these modern days this is not a big issue. One can open another terminal window and do as pleased. However, this is not necessary when working with vi. Vi, like many of the older interactive Unix tools, contains features to run operating system commands or start a command line interpreter (shell) from within the editor. This dates back to the times when there were no graphical user interfaces and an editor like vi would take up the complete screen of the terminal (a real terminal of course, not a terminal emulation). Being able to run commands from vi spares one the need to first quit the editor just to look something up, e.g. in a manual page.

In addition, vi provides features to insert the text output of other commands directly into the text under editing.

6.8.1 Ex Commands

The editor commands to run another command from within vi are in fact implemented as ex commands. That is, they start with the familiar ':' in command mode.

To execute one command from within vi, one would type

```
:!command<CR>
<CR>
```

At the end of the *command* one has to hit Return (the second <CR> shown above) to go back to vi. Vi then repaints the screen and continues where editing was left.

In order to repeat the last *command*, one can simply type

```
:!!<CR>
 <CR>
```

It is possible to append to a previous command by using `:!!`, follwed by whatever should be appended. For example, the second of the following two commands

```
:!ls<CR>
 <CR>
 :!! | more<CR>
 <CR>
```

is actually equal to

```
:!ls | more<CR>
 <CR>
```

(Note, **ls** is the Unix command to list a directory, **more** is the Unix command to paginate output, so it doesn't just scroll of the screen).

Once something is appended to a command, it becomes part of the last remembered command. So in the example above, another

```
:!!<CR>
 <CR>
```

would be equal to

```
:!ls | more<CR>
 <CR>
```

and not

```
:!ls<CR>
 <CR>
```

Two placeholders can be used in shell escapes to denote the current file name or the name of the previously edited file name:

%
: is a placeholder for the current file name,

#
: is a placeholder for the previously edited file name.

For example, if one is editing some shell script and wants to try it out, one could type the following commands to save the file (`:w`), set the file's attributes to executable

(`!chmod ...`), and run it (`!%`):

```
:w<CR>
 :!chmod 755 %<CR>
 <CR>
 :!%<CR>
 <CR>
```

If the file's name is, e.g. *script.sh*, the above would be equal to typing

```
:w<CR>
 :!chmod 755 script.sh<CR>
 <CR>
 :!script.sh<CR>
 <CR>
```

Instead of running a command from within vi it is also possible to start the shell from within vi. vi has an own command for this, which looks up the user's default shell (e.g. the Bourne shell or the C shell) and starts it. It is important to note that a new shell is started. The user is not returned to the shell from which vi was started. The command is called *:sh*, and it can be used as it follows:

```
<ESC>:sh<CR>
$ #shell commands, when done exit shell:
$ exit<CR>
```

6.8.2 Vi Commands

It is possible to filter all or parts of a text currently under editing through an external program. The original text is then replaced with the output of the external command.

The classic example for this feature is the usage of the Unix text formatter **fmt**. vi itself doesn't have any specific formating capabilities, however, by running the text or parts of it through an external formater from within vi, the desired formatting is easily achieved.

The vi command for filtering text is `!` (note, as opposite to the ex shell escape command, there is no leading `:`). `!` follows the usual vi command format. So one can specify the scope to which it should apply. E.g. `!!` means to filter the current line, or `!}` means to filter the current paragraph.

The `!` vi command has to be followed by by the name of the external program to be used for filtering. E.g. in order to format the current paragraph with the already mentioned Unix text formatter **fmt**, one would type

```
!}fmt<CR>
```

! can also be used to just insert the output of some external command into the currently edited text. To do so, one would first create a new empty line (e.g. with `o`), and then use `!!` to replace the empty line with the output of a command. For example,

```
o<ESC>!!ls<CR>
```

would include a listing of the files in the current directory into the text under Unix.

6.9 Execute command from Buffer

This module is a stub. You can help Wikibooks by fixing it.

@*b*
: Execute command stored in buffer *b*.

6.10 vi for Programmers

Classic vi provides a number of features which are useful for programmers. Vi was made by programmers for programmers -- but at a time when programming was different. Classic vi's programming support is, by today's standards, not too great, but of course still usable. And it is still more convenient to use vi for editing programming code than any of the *...pad* editors like **notepad** (Windows) or **dtpad** (CDE/Motif). vi probably works best on Unix systems due to the many text filters that come with Unix and the ease at which additional special-purpose filters can be scripted.

Useful features in vi for programmers are:

6.10.1 Autoindent and manual Shifting of Lines

See Indenting and shifting

6.10.2 Modelines

Modelines to set per-language defaults - if security is not an issue (see Modelines)

6.10.3 Tags for Navigating

See Tags

6.10.4 Shell Escapes

See Shell escape

- One way to use shell escapes is to run a *makefile* or the compiler from within vi. It is in general a good idea to first save the current file before trying to compile it:

```
<ESC>:w<CR>
:!make<CR>

or
<ESC>:w<CR>
:!cc %<CR>

and afterwards
<ESC>:w<CR>
:!!<CR>
```

- Another way is filter source code through an external command, e.g. through a comment-reformator for the specific language. For example, the following command will filter the current paragraph through a external comment-reformator called **recomment** (not a standard Unix program, but available as a separate script).

```
!}recomment
```

  See [1] (http://examples.oreilly.com/upt3/split/recomment) for the **recomment** script for Unix.

6.10.5 Editing multiple Files

vi can be started with a list of files:

```
vi file1 file2 file3 ...
```

Combined with other Unix features, like file matching this can be a proweful feature. E.g. to open all C source code, including header files in a directory (.h), the following command can be used:

```
vi *.[ch]
```

Or to find all files which contain a certain keyword and open them something like

```
vi `grep -l Keyword *.txt`
```

can be used.

Once vi has been started with a list of files it is possible to navigate within the list with the following commands:

:n
: next - Move to the next file in the file list

:rew

rewind - rewind the file list and open the first file in the list.

vi clones like vim typically provide more commands, e.g. to go back one file in the file list.

6.10.6 Flip between two Files

- Flip between two files using # as the name of the last file. E.g. to flip between a C source file and the corresponding header file.

```
:e x.c<CR>
:e x.h<CR>
some changes to x.h, then going back to x.c
<ESC>:w<CR>
:e#<CR>
```

- Flipping between two files using `CTRL-^`. This is one of the forgotten vi commands.

6.10.7 The error(1) Program

The **error**(1) program on Unix can be used to capture error messages from compilers and later jump from error message to error message in the editor. **error'**s way of working is archaic. It parses error messages from the compiler and inserts them as comments into the source file in which the compiler detected the error. Since **error** marks error messages with ### and `%%%` in the source file, navigation can be done with vi commands like `/` and `n`.

There are two common ways to use error with vi:

1. From outside vi and letting error start vi. error can start vi on all files which produced an error during compilation, using the `-v` flag:

```
$ cc *.c 2>&1 | error -v
# Notes:
#  cc    - the C compiler
#  *.c   - all C files in the current directory
#  2>&1  - Standard error output is redirected to normal standard output
#  |     - the output is feed to error as input
```

2. From inside vi, on the current file. First the file is saved, then it is tried to compile it, processing potential error messages with **error**, then the potentially changed source file is re-read (`:e!`), and finally the first mark is searched with `/###`:

```
first time
<ESC>:w<CR>
:!cc % 2>&1 | error<CR>
:e!<CR>
/###<CR>

and afterwards
<ESC>:w<CR>
:!!<CR>
:e!<CR>
/###<CR>
```

Note:

> **error** is a horrible kludge and can really mess up source code and a version control system! We recommend to try it at least once and form an own opinion. Also have a look at error's man page first. vi clones like vim provide a much more sensible system. Here the editor executes the compiler (or **make**(1s)) and captures the output of the compiler. The information is recorded in an error message file by vim. vim then allows to navigate the source code by extracting file names and line numbers from the error message and jump to these positions. This is the same mechanism as provided by IDEs.

6.10.8 Macros & Shortcuts

vi's provides the `:map` and `:map!` commands to define useful shortcuts and macros, and `ab` to provide abbreviations.

> *Todo: Provide a few such macros?*

6.11 nroff/troff Typesetting Support

6.11.1 Overview

vi provides support for editing text files for the Unix typesetters **nroff** and **troff**. The most "common" usage for this typesetter these days is probably to write manual (man) pages for Unix applications.

The support for writing nroff/troff input text files is always active, there is no special vi mode or option which needs to be turned on. Already from this it can be concluded that there aren't too many nroff/troff specific operations in vi. In fact, vi just provides simple ways to navigate between nroff/troff paragraphs and sections. Nevertheless, these features help when editing nroff/troff files.

nroff's/troff's text file format is simple. Normal text and macros are mixed, where macros indicate how the text should be formatted. A macro starts with a '.' in column one, followed by a one or two letter macro name followed by optional macro arguments. A typical nroff/troff text file with some macros might look like:

```
.SH "A SECTION HEADER"
Some text making up the first paragraph in the section.
More text in the paragraph.
.PP
A new paragraph has been started.
More text in this second paragraph.
.PP
Yet another paragraph.
.\"
.\" A comment infront of the next section header
.\"
.SH "SECTION HEADER 2"
Paragraph text.
```

To simplify navigation in such a text file vi knows the common macro names for sections (the `.SH` in the above example) and paragraphs (the `.PP` in the example) and provides commands to move to the next/previous section. The list of macro names is configurable in vi.

Several of the common vi features also help when editing nroff/troff text. E.g. shell escapes to run the typesetter from within vi. The following will format the current file with the manual page macros:

```
<ESC>:w<CR>
:!nroff -man % | more<CR>
```

6.11.2 Options

The following options are used to define the nroff/troff macro names as known by vi. Like all options, one changes them by using the `:set` ex command:

```
:set option[=value]
```

sections
: List of macro names which vi interprets as section delimiters. Two consequtive characters in the list form one macro name. Typically, vi's default contains `.SH`, `.NH`, `.H`, and `.HU`. So the *sections* option reads like

```
sections=SHNHH HU
```

paragraphs
: or

para
: List of macro names which vi interprets as paragraph delimiters. Two consecutive characters in the list form one macro name. vi's default typically contains `.IP`, `.LP`, `.PP`, `.QP`, `.PL`, `.Ib`, `.p`. So the *paragraphs* option reads like

```
paragraphs=IPLPPPQPPLIbp
```

6.11.3 Vi Commands

When in command mode, the following commands relate to working on nroff/troff text:

[[
: Moves to the previous section.

]]
: Moves to the next section.

{
: Moves to the previous paragraph.

}
: Moves to the next paragraph.

And, for completeness:

(
: Moves to the previous sentence.

)
: Moves to the next sentence.

7 Vi clones

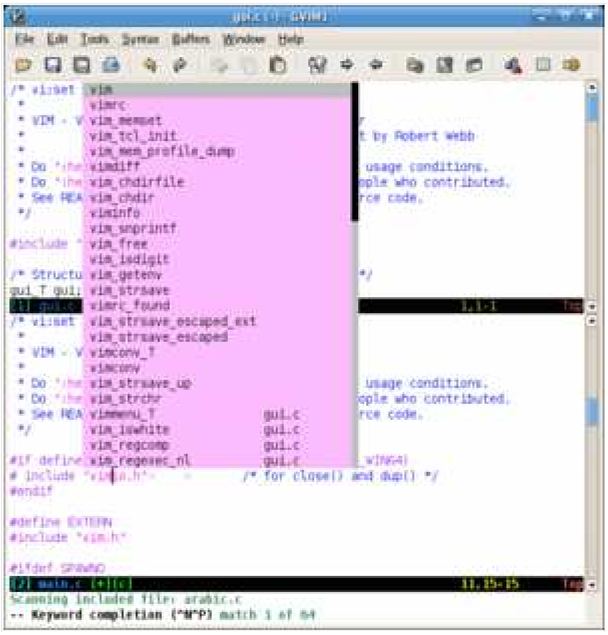
The following editors are derived from or share the spirit of the original vi editor, coupled to an easier to learn user interface:

- Vim (Feb 7, 2005)
    - Basic navigation (Nov 1, 2006)
    - Modes (Nov 1, 2006)
    - Tips and Tricks (Nov 1, 2006)
    - Useful things for programmers to know (Nov 1, 2006)
    - Enhancing Vim (Nov 1, 2006)
    - Exim Script language (Nov 1, 2006)
- vile (May 28, 2006, just a stub)
- BusyBox vi (May 28, 2006, just a stub)

8 Vim

8.1 Overview

- Basic navigation ⊞ (Nov 1, 2006)
- Modes ⊞ (Nov 1, 2006)
- Tips and Tricks ⊞ (Nov 1, 2006)
- Useful things for programmers to know ⊞ (Nov 1, 2006)
- Enhancing Vim ⊞ (Nov 1, 2006)
- Exim Script language ⊞ (Nov 1, 2006)



```
/* vi:set  vim
 *         vimrc
 * VIM - V vim_memset                     /
 *         vim_tcl_init                   t by Robert Webb
 *         vim_mem_profile_dump
 * Do ":he vimdiff                         usage conditions.
 * Do ":he vim_chdirfile                  ople who contributed.
 * See REA vim_chdir                      rce code.
 */        viminfo
           vim_snprintf
#include " vim_free
           vim_isdigit
/* Structu vim_getenv                     */
gui_T gui; vim_strsave
[1] gui.c  vimrc_found                                 1,1-1        Top
/* vi:set  vim_strsave_escaped_ext
 *         vim_strsave_escaped
 * VIM - V vimconv_T
 *         vimconv
 * Do ":he vim_strsave_up                  usage conditions.
 * Do ":he vim_strchr                     ople who contributed.
 * See REA vimmenu_T            gui.c     rce code.
 */        vim_iswhite          gui.c
           vim_regcomp          gui.c
#if define vim_regexec_nl       gui.c     _WIN64)
# include "vim.h"              /* for close() and dup() */
#endif

#define EXTERN
#include "vim.h"

#ifdef SPAWNO
[2] main.c [+][c]                                      11,15-15     Top
Scanning included file: arabic.c
-- Keyword completion (^N^P) match 1 of 64
```

Graphical Vim under GTK2

8.2 External links

- vim.org (http://www.vim.org) - documentation and many tips and plugins
- vi-improved.org (http://www.vi-improved.org) - Wiki dedicated to vim
- Public .vimrc files (http://students.iiit.ac.in/~deepakr/config/.vimrc)

8.3 Basic navigation

Basic navigation in vim is covered below.

8.3.1 Moving around

We can move around in the editor by first entering command mode, and then using the <h>, <j>, <k>, and <l> keys.

Note
: your arrow keys may be set up to work, and you can use them if you like, but for proficiency and for advanced work later, you should learn to use the letter keys.

- The <h> key, in command mode, moves the cursor one character **left**.
- The <j> key, in command mode, moves the cursor one character **down**.
- The <k> key, in command mode, moves the cursor one character **up**.
- The <l> key, in command mode, moves the cursor one character **right**.

If you have trouble remembering this, keep in mind that <h> is leftmost, the letter <j>goes down below the line, the letter <k>pokes up above the line, and the <l> key is rightmost. (J also resembles an arrow pointing downward, if you squint a bit.)

After you have moved the cursor using those keys, you can enter insert mode again by pressing <i>. When you do this, you insert text at the cursor, inserting text between the character to the **left** of the cursor and the **current** position of the cursor. Let's practice this in an exercise.

8.3.2 VIM Help system

vim is a very feature rich application. Unlike the 'vi' editor it includes a help system. Because the help system will allow you to teach yourself much more than any book on vim possibly could, you will benefit from the power of the vim editor much more if you learn to use it. On a normal vim installation you should be able to start the online help by pressing the <HELP> key . If your keyboard does not feature a <HELP> key then you can try <F1> instead. (Some system administrators may have changed how vim behaves. If you cannot get into vim's help system with these commands, perhaps your administrator can help.)

```
:help
```

Start vim and enter command mode by pressing escape. To get help on any command simply type `:help` *command*.

For example, if you would like to learn all the different ways the **:x** command can be used you could type `:h x`. To move around in the help files the same keys work, <h>, <j>, <k>, <l>. To leave the help files type **:quit**. If you know you want to do something, but you aren't sure what the command might be you can type partial commands like this `:help cut`. To learn to switch text from upper case to lower case you could type **:help lowercase**

When you search for help on any subject, vim will (normally by default) create a window (buffer) which you can navigate just like any window in vim. You can close the help window by typing **:quit** or **:q** and pressing enter.

The default help file (shown when you type "**help**) explains basic navigation for vim and for vim's help files.

8.4 Modes

VIM offers more modes than vi (which offers only the "normal", "insert" and "command–line" modes). Theses additional modes make VIM more powerful and easier to use; because of this, vim users should at least be aware that they exist. (NOTE: *If you ever enter a mode you are unfamiliar with, you can usually press* ***ESC*** *to get back to* normal *mode.*)

Here a short overview of each mode available in vim:

Name	**Description**	**help page**
insert	For inserting new text. The main difference from vi is that many important "normal" commands are also available in insert mode - provided you have a keyboard with enough meta keys (such as Ctrl, Alt, Windows-key, etc.).	`:help Insert-mode`

normal	For navigation and manipulation of text.	`:help Normal-mode`
visual	For navigation and manipulation of text selections, this mode allows you to perform most normal commands, and a few extra commands, on selected text.	`:help visual-mode`
select	Similar to visual but with a more MS-Window like behavior.	`:help select-mode`
command-line	For entering editor commands - like the help command in the 3rd column.	`:help Command-line-mode`
Ex-mode	Similar to the command-line mode but optimized for batch processing.	`:help Ex-mode`

Each mode is described below.

8.4.1 insert (and replace)

In insert mode you can type new text. In classic vi the insert mode was just that: insert text and nothing else. Vim makes use of many meta keys on modern keyboards; with a correctly configured vim, cursor keys should work in insert mode.

Insert mode can be reached in several ways, but some of the most common ones are <a> (append after cursor), <i> (insert before cursor), <A> (append at end of line), <I> (insert at begining of line), <C> (change to end of line), and <s> (substitute characters).

If sometimes whish for the "window way of live" of selecting some text and then replace it with new text then <C> is your friend. The visualy selected text is then deleted and you enter insert mode.

8.4.2 normal (command)

Unless you use the evim interface this is the standard mode for vim (vim starts in normal mode). Everything the user types in normal mode is interpreted as commands (including those which switch the user to other modes).

If vim is started as evim (*evim* on the command line), vim keeps the user in insert mode all the time. Normal mode can be reached for individual commands by pressing <Ctrl-O> followed by the desired command. After one command, the user is returned to insert mode. (Each normal command must be started first by pressing <Ctrl-O>).

8.4.3 visual

There are three different types of highlighting in visual mode. Each allows the user to highlight text in different ways. Commands that normally only affect one character, line, or area will affect the highlighted text (such as changing text to uppercase (<Ctrl-~>), deleting text (<d>), indenting lines (>>, <<, and =), and so forth).

There are three (sub)types of the visual modes which are *visual*, *block-visual* , and *linewise-visual*

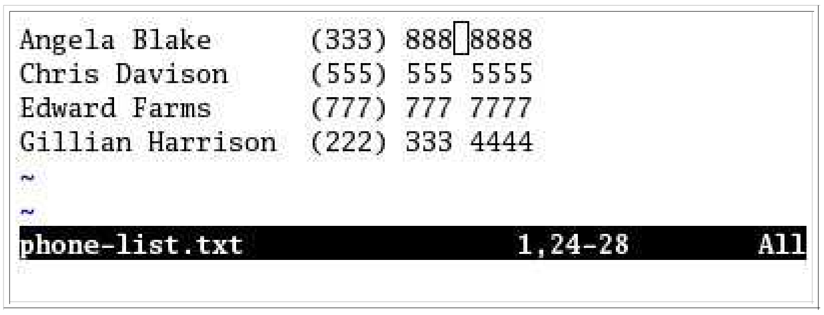
8.4.3.1 plain visual mode

The plain *visual* mode is started by pressing 'v' in normal mode. At any point, pressing ESC or <v> will leave VISUAL mode without performing an operation. Movement commands change the selection area, while other commands will generally perform the expected operation on the text (there are some exceptions where the behavior will change or where the command won't work, but if it doesn't do what you hoped you can always undo with <u>).

8.4.3.2 block visual mode

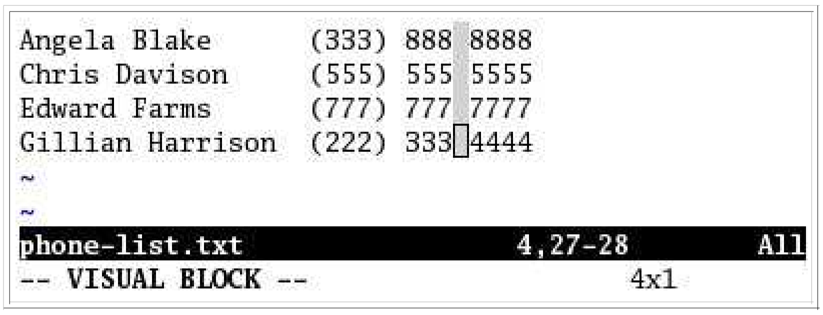
block-visual is started by pressing <Ctrl-V> (or <Ctrl-Q> in some windows versions. If neither of these works use ":help visual-block" to find out how). Visual blocks always maintain a rectangular selection, highlighting only specific columns of characters over multiple lines. In this following example the user wants to put a dash in each phone number between the second and third number fields:

The user first moves the cursor to the top of the column (you could start at the bottom if you want).



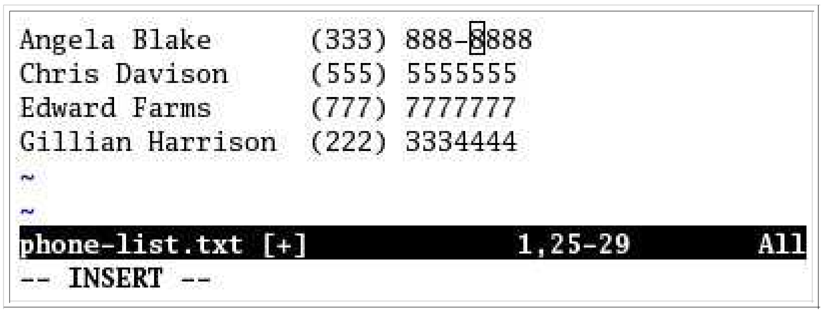
```
Angela Blake      (333) 888 8888
Chris Davison     (555) 555 5555
Edward Farms      (777) 777 7777
Gillian Harrison  (222) 333 4444
~
~
phone-list.txt                 1,24-28        All
```

Next, press <Ctrl-V>. This puts you in block-visual mode (VISUAL BLOCK appears at the bottom to tell you what visual mode you're in). Next, move down to the bottom desired line. You can see a single column highlighted in this example, but you could move right or left and highlight more columns.



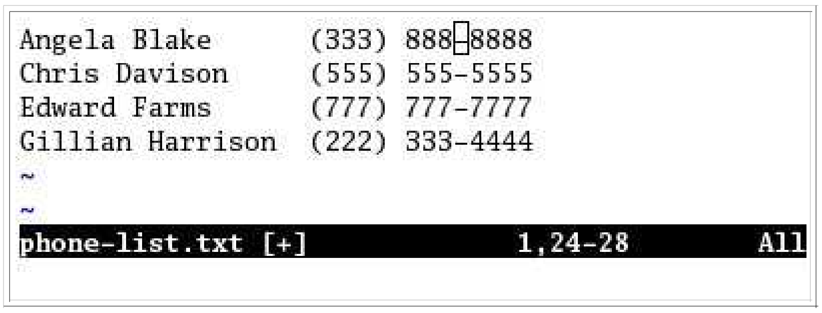
```
Angela Blake      (333) 888 8888
Chris Davison     (555) 555 5555
Edward Farms      (777) 777 7777
Gillian Harrison  (222) 333 4444
~
~
phone-list.txt                 4,27-28        All
-- VISUAL BLOCK --                    4x1
```

In this case, the user wants to *change* the spaces to dashes. To change text, we press 'c'. The spaces all disappear, and the changes are shown only in the current line while we type:



```
Angela Blake      (333) 888-8888
Chris Davison     (555) 5555555
Edward Farms      (777) 7777777
Gillian Harrison  (222) 3334444
~
~
phone-list.txt [+]          1,25-29       All
-- INSERT --
```

when we press <ESC>, though, the change is duplicated on all the lines.



```
Angela Blake      (333) 888-8888
Chris Davison     (555) 555-5555
Edward Farms      (777) 777-7777
Gillian Harrison  (222) 333-4444
~
~
phone-list.txt [+]          1,24-28       All
```

(Note: if you simply want to *insert* text rather than change it, you will need to use '<I>' or '<A>' rather than '<i>' or '<a>'.)

8.4.3.3 linewise visual mode

In **linewise-visual** mode, enterd by <Shift-V>, entire lines are highlighted. Otherwise, it generally works like the plain **visual** mode.

8.4.4 select

like the visual mode but with more CUA like behavior. This means that if you type a single character it replaces the selection. Of course you lose all the one key operation on selection like <U> to make a selection uppercase.

This mode is usualy activated by:

```
:behave mswin
```

which is default for MS-Windows installations. You can get the normal mode with

```
:behave xterm
```

8.4.5 command-line

Within the command-line you can run Ex commands, enter search patterns, and enter filter commands. At the bottom a command line appears where you can enter the command. Unlike vi - vim supports cursor keys which makes entering commands a lot easier. After one command the editor returns into normal mode.

You can enter an Ex command by typing a : in normal mode. Some examples include:

```
:set number
:substitute/search/replace/ig
```

You can enter a search pattern by typing `/` to search forward, or `?` to search backward. You can use vim's expanded regular expressions in these search patterns. For example,

```
/word
```

will jump to the next occurence of "word" (even if it is "sword" or "wordlessly"), but

```
/\<word\>
```

will jump only to a complete word "word" (not "sword" or "wordless").

You can enter a filter by typing `!` followed by a motion command, then a shell command to run on the text captured by the motion. For example, typing

```
!22jsort
```

in linux will sort the current and 22 following lines with the *sort* system command. The same thing can be done with

```
:.,.+22!sort
```

As a matter of fact, vim creates the above command for you if you follow the first example!

8.4.6 Ex-mode

The Ex mode is similar to the command line mode as it also allows you to enter Ex commands. Unlike the command-line mode you won't return to normal mode automatically. You can enter an Ex command by typing a `Q` in normal mode and leave it again with the `:visual` command. Note that the Ex mode is designed for Batch processing and as such won't support mappings or command-line editing.

For batch processing the Ex-mode is normally started from outside by calling the editor with the "-E" option. Here are real live example form a RPM Package Manager specification:

```
vim -E -s Makefile <<-EOF
   :%substitute/CFLAGS = -g$/CFLAGS =-fPIC -DPIC -g/
   :%substitute/CFLAGS =$/CFLAGS =-fPIC -DPIC/
   :%substitute/ADAFLAGS =$/ADAFLAGS =-fPIC -DPIC/
   :update
   :quit
EOF
```

The RPM uses Bash as script language which make the example a little difficult to understand as two different script languages are mixed in one file.

`vim -E -s`
: starts vim in "Ex-Improved" mode which allows for more advanced commands then the vi compatible Ex-mode (which is started with `vim -e -s`).

`<<-EOF`
: tells bash to copy all lines that follow into the standard input of the external program just started.

`:`
: are lines with Ex commands which vim will execute. The `:` is optional but helpful when two script languages are mixed in one file

`:update`
: A beginners mistake is to forget to actually save the file after the change - falsely assuming that this happens automatically.

`:quit`
: Last not least: don't forget to actually exit vim again.

`EOF`
: marks the end of the standard input redirection - from now on bash will execute the command itself again.

If your shell does not allow such nifty redirection of standart input then you can always use a more classic approach to I/O redirection using two files:

```
vim -E -s Makefile <Makefile-Fix1.vim
```

And if have no standard input redirection available then you can try the `-c` option in combination with the `source` command:

```
vim -E -s -c "source Makefile-Fix1.vim" Makefile
```

With the Exim-mode many task classically performed by awk or sed can be done with vim and often better so:

- awk and sed are stream oriented - they only read the file forward from be beginning to the end while vim is buffer oriented - you can move forward and backward in the file as you like.
- vim's regular expressions are more powerful then awk's and sed's expressions - for example vim can match over several lines and supports zero matches.

The Vim Tipbook is a collection of tips, hints and HowTos for using the Vim text editor (http://www.vim.org) . It is an outgrowth of the Vim tips database (http://www.vim.org/tips/index.php) in a more flexible format, and also includes some helpful posts from the Vim mailing lists (http://vim.sourceforge.net/maillist.php) .

For information on the general use of Vim, please see the Learning the vi editor/Vim Wikibook.

8.5 About this Book

8.5.1 Tips for Editing

- Where possible, extensive personal configurations should be avoided. Keep suggestions within the scope of a single tip. You may wish to link to several other tips that might be used alongside, much in the same way a food cookbook would suggest dishes that go well together.
- Always provide enough information so that a tip can be used from Vim's default `compatiable` settings.

8.5.2 Conventions Used

- <key> represents a single press of keyboard *key*
- `'nocompatible'` (text in a mono-faced font) represents a setting, variable, or command.
- Where a series of commands are required to be entered, these might be listed in a pre-formatted block:

```
{{Vi/Ex|set} number
{{Vi/Ex|set} wrapwidth=70
{{Vi/Ex|set} wrap
```

8.6 Vim Help

If you are new to vi, try the `vimtutor` command. It's an excellent guide for the beginner.

Vim has an extensive help system. EVERYTHING is covered. This system is so extensive, however, that finding the needed information is sometimes akin to finding one's own little needle in a huge stack of hay. But even for that, there are Vim tools:

- (assuming `'nocompatible'` is already set)

```
set wildmenu
```

- Help tag completion: if you think 'foo' is part of something which has a hyperlink in the help system, use

```
help foo<Tab>
```

where <Tab> means "hit the Tab key", and if there is only one possible completion Vim fills it in for you; if there is more than one the bottom status line is replaced by a menu which can be navigated by hitting the <Left> and <Right> arrow keys; accept a selection by hitting <Enter>, abort by hitting <Esc>.

- The `:helpgrep` function: if you think that some regular expression describe text you want to search for in the text of all the help files, use

```
helpgrep <pattern>
```

where <pattern> is a Vim regular expression, like what you can use after / or ? . It may take some time for Vim to look up all its help files, and it may or may not display interim information which may require you to hit Enter to clear the `|more-prompt|` (q.v.) When the blinking cursor reappears in your editfile, it means Vim has compiled the list of all help locations where your regexp matches. See them by means of the following commands:

```
cfirst or :cr
cnext or :cn
cprevious or :cprev or :cN
clast or :cla
```

8.7 Inserting Text From a File or Register

If your text is in a file on its own, you can use `:r` with a line number (the number of the line after which to insert, or `0` for "before first line", or `.` for "after cursor line", or `$` for "after last line"; default is after cursor line) in the "range" position, i.e. just before the `r`. The file name comes as an argument at the end.

Example (after line 5):

```
 5r ~/template.txt
```

If your text is in a register, you can use `:put` with a line number (again) in the range position and the register name (including `"`, which must be escaped as `\"`, for the default register; or + for the system clipboard) after the `:put`.

Example (before cursor line):

```
 .-1put \"
```

You can also insert a string directly using `:put` and direct assignment:

```
 :put ='This is text to insert.'
```

See

```
:help :read
:help :put
```

8.8 Full Screen Mode

To achieve a full screen editing window on any version of gvim you can do:

```
:set go-=m go-=T go-=l go-=L go-=r go-=R go-=b go-=F
:set lines=999 columns=999
```

- `'guioptions'`: We remove the flags one-by-one to avoid problems if they appear in the option in a different order, or if some of them do not appear at all. By choosing which ones to remove (or not) you can customize your own flavour of "full-screen Vim".

```
m   when present, menu bar is present
T   present, toolbar is present on versions which support it (W32, GTK
l   when present, left scrollbar is always present
L   when present, left scrollbar is present if there is a vertical spl
r   when present, right scrollbar is always present
R   when present, right scrollbar is present if there is a vertical sp
b   when present, bottom scrollbar is present
F   when present, gvim (Motif) will display a footer
```

- `'lines'`, `'columns'`: setting them to a large value will maximize the window.

For more, see:

```
:help 'guioptions'
:help 'lines'
:help 'columns'
```

8.9 Useful things for programmers to know

There are quite a few things programmers ought to know about vi that will make their experience that much easier. Programmers can save hours and weeks of man-hours over the long haul with effective editors. Here are some tricks and tools that vim provides. With the time you save, you might speed up your work and have some extra time for a quick Quake deathmatch or eventually increase your productivity to help justify a larger wage increase.

8.9.1 Word, variable, function, and line completion

Sometimes the word you're typing is really long. You shouldn't have to type it all out. If it's in your dictionary, or in the current file, you can save a lot of time with <Ctrl-P> and <Ctrl-N>. Let's take a closer look at how this works:

- *Word/variable/function name Completion*

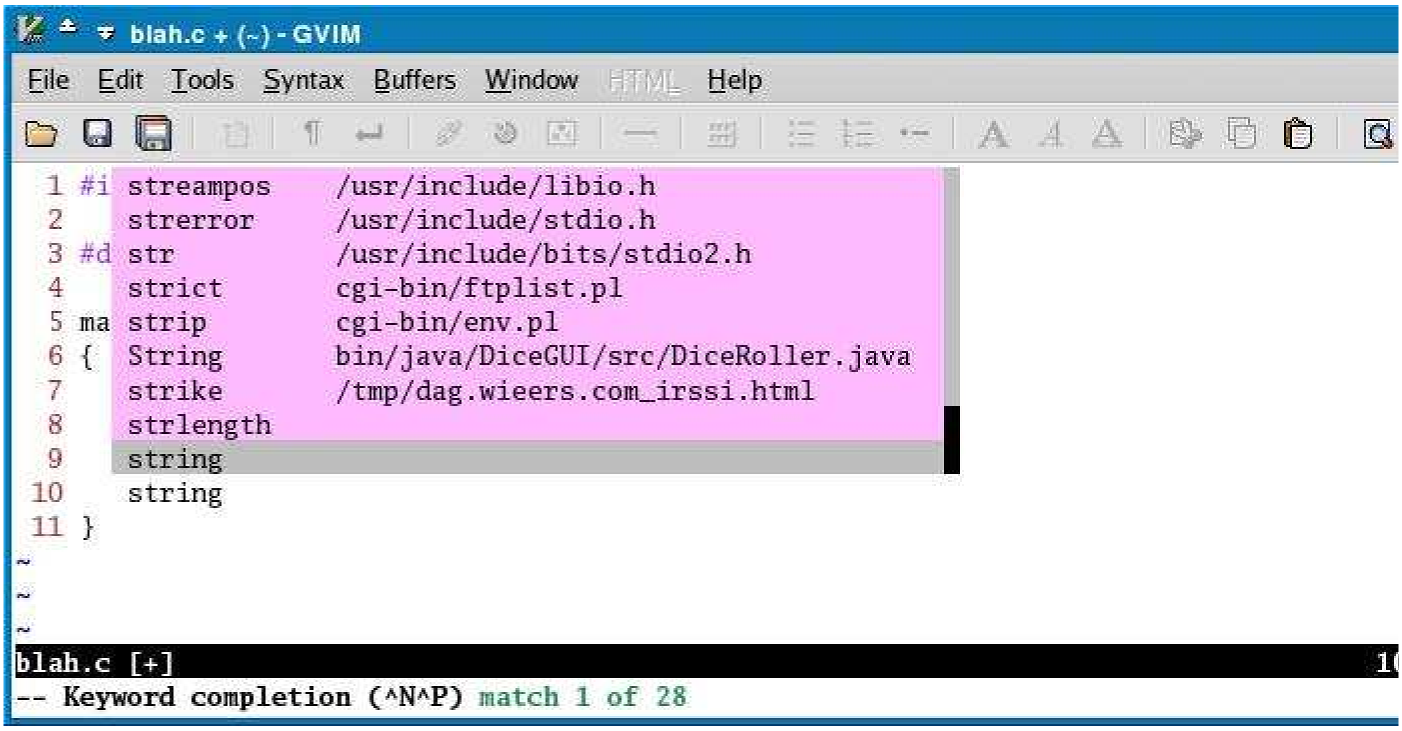
Generally, any word in the current file, or any of the other files (buffers) you are editing in the same instance of vim, will match for completion. This means once you've typed it once, you can type the first couple letters next time, and press Ctrl-P (several times if you need to cycle through several options) until you find the word you're looking for.

Technically, this isn't true. You can tell vim where to look for words in the `complete` function. In Vim 7, the `complete` function will generally be set to figure out a lot about what you're typing — drawing information from function libraries (As of the last update on this book, the author knows C and C++ are supported by default). Keyword completion since Vim 7 will also show a popup menu.

You can also define a dictionary of your own for completion. For more detail, you might want to consult the vim help system ":help complete", ":help complete-functions", and so forth.

Example 1

As an example, you might edit a C program file, "blah.c". You want a function that starts with "str", but you can't remember what it is. You first type "str". It remains regular text until you press <Ctrl-P> or <Ctrl-N>. In vim 7, you will see a menu appear, like this:



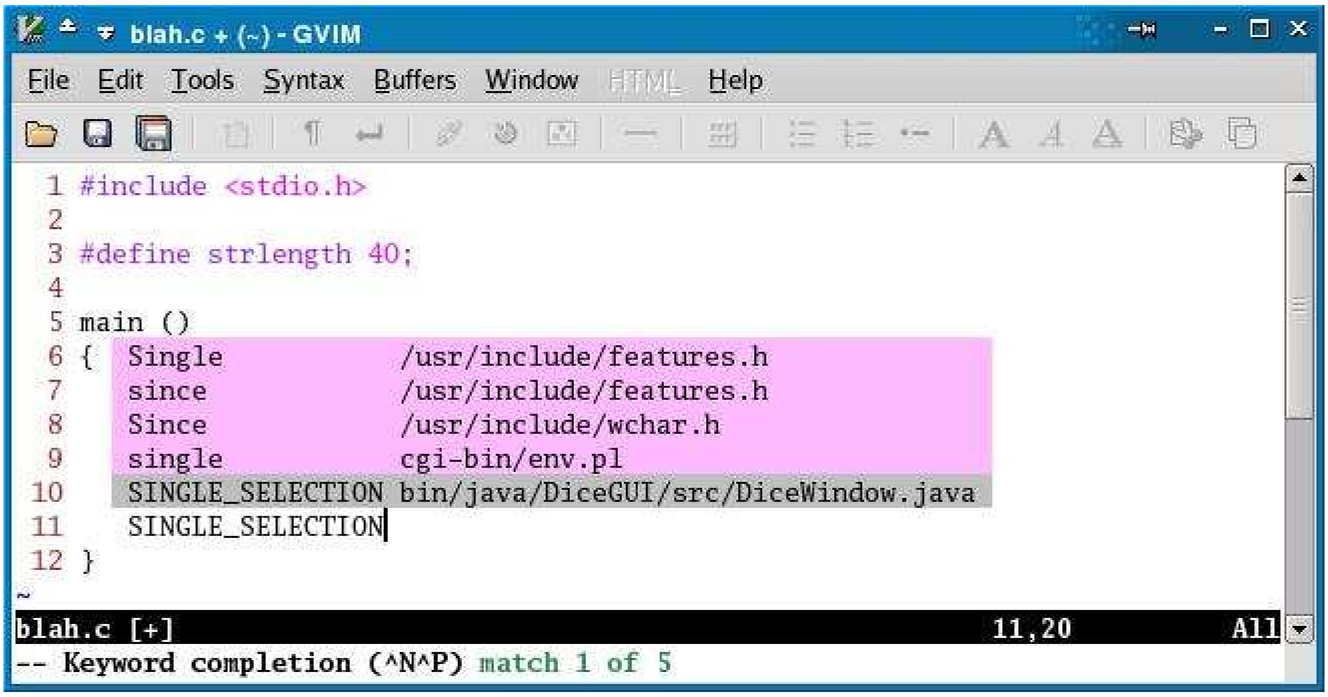
```
#i streampos    /usr/include/libio.h
   strerror     /usr/include/stdio.h
#d str          /usr/include/bits/stdio2.h
   strict       cgi-bin/ftplist.pl
ma strip        cgi-bin/env.pl
{  String       bin/java/DiceGUI/src/DiceRoller.java
   strike       /tmp/dag.wieers.com_irssi.html
   strlength
   string
   string
}
~
~
~
blah.c [+]
-- Keyword completion (^N^P) match 1 of 28
```

You can use <Ctrl-N> and <Ctrl-P> to cycle through the entries shown. In Vim versions 7 and higher, you can actually use the arrow keys to cycle through entries in the menu. There might be too many to show on the screen at once (you will notice the black box on the right represents a scroll position on a gray bar--not all the options are shown on-screen in this example.) The files from which the options were drawn are shown, to help you decide if it's what you're looking for.

(You will also notice in this example that words show up from files the author has recently edited, such as 'strict' from 'cgi-bin/ftplist.pl' -- we certainly don't want that.)

Example 2

Now, suppose you need the sine function, but you know it has an odd name. You type `sin` and press <Ctrl-P>, and it doesn't show up:

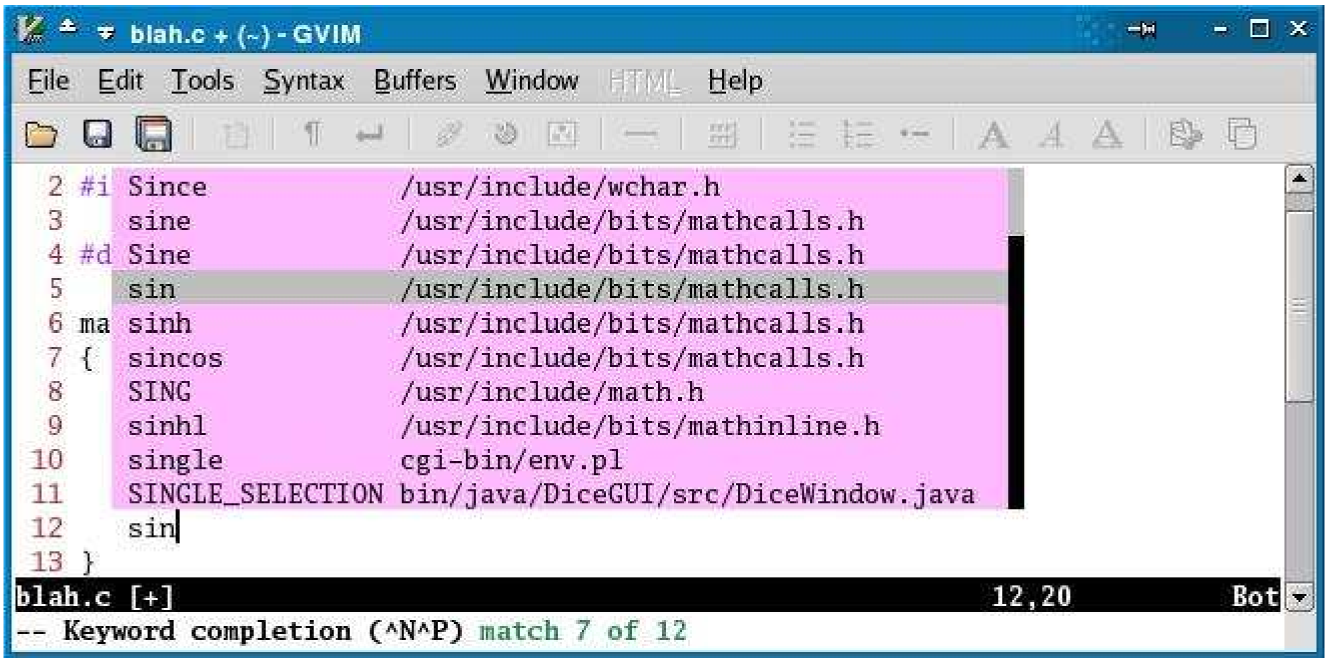


```
#include <stdio.h>

#define strlength 40;

main ()
{  Single           /usr/include/features.h
   since            /usr/include/features.h
   Since            /usr/include/wchar.h
   single           cgi-bin/env.pl
   SINGLE_SELECTION bin/java/DiceGUI/src/DiceWindow.java
   SINGLE_SELECTION
}
~
blah.c [+]                                          11,20          All
-- Keyword completion (^N^P) match 1 of 5
```

You're not out of luck, you just haven't included the math library yet. All you have to do is add the line

```
#include <math.h>
```

and try again. This time you see the function name you wanted (it wasn't easy to remember since it has an odd name)



```
#i Since            /usr/include/wchar.h
   sine             /usr/include/bits/mathcalls.h
#d Sine             /usr/include/bits/mathcalls.h
   sin              /usr/include/bits/mathcalls.h
ma sinh             /usr/include/bits/mathcalls.h
{  sincos           /usr/include/bits/mathcalls.h
   SING             /usr/include/math.h
   sinhl            /usr/include/bits/mathinline.h
   single           cgi-bin/env.pl
   SINGLE_SELECTION bin/java/DiceGUI/src/DiceWindow.java
   sin
}
blah.c [+]                                          12,20          Bot
-- Keyword completion (^N^P) match 7 of 12
```

- *Line Completion*

You can complete entire lines if you need to, though this is less likely. <Ctrl-X>, <Ctrl-L> will load the matching lines (white space matters!) into the menu, and from there you can move forward and backward with arrows or <Ctrl-P> and <Ctrl-N> (for **P**revious and **N**ext)

8.9.2 Indentation

Vim can figure out how to indent most common filetypes.

For most of the popular programming languages, vim can detect the file type by the filename's extension, and from there it will decide how to indent your files. If you don't see it automatically creating the proper indentation for you, try

```
:filetype indent plugin on
```

In the GUI version, you might be able to turn it on at the same time you turn syntax highlighting on for that file. Choose Syntax -> on/off for this file, or Syntax -> Show Filetypes in menu, then go back into the syntax menu and choose the appropriate file type from the list.

You might want to put the above-mentioned ":filetype ..." line in your vimrc file (discussed earlier) and open your program file again, though this really shouldn't be necessary.

If you still have problems, you might want to check that your runtimepath variable is set properly (:help runtimepath). It's also possible (though unlikely) that your programming language is rare enough that nobody has written an indent plugin for it yet. The official site for vim, vim.org, may have an indent plugin file that meets your needs, even if it didn't come with your default installation of vim.

For those times when you've pasted some text in and the indentation is wrong, (your indent plugin must be loaded), you can use the = command. It's probably easiest to type '10=' to re-indent the next ten lines, or to use visual mode and press <=>.

If you want, you can indent lines with ">>" and unindent them with "<<".

If you are in insert mode, use <Ctrl-D> and <Ctrl-T> to change the indentation of the line (<Ctrl-D> decreases indentation by one level and Ctrl-T increases it by one level)

If you can't manage to get filetype specific indentation working, you might try setting one or more of the following options: smartindent, autoindent, cindent, and copyindent. Chances are these won't work completely right, so <Ctrl-D> and <Ctrl-T> will be more important. To turn autoindent on, type `:set autoindent`. To turn autoindent off, type `:set noautoindent`

8.9.3 Repeating commands, or Performing the Same Command Many Times

If you're pretty good with vim, you can record your keystrokes to an invisible buffer and repeat them later. It might be easier to write a vim script, or even filter your file with another program (such as a perl script) for complex enough actions.

That said, sometimes it really is easier to record a command and reuse it, or even perform it on any matching line. You might also consider creating a mapping (discussed below) or running a :global command (also discussed below).

8.9.3.1 Repeating the last single command

Suppose I want to put a semicolon on the end of a few lines, where I forgot:

```
cout << "Hello world\n"
i = j + k
cout << "i is " << i << endl
```

On the first line, I type `A;` followed by the `ESC` key. I move to the next line and press `.` (the period tells vim to repeat the last command -- don't worry, it doesn't duplicate movement commands).

8.9.3.2 Recording a command

To start recording a command, press `q` followed by a buffer name. Buffers have only one character in their name (generally), and you should probably stick to an alphabetic name. Finish the recording by pressing `q` again. Be careful about your movement commands, because you may need to move to the end of a word, not just four characters to the right.

(Warning: Keep in mind that if you are using someone else's scripts or mappings, they may be using the buffer you pick--so if you have a problem, you should consider trying a different buffer)

As a simple example, I have the lines

```
The quick brown fox jumps over the lazy dog
The sly gray fox circles around the unsuspecting rabbit
The slow gray fox crawls under the rotting fence
```

And I want to repeat the last word on each line twice, so that they look like this:

```
The quick brown fox jumps over the lazy dog dog
The sly gray fox circles around the unsuspecting rabbit rabbit
The slow gray fox crawls under the rotting fence fence
```

I start recording into buffer 'r' with `qr`, then append the line with 'A', type a space, then Ctrl-P, then press the `ESC` key. To finish recording, I press `q` again. I move to the next line down, and repeat the command by typing `@r` (which means execute whatever is in buffer r as if it were a command).

In my next example, I have the same sample lines, and I want to put bold html tags around the animal names (fox needs to be <b>fox</b>). To start, move to the first animal named--'fox'. My cursor is positioned on the 'o' in fox. I begin recording by typing 'qa'. Next, I delete the word into buffer 'b' by typing `"bdaw` ("Into buffer *b*, *d*elete *a* *w*ord"). Next, I enter insert mode with 'i' and type '<b>' followed by Ctrl-O, then `"bp` ("from buffer b, put."), then "</b>"' and press ESC. Finally, I finish my recording by typing `q`. I repeat the command on each word which needs to be surrounded by bold tags.

As an exercise, the user might want to make a recording that will insert the text `" = "` after the fourth word on a line, or make a recording to change the next occurence of the word "int" to "float".

8.9.3.3 Mapping a command

One of the advantages of mapping a new command is that you can put the mapping into your vimrc file for use later. I have the following mapping

```
map <C-k> :!%:p<C-m>
```

What this does is maps the *normal* command Ctrl-K to run the current file as a script. You could also perform the second "Recording a command" exercise from above (encase a word in bold tags) with this mapping:

```
map <C-x>b "bdawi<b><C-o>"bp</b><ESC>
```

Note that in this case, I can get away with "<b>" because it doesn't match a special character name. However, if I needed to avoid a conflict (if I wanted to map an insertion of "<ESC>") I would use <lt> and <gt> for the less than and greater-than symbols. (See ":help key-codes")

If you want a to map a command for insert mode, use imap instead of map.

If you want to prevent any of the commands in your map definitions from being interpreted in other mappings, use noremap (or inoremap for insert mode *inoremap meaning **i**nsert mode **no re-map***)

8.10 Enhancing VIM

8.10.1 The .vimrc file

You can make a configuration-file called in your home directory and save any particular settings. The existence of a **vim**rc has the side effect of making vim enable all of vim's incompatible changes to vi, making it more user-friendly. The name of the file depends on the operation system and userinterface used:

.vimrc	Text-User-Interface on UNIX and VMS

_vimrc	Text-User-Interface on MS-Windows and VMS
.gvimrc	Graphical-User-Interface on UNIX and VMS
_gvimrc	Graphical-User-Interface on MS-Windows and VMS

The alternatives with the underscore are for compatiblity with older filesystem. If you use vim on several operating system and use a modern MS-Windows filesystem you don't have to maintain two configurations files. it is perfectly Ok to set _vimrc to:

```
source ~/.vimrc
```

and do all your configurations in `.vimrc`.

This is an example .vimrc file here:

```
"Everything after a double quote is a comment.

"Wrap text after 72 characters
:set textwidth=72

"Set tabs to 4 spaces
:set tabstop=4
:set shiftwidth=4
:set stselect=4
:set expandtab

"tell vim I use a dark background.  Syntax highlighting (color coded tex
:set background=dark

"misc overwrites of default color highlighting.
:hi Comment ctermfg=DarkGreen
:hi String ctermfg=DarkMagenta
:hi pythonPreCondit ctermfg=Green

"make sure that bottom status bar is running.
:set ruler
:set laststatus=2

"make a mapping for "Q" which will reformat the current paragraph, comme
"or code block according to the formatoptions setting:
:map Q gqap
```

8.10.2 Syntax Highlighting

Syntax highighting is what allows you to highlight program code and other files for better readability, using colorization, bold, and other font modifications.

You may want to write simple syntax highlighting statements for easily detected patterns in your file. That said, if you are thinking you need syntax highlighting for html, don't worry: most users do not need to define a syntax highlighting file for common filetypes--most of the file types common developers are interested have already been given a default syntax highlighting definition with vim. Even if it doesn't come with vim, you can usually find someone who has shared their work on vim.org. However, if you need to write something simple, this section is for you. (If you need a syntax highlighting definition that will correctly show perl code even inside an HTML "pre" tag inside a perl print statement within a shell heredoc in a shell script, you're probably out of luck and this section probably won't meet your needs--but you might as well search vim.org, just in case someone has done it for you already).

Syntax Highlighting is one of the most powerful features of VIM. However, it can also be one of the most difficult things to set up--if you don't know what you're doing (or if you just don't have the patience, or if you're dealing with complex program language grammars). So lets have a look at some easy highlighting definitions:

8.10.2.1 Lesson 1: Highlight Tabs

... or how to highlight a special characters

Say you need to know where in your file are tabs and where are spaces. With the following highlight you make tabs visible:

```
:syntax match Special "\t"
```

syntax match matches a regular expression and applies the given color to it. In this case it is the color "Special". You must make sure that the color "Special" has a non standard background - otherwise you won't see a difference:

```
:highlight Special guifg=SlateBlue guibg=GhostWhite
```

You can also create a map in your .vimrc - so you can always activate the tab highlight:

```
:nnoremap <F12><Tab>      :syntax match Special "\t"<CR>
:inoremap <F12><Tab> <C-O>:syntax match Special "\t"<CR>
```

8.10.2.2 Lesson 2: Highlight Space errors

... or how to highlight at the end of line
... or how to find spaces and/or tabs

```
:syntax match Error "\s\+$"
```

8.10.2.3 Lesson 3: Highlight Tab errors

... or how to not highlight characters serving as delimiter

Knowing where tabs are is good. But what about spaces before tabs? They just waste space in your file. The following highlight will show them to you:

```
:syntax match Error " \+\t"me=e-1
```

The regular expression `" \+\t"` searches for one or more space followed by a tab. That alone would solve the problem but would highlight the tab as error as well. Even better would be if we could highlight only the spaces and leave the tab as it is. And indeed this is possible and done by the `me=e-1`. Basically it says: End the highlight one character before the last character found.

8.10.2.4 Lesson 4: Highlight Line length

... or how to highlight at a specific column
... or how to highlight inside other patterns
... or how to allow other pattern inside

The following match will highlight the column 78 when the line is 78 characters long. This can serve as a warning when you have a line which is longer than you wish it to be. Of course, such a highlight should not interfere with any other highlight you might use:

```
:syntax match Error "\(^.\{79\}\)\@<=." contains=ALL containedin=ALL
```

Here is a description of how it works:

1. The regular expression `\(^.\{79}\)` searches for exactly 79 characters from the beginning of the line `^` and group `\( \)` the result.
2. \@<= will now "zero match" the group from 1. "zero match" means the text must be present, but is ignored once found.
3. with `.` one more character is matched. This character is highlighted using the Error colour.
4. With `contains=ALL` we allow other highlight patterns to start inside our pattern.
5. `containedin=ALL` we allow our highlight pattern to start inside another pattern.

An alternative method is suggested by the vim help system. This pair of commands will highlight all characters in virtual column 79 and more:

```
:syntax highlight rightMargin ctermfg=lightblue
:syntax match rightMargin /.\%>79v/
```

And this pair will highlight only column 79:

```
:syntax highlight col79 ctermbg=red
:syntax match col79 /\%<80v.\%>79v/
```

Note the use of two items to also match a character that occupies more than one virtual column, such as a TAB.

8.10.3 Omni Completion

From version 7 onwards vim supports omni completions. This form of completions should work over several files and support all the twirks of a programming language. However, for it to work you need an appropiate "*complete.vim" script in your "autoload/" directory. This script must define a function called **...#Complete** which does all the completion work for the programming language at hand.

However writing a useful complete function can be a diffcult task. All the provided complete functions span several hundred lines of code.

Here a simple implementation used for the Ada programming language described in detail so you can create your own. This implementation need a "tags" file which - for ada - you can create with `gnat xref -v`.

The full version can be download from the vim.org side (http://www.vim.org/scripts/script.php?script_id=1609)

8.10.3.1 Step by Step walkthrue

Set completion with <C-X> <C-O> to autoloaded function. This check is in place in case this script is sourced directly instead of using the autoload feature.

```
if exists ('+omnifunc') && &omnifunc == ""
    setlocal omnifunc=adacomplete#Complete
endif
```

Like most script this script is protected against beeing sourced twice. Also: It won't work with any vim less then 7.00.

```
if exists ('g:loaded_syntax_completion') || version < 700
    finish
else
    let g:loaded_syntax_completion = 20
```

All **complete#Complete** functions have to cover two options: `a:findstart == 1` and `a:findstart != 1`.

```
    function adacomplete#Complete (findstart, base)
```

When `a:findstart == 1` then we have to find out how many characters left of the cursor could be a part of an completion:

```
   if a:findstart == 1
```

For our simple example finding the beginning of the word is pretty simple. We look left until we find a character wich cannot be part of a word. For most languages searching for “\i” should do the trick. However, for Ada we want to expand Attributes as well - hence we add “'” to the list of word characters.

```
      let line = getline ('.')
          let start = col ('.') - 1
          while start > 0 && line[start - 1] =~ '\i\|'''
              let start -= 1
          endwhile
          return start
```

When `a:findstart != 1` then we need to find possible competions for `a:base`. There are two option open to return the found completions:

1. returning them all as a List with **return**.
2. calling **complete_add** for each completion found.

You can also use a combination of both - they will be merged then - so beware not to create duplicates. A completion can either be a String or a Directory.

In this example we use **complete_add**.

```
  else
```

The search patter should look for a:base at the beginning of the text matched.

```
      let l:Pattern    = '^' . a:base . '.*$'
```

In a first step we add all known Ada Keywords, Pragmas, Attributes and Types. They have been prepared as a List of Directorys by the Ada file-type plugin (http://www.vim.org/scripts/script.php?script_id=1548) . All we have to to is iterate over the list and add all where the Directory entry “word” matches the pattern.

```
      if exists ('g:Ada_Keywords')
              for Tag_Item in g:Ada_Keywords
                  if l:Tag_Item['word'] =~? l:Pattern
```

Add the value - incl. simple error handling.

```
                    if complete_add (l:Tag_Item) == 0
                         return []
                     endif
                     {{vi/Ex|if} complete_check ()
                         return []
                     endif
                 endif
             endfor
         endif
```

Here the real work is done: We search for matches inside the tag file. Of corse you need a tag file first. There are many tools to create vim compatible tag files. Just have a look around for one.

```
     let l:Tag_List = taglist (l:Pattern)
```

Again we need to iterate over all List elements found.

```
     for Tag_Item in l:Tag_List
             if l:Tag_Item['kind'] ==
                 let l:Tag_Item['kind'] = 's'
             endif
```

Since the Directory structure for tags and completions are different the data needs to be converted.

The informations available inside the tag depend on the tag-file creation tool. But the minimum is:

“name”
: Name of the tag.

“filename”
: Name of the file where the tag is defined.

“cmd”
: Ex command used to locate the tag in the file.

“kind”
: Type of the tag. The value for this entry depends on the language specific kind values generated by the ctags tool.

The contest of the completion is fixed and contains the following:

“word”
: The actual completion

“kind”
: The type of completion, one character, i.E. “v” for variable.

“menu”
: Short extra info displayed inside the completion menu.

“word”
: Long extra info displayed inside an extra window.

“icase”

Ignore case

So for simple tags without any extras the conversion could look like this:

```
           let l:Match_Item = {
                    \ 'word':  l:Tag_Item['name'],
                    \ 'menu':  l:Tag_Item['filename'],
                    \ 'info':  "Symbol from file " . l:Tag_Item['filenam
                    \ 'kind':  l:Tag_Item['kind'],
                    \ 'icase': 1}
                if complete_add (l:Match_Item) == 0
                    return []
                endif
                if complete_check ()
                    return []
                endif
            endfor
```

Please note

The current Ada plugin has been extended to also supports `ctags` which gives more informations then `gnat xref -v`. However we have not updated the walkthrue as we want to keep the example simple and easy to follow.

We allready added all matches via **complete_add** so we only return an empty list.

```
       return []
        endif
    endfunction adacomplete#Complete
    finish
endif
```

One last advice: It you tag tool does not sort the entries then you should sort them seperatly. Searches on sorted tag-files are significantly faster.

8.11 Ex/Exim Script language

Ex/Exim stands for *Ex-Improved* and is based on a very old editor named Ex. Exim is full feature scripting language (Meaning it can solve almost any text proccessing problem you might have.)

8.11.1 Statements

This section is incomplete. You can help wikimedia by expanding it

8.11.1.1 Assignement

To set a variable use:

```
let variable = expression
```

To set a setup-variable you have two options:

```
set setting = expression
let &setting = "expression"
```

8.11.2 Data types

There are five types of datatype:

8.11.2.1 Number

A 32 bit signed integer.

8.11.2.2 String

A NUL terminated string of 8-bit unsigned characters (bytes). Strings can be created by ‘'’ or ‘"’ qoutes. When using ‘"’ the text is interpreted i.E. "\n" becomes a new line while ‘'’ are not interpreted i.E. '\n' means just that a backslash and a n.

```
let String_1 = "C:\\WinNT"
let String_2 = 'C:\WinNT'
```

Any other datatype can be converted into a string using the **string** () function.

8.11.2.3 Funcref

A reference to a function. A Funcref can be created from a string by the use of the **function** function.

```
let Function_1 = function ("MyFunc")
```

8.11.2.4 List

An ordered sequence of items.

```
let List_1 = [
    \ "a",
    \ "b",
    \ "c"]
```

A list can be created from a string by the use of the **split** function.

```
let List_2 = split ("a b c")
```

8.11.2.5 Dictionary

An associative, unordered array: Each entry has a key and a value.

```
let Dictonary_1 = {
    \ 1: 'one',
    \ 2: 'two',
    \ 3: 'three'}
```

8.11.2.6 Objects

VIM also supports object orientated programming by combining Funcref and Dictionary to an Object:

```
let mydict = {
    \'data': [0, 1, 2, 3]}

function mydict.len () dict
   return len (self.data)
endfunction mydict.len
```

for more informations see Object orientated programming

8.11.3 Control Structures

The existance of control scructures is the main difference between vi's ex commands and vim's exim language. They make the difference between a simple command set (vi) and a full features script language (vim).

8.11.3.1 condition

```
if condition
    operations
elseif condition
    operations
else
    operations
endif
```

8.11.3.2 loop

8.11.3.2.1 while

```
while condition
    operations
endwhile
```

8.11.3.2.2 for

For loops are available from vim 7 onwards. They iterate over List or Directory structures.

```
for var in list
    operations
endfor
```

8.11.3.3 exceptions

```
try
    operations
catch /pattern/
    error handling operations
finally
    clean-up operations
endtry
```

8.11.4 Subprograms

8.11.4.1 Simple Subprograms

Like most Shell-Languages all subprograms are stored in separate files which you load either with the **source** or **runtime** command. The difference lies in the use of a search path. **runtime** uses a search path and allows wildcards to find the sub-program while **source** need the full patch. The following command do the same - provided that "~/vimfiles" is part of your runtime search path:

```
runtime setup.vim
source  ~/vimfiles/setup.vim
```

For both commands need to add the `.vim` extension. Since **runtime** supports both a search path and wildcards more than one match is possible. If you want **runtime** to load all the matches - and not just the first hit - use **runtime!**.

8.11.4.2 Functions

```
function f ( parameter )
    operations
endfunction
```

New with vim 7 is the autoload option for functions. If you name a function *Filename#Functionname* or *Directory#Filename#Functionname* then the function will be automaticly loaded on first call. The file containing the function must be placed in one of the "autload" runtime directories and be named "Filename.vim" or "Directory/Filename.vim". This option is especialy usefull for functions which you don't allways need on in Object orientated programming.

8.11.4.3 Commands

```
command Command Command
```

Command are often used as shortcut for functions and subprograms:

```
command C -nargs=* call F ( <f-args> )
command C source ~/vimfiles/s.vim
```

8.11.5 Object orientated programming

Vim 7 now allows object orientated programming. However, in order to make it real you need to combine several features, namely Dictionaries, Funcrefs and the new function autoload.

The following example class is taken from the gnat compiler plugin for vim. The actual function implemetations have been removed as they are not needed to understand the concept. If you like to have a look at the full version you can download the plugin from vim.org side (http://www.vim.org/scripts/script.php?script_id=1609) .

8.11.5.1 Step by Step walkthrough

We add our new class to a autoload script. That way the class is available when and only when needed: |

```
if exists ("g:loaded_gnat_autoload") || version < 700
    finish
else
   let g:loaded_gnat_autoload=34
```

Each function we define need to be defined with the "dict" attribute. Appart from that they are just normal Exim functions.

```
function gnat#Make () dict
   ...
   return
endfunction gnat#Make

function gnat#Pretty () dict
   ...
   return
endfunction gnat#Make

function gnat#Find () dict
   ...
   return
endfunction gnat#Find

function gnat#Tags () dict
   ...
   return
endfunction gnat#Tags

function gnat#Set_Project_File (...) dict
   ...
   return
endfunction gnat#Set_Project_File

function gnat#Get_Command (Command) dict
   ...
   return ...
endfunction gnat#Get_Command
```

The most important step is the composition of the object. In most OO languages this happens autmaticly - But with vim we have to do this ourself. For best flexibility the use of a so called construtor function is suggested. The contructor is not marked with "dict":

```
function gnat#New ()
```

The contructor creates a dictionary which assigns all the object functions to one element of the dictionary:

```
   let Retval = {
      \ 'Make'             : function ('gnat#Make'),
      \ 'Pretty'           : function ('gnat#Pretty'),
      \ 'Find'             : function ('gnat#Find'),
      \ 'Tags'             : function ('gnat#Tags'),
      \ 'Set_Project_File' : function ('gnat#Set_Project_File'),
      \ 'Get_Command'      : function ('gnat#Get_Command'),
      \ 'Project_File'     : ,
```

We optionaly can now add data entries to our object:

```
   \ 'Make_Command'     : '"gnat make -P " . self.Project_File . "  -F
        \ 'Pretty_Command'   : '"gnat pretty -P " . self.Project_File .
        \ 'Find_Program'     : '"gnat find -P " . self.Project_File . "
        \ 'Tags_Command'     : '"gnat xref -P " . self.Project_File . "
        \ 'Error_Format'     : '%f:%l:%c: %trror: %m,'   .
                             \ '%f:%l:%c: %tarning: %m,' .
                             \ '%f:%l:%c: (%ttyle) %m'}
```

If needed additional modifications to the object are also possible. At this stage you can already use the OO-way:

```
      if argc () == 1 && fnamemodify (argv (0), ':e') == 'gpr'
         call Retval.Set_Project_File (argv(0))
      elseif  strlen (v:servername) > 0
         call Retval.Set_Project_File (v:servername . '.gpr')
      endif
```

The last operation of the contructor it the return of the newly created object.

```
      return Retval
   endfunction gnat#New
```

Is is also possible to defined additional non dict functions. Theese functions are the equivalent to the "static" or "class" methods of other OO languages.

```
   function gnat#Insert_Tags_Header ()
      ...
      return
   endfunction gnat#Insert_Tags_Header

   finish
endif
```

9 vile

9.1 Vile - vi like Emacs

9.1.1 Overview

vile is a vi clone which doesn't claim to be a vi clone. The idea behind vile is to have an editor which works similar to vi but provides features for editing multiple files in multiple window-areas like emacs. In fact, vile development started by using

MicroEMACS as the base, and not a vi clone, and also not full blown Emacs. MicroEMACS is an emacs-like editor (MicroEMACS's author didn't like full-blown emacs, and the vile authors didn't like (Micro)EMACS mode-less way of working). So vile was developed.

vile provides the most common vi commands (as used by their authors), but not all vi commands. The implemented commands are supposed to work more or less like the original vi commands. The window management and buffer management came from MicroEMACS.

Much work has gone into the vile documentation after the first versions were almost undocumented. It is recommended to consult the documentation to find out the differences and extensions of vile, compared to vi.

9.1.2 Resources

- vile (ftp://invisible-island.net/vile/)
- MicroEMACS (http://uemacs.tripod.com/nojavasc.html)

10 BusyBox vi

BusyBox is a very popular program on many embeded Linux systems. In fact, chances are very high to encouner BusyBox if someone works on some embedded Linux system. BusyBox combines tiny versions of many common UNIX utilities into a single relatively small executable. One of the included utilities is a vi clone.

The BusyBox vi clone is limited. Among the limits are:

- It does not support all common vi commands.
- It does not support the '!' command to execute a child process and capture its output
- It also lacks the normal vi crash recovery feature.
- It always assumes a vt102 type terminal (emulator)
- Only very few settings are configurable via `:set`
- `.exrc` configuration and configuration via environment variables are not supported
- Line marks are not correctly adjusted if lines are inserted or deleted before the mark.
- Only whole-line undo (uppercase 'U'), no last-change undo (lowercase 'u') is supported.
- Search is done case-insensitive.
- Command-counts need to prefix a command, and
- command counts for `a`, `c`, `d`, `i`, `r`, `y` and several other commands are not supported.
- Ex commands are not supported.

In short, a lot of information in this vi tutorial is not applicable to BusyBox vi.

However, BusyBox vi also has enhancements over classic vi:

- Curser navigation in insert and command mode
- <INSERT> key changes to insert mode

- No wrapping of long lines. Long lines are displayed via side-scrolling.

10.1 Weblinks

- BusyBox home page (http://busybox.net/)
- BusyBox vi source code (http://busybox.net/cgi-bin/viewcvs.cgi/trunk/busybox/editors/vi.c)

11 vi Reference

The following conventions are used in this reference.

<c>
: A single character, such as 'a' or '1'.

<ESC>, <Ctrl-[>
: Indicates that the Escape (Esc) key on your keyboard should be pressed, which is identical to Control and '['.

<CR>
: Indicates that the Return (Enter) key should be pressed.

<TAB>
: Indicates that the Tabulator key should be pressed

<Ctrl-x>, <C-x>
: Indicates that the Control key and the 'x' key should be pressed simultaneously. 'x' can be almost any other key on your keyboard.

<Shift-x>, <S-x>, <X>
: Indicates that the Shift key and the 'x' key should be pressed simultaneously

<Meta-x>, <M-x>
: Indicates that the Meta or Alt key and the 'x' key should be pressed simultaneously.

:quit, **:q**
: An Ex command. started with <:>, followed by the command and ends with <CR>. For many Ex commands there is a long form (**:quit**) and a short form (**:q**).

/*pattern*/, ?*pattern*?
: A Search pattern. Search pattern in vi are regular expressions.

:*range***s**/*search*/*replace*/*options*, **:global** /*pattern*/ **delete**
: A Search pattern combined with an Ex command.

All commands in *vi* are case sensitive.

c	A single character, such as 'a' or '1'.
m	A single lowercase letter, used to mark text.
string	Several characters, such as 'abc bed'.
pattern	A string used in searching, which may contain regular expressions. For example 'abc' or '^ab[123]'.
myfile	The name of a file to be edited.

11.1 Invocation

vi myfile	Open the file *myfile* for editing. If it does not exist, a new file is created. Multiple files can be opened at the same time.
vi +line myfile	Open the file *myfile* with the cursor positioned at the given line. ▪ **vi +5 myfile** opens *myfile* at line 5. ▪ **vi + myfile** opens *myfile* at the last line.
vi +/string/ myfile	Open the file *myfile* with the cursor positioned at the first line containing the string. If the string has spaces it should be enclosed in quotes. ▪ **vi +/"search string"/ myfile** opens *myfile* at the first line containing *search string*.
vi -r	Lists recovery copies of files. A recovery copy is taken if a **vi** session is killed or the system crashes.
vi -r myfile	Opens a recovery copy of the file *myfile*.
view myfile	**view** is a read only version of **vi**. All **vi** commands, include those to change the file are allowed and act as in **vi**. The difference is that normal attempts to save, **ZZ** or **:wq** do not work. Instead **:x!** or **:w** need to be used.

11.2 vi Commands

11.2.1 Movement

vi can be set up on most systems to use the keyboard movement buttons, such as *cursor left*, *page up*, *home*, *delete*, etc.

<G>	Move to the last line of the file. Can be preceded by a number indicating the line to move to, <1><G> moves to the first line of the file.
<h>	Move left one character, or cursor left. Can be preceded by a number, <5><h> moves left 5 places.
<j>	Move one line down, or cursor down. Can be preceded by a number, <5><j> moves down 5 lines.
<k>	Move one line up, or cursor up. Can be preceded by a number, **5k** moves up 5 lines.
<l>	Move forward one character, or cursor right. Can be preceded by a number, **5l** moves right 5 places.
<H>	Moves to the line at the top of the screen.

<M>	Moves to the line in the middle of the screen.
<L>	Moves to the line at the bottom of the screen.

-	Moves to the first non-whitespace character of the line above. Can be preceded by a number. ■ **10-** moves up 10 lines.
<+>	Moves to the first non-whitespace character of the line below. Can be preceded by a number. ■ **10+** moves down 10 lines.
<CR>	Same as <+>.
<>	Must be preceded by a number. Moves to the specified column on the current line. ■ **10|** moves to column 10.

<w>	Moves to the start of the next word, which may be on the next line.
<W>	As **w** but takes into account punctuation.
<e>	Moves to the end of the current word or to the next word if between words or at the end of a word.
<E>	As **e** but takes into account punctuation.
<b>	Moves backwards to the start of the current word or to the previous word if between words or at the start of a word.
<B>	As **b** but takes into account punctuation.

<f>*c*	Find first occurence of character *c* on the same line. This command may be repeated using <;> or <,> (reverse direction). ■ <3><f><x> moves forward on the third occurence of x (if present). Same as <f><x><;><;>
<F>*c*	Same as **f** but backward.
<t>*c*	Find the character before the first occurence of character *c* on the same line.
<T>*c*	Same as **t** but backward, placing the cursor after character *c*.

<0>	Moves to the start of the current line.
<^>	Moves to the first non-whitespace character on the current line.
<$>	Moves to the end of the current line.

<Ctrl-F> Move forwards one page.

- **5<Ctrl-F>** moves forwards five pages.

<Ctrl-B> Move backwards one page.

- **5<Ctrl-B>** moves backwards five pages.

<Ctrl-D>	Move forwards by half a page.
<Ctrl-U>	Move backwards by half a page.
<Ctrl-E>	Display one more line at the bottom of the screen.
<Ctrl-Y>	Display one more line at the top of the screen.

11.2.2 Inserting

All insert commands put **vi** into *insert mode*. *Insert mode* is terminated by the ESC key.

<i>	Enters *insert mode* at the cursor position.
<I>	Enters *insert mode* at the start of the current line.

<a>	Enters *insert mode* after the cursor, or appends.
<A>	Enters *insert mode* at the end of the current line, or append to the end of the current line.

<o>	Inserts a new line underneath the current line and then goes into *insert mode*.
<O>	Inserts a new line above the current line and then goes into *insert mode*.

11.2.3 Replacing

r Replaces the character underneath the cursor with the next character typed. Can be preceded by a number, **5ra** replaces 5 characters with the letter *a*.

R Enters *replace mode*. Each time a letter is typed it replaces the one under the cursor and the cursor moves to the next character. *Replace mode* is terminated by the ESC key. Can be preceded by a number, **5Rab** followed by ESC replaces the character under the cursor by *a*, the next character by *b* and then inserts another 4 *ab*s. The original line is placed into the buffer, replacing any text already there.

11.2.4 Deleting

Each time a delete command is used, the deleted text is placed into the buffer, replacing any text already in the buffer. Buffered text can be retrieved by **p** or **P**.

dd	Deletes the current line. Can be preceded by a number. ▪ **5dd** deletes five lines. **d5d** is the same as **5dd**.
de	Deletes from the character underneath the cursor to the end of the word. Can be preceded by a number. ▪ **5de** deletes five words. **d5e** is the same as **5de**.
dE	As **de** but takes into account punctuation.
dw	Deletes from the character underneath the cursor to the start of the next word. Can be preceded by a number. ▪ **5dw** deletes five words. **d5w** is the same as **5dw**.
dW	As **dw** but takes into account punctuation.
db	Deletes from the left of the cursor to the start of the previous word. Can be preceded by a number. ▪ **5db** deletes five words to the left of the cursor.
dB	As **db** but takes into account punctuation.
dt*c*	Deletes from the cursor position to before the first instance of the character. ▪ **dta** deletes text up and to, but not including, the first letter 'a'.
df*c*	Deletes from the cursor position to the first instance of the character. ▪ **dfa** deletes text up and to, and including, the first letter 'a'.
dG	Deletes the current line and everything to the end of the file.
d/string	Deletes from the cursor to the string, either forwards or backwards.

D	Deletes from the cursor to the end of the line.
d$	Same as **D**.

d^ Deletes from the left of the cursor to the start of the line.

---

x Delete the character underneath the cursor. Can be preceded by a number.

- **5x** deletes the character underneath the cursor and the next 4 characters.
- **xp** swaps the character underneath the cursor with the one to the right of it.

X Delete the character to the left of the cursor, but will not delete the end of line marker or any characters on the next line. Can be preceded by a number.

- **5X** deletes 5 characters to the left of the cursor.

11.2.5 Changing

The change commands all select text to be removed, the end of which is indicated by a **$**. Insert mode is entered and new text overwrites or extends the text. When the <ESC> key is pressed to terminate the insert, any remaining original text is deleted.

Text deleted during a change is placed into the buffer, replacing any text already there. Buffered text can be retrieved by **p** or **P**.

C Change from the cursor position to the end of the line. Can be preceded by a number.

- **5C** changes 5 lines, the current line and the next 4 lines.

---

cc Change the current line. Can be preceded by a number.

- **5cc** changes 5 lines, the current line and the next 4 lines.

ce Change the current word. Can be preceded by a number.

- **5ce** changes five words. **c5e** is the same as **5ce**.

cw Exactly the same as **ce**.

This command is inconsistent with the ususal vi moving: **ce** is the same as **dei** but **dwi** removes trailing spaces too.

ct*c* Changes from the cursor position to the first instance of the character.

- **cta** changes text up and to, but not including, the first letter 'a'.

cf*c* Changes from the cursor position to the first instance of the character (including the character *c*).

cG Changes from the start of the current line to the end of the file.

---

s Change the character underneath the cursor. Can be preceded by a number.

- **5s** changes 5 characters, the one under the cursor and the next 4.

11.2.6 Cut and Paste

The *yank* commands copy text into the *vi* buffer. Text is also copied into the buffer by delete and change commands. The *put* or *place* commands retrieve text from the buffer.

yy Yanks the current line into the buffer. Can be preceded by a number.

- **5yy** yanks five lines.

Y Same as **yy**.

yw Yanks from the cursor to the start of the next word into the buffer. Can be preceded by a number.

- **5yw** yanks five words.

---

p If the buffer consists of whole lines, they are inserted after the current line. If it consists of characters only, they are inserted after the cursor.

P If the buffer consists of whole lines, they are inserted before the current line. If it consists of characters only, they are inserted before the cursor.

11.2.7 Searching

Searching uses regular expressions.

/*pattern*/ Searches for the string, which could be a regular expression. Searching is from the cursor position downwards, stopping at the first match. If not found, it will continue from the start of the file to the cursor position. The trailing slash character is optional.

- /abc/ seaches for the first occurrence of *abc*.

/*pattern*/+ Goes to the line after the one containing the search string.

- /abc/+3 goes to the third line after the one containing *abc*.

/*pattern*/e	Leaves the cursor on the last character of the string that *pattern* matched.* By adding **+***num* or **-***num* after **e** you can supply an offset in characters to where the cursor gets left. For example: /foo/e+3 will leave the cursor 3 characters past the next occurance of **foo**.* By using **b** instead of **e** you can specify a character offset from the beginning of the matched string.
/**\c***pattern*/	Does a case insensitive search.
?*pattern*?	As /*pattern*/ but searches upwards. The trailing question mark character is optional.
?*pattern*?-	Goes to the line above the one containing the search string.

- ?abc?-3 goes to the third line above the one containing *abc*.

<n>	Repeat last search.
<N>	Repeat last search but in the opposite direction.
<f>*char*	Search forward on the current line for the next occurance of *char*.
<F>*char*	Search backward on the current line for the next occurance of *char*.
<;>	Repeat the last **f** or **F** search.

11.2.8 Search and Replace

Search and replace uses regular expressions and the Ex command **:substitute** (short **:s**) which has syntax similar to the sed utility - which is not supprising sed, Ex and w:Vi have common roots - the Ed editor.

:.**s**/*pattern*/*replacement*/	Replaces the first occurance of *pattern* on the current line with *replacement*.* If *pattern* contains **\(** and **\)** they are used to remember what matched between them instead of matching parenthesis characters. For example **:.s/\(\d*\)-\(\d*\)/\2:\1/** could match the string **12345-6789** and substitute **6789:12345** for it.
:.**s**/*pattern*/*replacement*/g	Replaces all occurances of *pattern* on the current line with *replacement*.
:%**s**/*pattern*/*replacement*/g	Replaces all occurances of *pattern* in the whole file with *replacement*.
:x,y**s**/*pattern*/*replacement*/g	Replaces all occurances of *pattern* on lines *x* through *y* with *replacement*.* For example: :14,18**s**/foo/bar/g will replace all occurances of **foo** with **bar** on lines 14 through 18.

- The character **.** can be used to indicate the current line and the character **$** can be used to indicate the last line. For example: :.,$**s**/foo/bar/g will replace all occurances of **foo** with **bar** on the current line through the end of the file.

11.2.9 Mark Text

Marked lines can be used when changing or deleting text.

<m>*m* Mark the current line with the letter.

- <m><a> marks the current line with the letter *a*.

<'>*m* Move to the line marked by the letter.

- <'><a> moves to the line marked by *a*.

11.2.10 Screen Refresh

<Ctrl-L> Refresh the screen.

z<CR> Refreshes the screen so that the current line is at the top. Can be preceded by a line number.

- **35z** refreshes the screen so that line 35 is at the top.

/pattern/z Finds the line with the first occurrence of *string* and then refreshes the screen so that it is at the top.

z. Refreshes the screen so that the current line is in the middle of the screen. Can be preceded by a line number, in which case the line is at the middle.

- **35z.** refreshes the screen so that line 35 is in the middle.

/string/z. Finds the line with the first occurrence of *string* and then refreshes the screen so that it is in the middle.

z- Refreshes the screen so that the current line is at the bottom. Can be preceded by a line number, in which case the line is at the bottom.

- **35z-** refreshes the screen so that line 35 is at the bottom.

/string/z- Finds the line with the first occurrence of *string* and then refreshes the screen so that it is at the bottom.

11.2.11 Others

<~> Changes the case of the character underneath the cursor and moves to the next character. Can be preceded by a number, so that **5~** changes the case of 5 characters.

<.> Repeats the last insert or delete. Can be preceded by a number, **dd** followed by **5.** deletes a line and then deletes another 5 lines.

<%> Moves the cursor to the matching bracket, any of (), [] or {}.

<Ctrl-G>	Temporarily displays a status line at the bottom of the screen.
:f	Same as <Ctrl-G>.
<J>	Joins the next line to the end of the current line. Can be preceded by a number. Both **1J** and **2J** do the same as **J**. ▪ **3J** joins three lines together, the current line and the next two lines.
<u>	Undoes the last change. A second **u** puts the change back.
<U>	Undoes all changes to the current line.
<Ctrl-Z>	Puts *vi* into the background, that is control is returned to the operating system. In UNIX, the *vi* session can be returned to the foreground with **fg**.

11.2.12 Saving and Quitting

<Z><Z>	Saves and quits. It is symbolic of sleep, indicating the end of work.
:quit **:q**	Quits, but only if no changes have been made.
:quit! **:q!**	Quits without saving, regardless of any changes.
:write **:w**	Saves the current file without quitting. ▪ {Vi/Ex|:write!}} **myfile** saves to the file called *myfile*.
:write!*filename* **:w!***filename*	Saves to the file, overwriting any existing contents.
:wq **:write|quit**	Saves and quits.
:exit **:xit** **:x**	Saves and quits.
:exit! **:xit!** **:x!**	Used to save and quit in **view**.

11.2.13 Files

:e filename	Quits the current file and starts editing the named file.
:e + filename	Quits the current file and starts editing the named file with the cursor at the end of the file. ▪ **:e +5 myfile** quits the current file and begins editing *myfile* at

line 5.

:e!	The current file is closed, all unsaved changes discarded, and the file is re-opened for editing.
:e#	Quits the current file and starts editing the previous file.
:n	When multiple files were quoted on the command line, start editing the next file.
:n files	Resets the list of files for **:n**. The current file will be closed and the first file in the list will be opened for editing.
:r filename	Read a file, that is insert a file.

- **:r myfile** inserts the file named *myfile* after the cursor.
- **:5r myfile** inserts the file after line 5.

11.3 vi Options

All options are *ex* options, and so require an initial colon.

Default options may be placed into a file in the user's home directory called **.exrc**. Options in this file do not have the initial colon, e.g.

set ic

:set all Displays all the current settings.

! Set on	! Set off	! Meaning
:set ignorecase **:set ic**	**:set noignorecase** **:set noic**	Ignore case. Makes searching case insensitive.
:set list	**:set nolist**	Shows control characters. *<Ctrl-I>* is tab, *$* is linefeed.
:set number **:set nu**	**:set nonumber** **:set nonu**	Turns on line numbering.
:set term		Displays the terminal type.

11.4 ex Commands

ex commands start with **:**, which puts *vi* into *last line mode*, entered on the last line of the screen. Spaces within the command are ignored.

:! command Executes the named operating system command and then returns to **vi**.

- **:! ls** runs the UNIX *ls* command.

:sh Starts up a shell. **exit** returns to the **vi** session.

:vi Exit *last line mode* and return to normal *command mode*.

11.4.1 ex line commands

These commands edit lines and have the following syntax:

1. No line number, meaning work on the current line.
2. With **%**, meaning work on all lines.
3. A pair of line numbers, such as '3,5' meaning work on lines 3 to 5 inclusive. Either number can be replaced with **.**, standing for the current line or **$** standing for the last line. So **.,$** means from the current line to the end of the file and **1,$** means the same as **%**. Additionally simple arithmetic may be used, so **.+1** means the line after the current line, or **$-5** means 5 lines before the last line.

co Copy, followed by the line position to copy to.

- **:co 5** copies the current line and places it after line 5.
  - **:1,3 co 4** copies lines 1 to 3 and places after line 4.

d Delete.

- **:d** deletes the current line.
  - **:.,.+5d** delete the current line and the next 5 lines.
  - **:%d** deletes all lines.

m Move, followed by the line position to move to.

- **:m 10** moves the current line and places it after line 10.
  - **:1,3 m 4** moves lines 1 to 3 and places after line 4.

11.4.2 Mapping / Remapping vi Commands

:map Create new command or overwrite existing in vi command mode.

- **:map v i--<Ctrl-[>** new command **v** will insert -- and return to command mode. <Ctrl-[> is the escape character typed as <CTRL-V><ESC>.

:map! Create new command in both comamnd and insert mode.

- **:map! ;r <Ctrl-[>** typing *;r* in insert mode will return to command mode.

11.5 External link

- vim Official Reference Manual (http://vimdoc.sourceforge.net/htmldoc/ref_toc.html)

12 Authors

This book has many authors, including the public: it is open for anyone and everybody to improve. Therefore, this is more properly a list of acknowledgements of contributors than a list of authors. Whoever we are, this is where we get to brag about our accomplishments in writing this book.

12.1 List of major contributors

- Dysprosia (Contributions)
- Martin Krischik Learning_vi:Vim, Template:Vi (Contributions)
- N.N.(anonymous): The adanced/Power User/Extra Details part (Learning_vi:Details) - whatever it is currently called .
- Others (add your name and description if you made a major contribution)
- Various anonymous persons

Learning the vi editor: Getting acquainted — Basic tasks — Making your work easier — Advanced tasks — Details — Vi clones (**Vim** – Basic navigation – Modes – Tips and Tricks – Useful things for programmers to know – Enhancing Vim – Exim Script language, **Vile**, **BB vi**) — vi Reference