A byte of Vim

Swaroop C. H.

25 November 2008

Chapter 1

Vim

1.1 Introduction

"A Byte of Vim" is a book which aims to help you to learn how to use the Vim editor (version 7), even if all you know is how to use the computer keyboard.

The first part of this book is meant for new users who want to understand what Vim is and learn how to use it.

The second part of this book is for people who already know how to use Vim and want to learn about features that make Vim so powerful, such as windows and tabs, personal information management, making it a programmer's editor, how to extend Vim with your own plugins, and more.

1.2 License and Terms

2. Attribution *must* be shown by linking back to and clearly indicating that the original text can be fetched from this location.
3. All the code/scripts provided in this book is licensed under the 3-clause BSD License unless otherwise noted.
4. Some sample text used in this book has been retrieved from and under the GNU Free Documentation License.
5. Volunteer contributions to this original book must be under this same license *and* the copyright must be assigned to the main author of this book.

Chapter 2

Vim Preface

2.1 About Vim

Vim is a computer program used for writing which provides a range of features to help you **write better**.

2.2 Why Vim?

Let's face it, it's very rare to produce your best work on the first attempt. Most likely, you will keep editing it frequently until it becomes 'good'.

As Louis Brandeis once said:

> *There is no great writing, only great rewriting.*

Making these numerous rapid changes would be a lot easier if we had a capable editor to help us, and that is *exactly* where Vim shines, and is far better compared to most plain text editors and rich document editors.

2.3 Why Write This Book?

I have been using the Vim editor ever since I learned to use the old vi editor during Unix classes in college. Vim is one of the few pieces of software that I use for nearly 10 hours a day. I knew there were many features which I didn't know about that could potentially be useful to me, so one day I started exploring Vim little by little.

To crystallize my understanding and to help others also explore Vim, I started writing this collection of notes, and called it a book.

Some of the principles I have tried to keep in mind while writing these notes are:

1. Simple literature. The importance of this should be reinforced again and again.
2. Emphasis on examples and how-to.
3. The one-stop shop for readers to learn Vim – from getting started to learning advanced stuff.
4. Get the user to understand how to do things the Vim way – from modes to buffers to customization. Most people learn only the basic vi commands and do not attempt to learn anything beyond that. Learning such concepts is the tipping point, they become hardcore Vim users, i.e., Vimmers, which means they extract the most out of Vim, which is the intent of this book.
5. A lot of things are documented and stored here as a reference for people such as how to use Vim as an IDE, etc. There are various ways of doing it and instead of the user struggling to figure out which plugins to try out, the book already has the basic background work already for the reader.
6. Just enough info to get you to understand and use, not everything required (Pareto principle)
7. Relatedly, the book shouldn't attempt to rewrite the reference manual. Where appropriate, it should simply point out the relevant parts. This way, there is no redundancy, the user learns to use the awesome built-in reference manual which is important, *and* the book can stand on its own strengths as well.

To summarize, the mantra is *Concepts. Examples. Pithy.*

2.4 Something To Think About

> *Books aren't written – they're rewritten. Including your own. It is one of the hardest things to accept, especially after the seventh rewrite hasn't quite done it.*
>
> – *Michael Crichton*

> *Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.*
>
> – *Antoine de Saint-Exupery*

Chapter 3

Vim Introduction

3.1 What is Vim?

Vim is a computer program used for writing any kind of text, whether it is your shopping list, a book, or software code.

What makes Vim special is that it is both *simple and powerful*.

Simple means it is easy to get started with. Simple means that it has a minimalistic interface that helps you to concentrate on your main task – writing. Simple means it is built around few core concepts that help you learn deeper functionality easily.

Powerful means getting things done more quickly, better, and more easily. Powerful means making not-so-simple things possible. Powerful does not mean complicated. Powerful means following the paradigm of *Minimal effort. Maximal effect.*

3.2 What can Vim do?

I can hear you say, "So it's a text editor. What's the big deal anyway?"

Well, a lot.

Let's see some random examples to compare Vim with your current editor. The point of this exercise is for you to answer the question *"How would I do this in the editor I currently use?"* for each example.

> Note:
>
> Don't worry too much about the details of the Vim commands here, the point here is to enlighten you with the possibilities, not to start explaining how these things work. That is what the rest of the book is for.

1. How do you move the cursor down by 7 lines?

- Press `7j`

2. How do you delete a word? Yes, a "word".

- Press `dw`

3. How do you search the file for the current word that the cursor is currently placed on?

- Press `*`

4. How to do a find and replace only in lines 50-100?

- Run `:50,100s/old/new/g`.

5. What if you wanted to view two different parts of the same file simultaneously?

- Run `:sp` to 'split' the view

6. What if you wanted to open a file whose name is in the current document and the cursor is placed on that name?

- Press `gf` (which means 'g'o to this 'f'ile)

7. What if you wanted to choose a better color scheme for the display?

- Run `:colorscheme desert` to choose the 'desert' color scheme (my favorite)

8. What if you wanted to map the keyboard shortcut `ctrl-s` to save the file?

- Run `:nmap :w`. Note that " means a 'c'arriage 'r'eturn, i.e., the enter key.

9. What if you wanted to save the current set of open files as well as any settings you have changed, so that you can continue editing later?

- Run `:mksession ~/latest_session.vim`, and open Vim next time as `vim -S ~/latest_session.vim`.

10. What if you wanted to see colors for different parts of your code?

- Run `:syntax on`. If it doesn't recognize the language properly, use `:set filetype=Wikipedia`, for example.

11. What if you wanted different parts of your file to be folded so that you can concentrate on only one part at a time?

- Run `:set foldmethod=indent` assuming your file is properly indented. There are other methods of folding as well.

12. What if you wanted to open multiple files in tabs?

- Use `:tabedit` to open multiple files in "tabs" (just like browser tabs), and use `gt` to switch between the tabs.

13. You use some words frequently in your document and wish there was a way that it could be quickly filled in the next time you use the same word?

- While in insert mode, press `ctrl-x ctrl-n` to see the list of "completions" for the current word, based on all the words that you have used in the current document. Switch between possibilities with `ctrl-n`ext and `ctrl-`previous Alternatively, use `:ab mas Maslow's hierarchy of needs` to expand the abbreviation automatically when you type `m a s`.

14. You have some data where only the first 10 characters in each line are useful and the rest is no longer useful for you. How do you get only that data?

- Press `ctrl-v`, select the text and press `y` to copy the selected rows and columns of text.

15. What if you received a document from someone which is in all caps, find it irritating and want to convert it to lower case?

- In Vim, run the following:

```
:for i in range(0, line('$'))
:    call setline(i, tolower(getline(i)))
:endfor
```

Don't worry, other details will be explored in later chapters. A more succinct way of doing this would be to run `:%s#(.)#l1#g` but then again, it's easier to think of the above way of doing it.

There is an even simpler method of selecting all the text (`ggVG`) and using the `u` operator to convert to lowercase, but then again that's too easy, and isn't as cool as showing off the above way of making Vim do steps of actions.

Phew. Are you convinced yet?

In these examples, you can see the power of Vim in action. Any other editor would make it insanely hard to achieve the same level of functionality. And yet, amazingly, all this power is made as understandable as possible.

Notice that we didn't use the mouse even once during these examples! This is a good thing. Count how many times you shift your hand between the keyboard and the mouse in a single day, and you'll realize why it is good to avoid it when possible.

Don't be overwhelmed by the features here. The best part of Vim is that you don't need to know all of these features to be productive with it, you just need to know a few basic concepts. After learning those basic concepts, all the other features can be easily learned when you need them.

Chapter 4

Vim Installation

Let's see how to get Vim installed on your computer.

4.1 Windows

If you use Microsoft Windows, then the following steps will help you get the latest version of Vim 7 installed on your computer:

1. Visit http://www.vim.org/download.php#pc
2. Download the "Self-installing executable" (`gvim72.exe` as of this writing)
3. Double-click the file and install Vim like any other Windows-based software.

4.2 Mac OS X

If you use Mac OS X, then you already have the terminal version of Vim installed. Run the menu command Finder → Applications → Utilities → Terminal. In the terminal, run the command `vim` and press enter, you should now see the Vim welcome screen.

If you want to use a graphical version of Vim, download the latest version of the Cocoa-based MacVim project. Double-click the file (such as `MacVim-7_2-stable-1_2.tbz`), it will be unarchived and a directory called `MacVim-7_2-stable-1_2` will be created. Open the directory, and copy the `MacVim` app to your `Applications` directory.

For more details MacVim differences, including how to run MacVim from the terminal see the macvim reference:

1. Click on Finder → Applications → MacVim.
2. Type `:help macvim` and press the Enter key.

4.3 Linux/BSD

If you are using a Linux or *BSD system, then you probably have at least a minimal console version of Vim already installed. Open a terminal program such as `konsole` or `gnome-terminal`, run `vim` and you should see the Vim welcome screen.

If you get a message like `vim: command not found`, then Vim is not installed. You will have to use your system-specific tools to install Vim, such as `aptitude` in Ubuntu/Debian Linux, `yum` in Fedora Linux, `pkg_add` or `port` in FreeBSD, etc. Please consult your specific system's documentation and forums on how to install new packages.

If you want the graphical version, install the `vim-gnome` package or alternatively, the `gvim` package.

4.4 Summary

Depending on how it is installed, you can run the `vim` command in the shell or use your operating system's menus to open a graphical version of the `Vim` application.

Now that we have Vim installed on your computer, let us proceed to use it in the next chapter.

Chapter 5

Vim First Steps

5.1 Starting Vim

First step is, of course, to learn how to start Vim.

5.1.1 Graphical version

Windows Click on Start → Programs → Vim 7 → gVim. Or type the windows key + r type 'gvim'

Mac OS X Click on Finder → Applications → MacVim.

Linux/BSD Click on Applications → Accessories → GVim Text Editor, or press Alt+F2, type `gvim` and press the enter key.

5.1.2 Terminal version

Windows Click on Start → Run, type `vim` and press the enter key.

Mac OS X Click on Finder → Applications → Utilities → Terminal, type `vim` and press the enter key.

Linux/BSD Click on Applications → Accessories → Terminal, or press Alt+F2, type `konsole`/`gnome-terminal` and press the enter key. Then, type `vim` and press the enter key.

From now onwards when we say 'open Vim', use either of the two methods mentioned above.

> Note:
>
> When you started Vim, you might have noticed that you can't immediately start typing text. Don't panic, all will be explained in a little while.

5.2 Graphical or Terminal?

The graphical version of Vim has menus at the top of the application as well as various options accessible via the mouse, but note that this is completely optional. You can still access all the features of Vim using *only* the keyboard.

Why is this important? Because once a person becomes efficient at typing, using only the keyboard makes the person much faster and less error-prone, as opposed to using the mouse. This is because the hand movement required to switch between the keyboard and the mouse is slow and there is a context switch required in the mind of the person when shifting the hand between the keyboard and the mouse. If you make it a habit to use the keyboard as much as possible, you're saving valuable hand movement.

Of course, this is subjective. Some people prefer the mouse and some prefer the keyboard. I encourage you to use the keyboard as much as possible to experience the real power of Vim.

5.3 Introduction to Modes

Imagine it's a Saturday evening and you're bored with the shows on television. You want to watch an old favorite movie instead. So, you *switch the TV to video mode* so that it shows what the DVD player is displaying instead of the cable channels. Note that the television is still displaying video, but you switch the context on whether you want to watch a DVD or a live television channel.

Similarly, Vim has modes. For example, Vim has a mode for writing text, a mode for running commands, etc. They are all related to the main purpose of editing text, but you switch context depending on whether you want to simply type stuff or you want to run some commands on the text.

Isn't that simple?

Traditionally, the concept of modes is the most oft-cited reason by beginners on why they find Vim "confusing". I compare it to riding a bicycle – you'll trip a few times, but once you've got the hang of it, you'll wonder what the fuss was all about.

So why does Vim have modes? To make things as simple as possible, even though its usage may seem "strange" at first.

What do I mean by that? Let's take an example – one of the key goals in Vim is to make it fully accessible from the keyboard without ever having to need to use a mouse (you can still use the mouse if you want to but it is strictly optional). In such a scenario, how would you distinguish between the text you want to write, and the commands that you want to run? Vim's solution is to have a "normal" mode where you can execute commands and an "insert" mode where you are simply writing text. You can switch between the two modes any time. For example, pressing `i` switches Vim to insert mode, and pressing " switches Vim back to normal mode.

How do traditional editors achieve this distinction between commands and writing text? By using graphical menus and keyboard shortcuts. The problem is that this does not scale. First of all, if you have hundreds of commands, creating menus for each of these commands would be insane and confusing. Secondly, customizing how to use each of these commands would be even more difficult.

Let's take a specific example. Suppose you want to change all occurrences of the word "from" to the word "to" in a document. In a traditional editor, you can run a menu command like Edit -> Replace (or use a keyboard shortcut like Ctrl-R) and then enter the 'from' word and the 'to' word and then click on 'Replace'. Then, check the 'Replace All' option. In Vim, you simply run `:%s/from/to/g` in the normal mode. See how simple this is?

> Explanation of the substitute command:
>
> Consider the command `:%s/from/to/g`. The `%` is a shortcut for the range on which we want to run the command and it means "all the lines of the file". Alternatively you can specify the range of lines such as `0,10` which means lines 0 to 10. The `:s` is the "substitute" command. The 'from' and 'to' are the respective text to be found and replaced. The `g` (for "global") indicates that if there is more than one 'from' string on the same line, it should replace all of them. At first glance, it seems like a lot of things, but imagine you learn this once, and every time it is available with a key press away in your command mode instead of having to hunt down menus and fiddle around with a dialog box to do exactly what you want.

What if you want to now run this substitution only in the first 10 lines of the text and you want to have a yes/no confirmation for each replacement? In traditional text editors, achieving the yes/no confirmation is easy by unchecking the 'Replace All' option, but notice that you have to first search for this option and then use the mouse to click on the option (or use a long series of keys using the keyboard). But how will you run the Replace for only the first 10 lines? In Vim, you can simply run `:0,10s/from/to/gc`. The new `c` option we are using means we want a 'c'onfirmation message for every replace.

By separating the writing (insert) and command (normal) modes, Vim makes it easy for us to switch the two contexts easily.

Notice how the first steps to using Vim seem a little "weird", a little "strange", but once you have seen it in action, it starts to make sense. The best part is that these core concepts will help you to understand all you need to know on how to use Vim.

Since you now understand the difference between normal mode and insert mode, you can look up the various commands you can run in the normal mode, and you can immediately start using them. Compare that to learning new commands in traditional editors which generally means having to read a lot of documentation, searching a lot of menus, a lot of trial and error or plain asking someone for help.

Personally, I find the names of the modes not intuitive to beginners. I prefer calling the insert mode as "writing" mode and the normal mode as "rewriting" mode, but we will stick to the standard Vim terminology to avoid confusion.

Note:

All commands in the normal mode should end with the enter key to signal Vim that we have written the full command. So, when we say run `:help vim-modes-intro`, it means you should type `:help vim-modes-intro` and then press the enter key at the end of the command.

5.4 Writing a file

Let us see how to open, write and close a file.

1. Open Vim.
2. Type `:edit hello.txt` and press the enter key.
3. Press `i`.
4. Type the text `Hello World`.
5. Press the `<Esc>` key.
6. Type `:write` and press the enter key.
7. Close Vim by running `:q`.

Congratulations! You just wrote your first file using Vim:-).

Does this seem like a lot of steps? Yes, it does, *at first*. That is because this is the first time we are getting used to opening Vim, writing a file and closing Vim. You have to keep in mind that this will only be a minor part of your time compared to the actual time that goes into editing the content of the document.

Let us see what the above commands do.

- `:edit hello.txt` or simply `:e hello.txt`
  - This opens a file for *e*diting. If the file with the specified name does not exist, it will be created the first time we "save" the file.
- Press `i`
  - This switches Vim to the insert mode
- Type the text `Hello World`
  - This is where you type the actual text that you want to write.
- Press `<Esc>`
  - This escapes Vim back to normal mode
- `:write` or simply `:w`
  - This tells Vim to *w*rite the text (which is currently stored in a temporary file created by Vim automatically) to the file on the hard disk. This means that whatever we wrote so far is now permanently stored.
- `:quit` or simply `:q` to quit the file in the "window" that we are editing. If there was only one "window" open, this will also close Vim (Concept of windows will be discussed in a later chapter).

Try to repeat this process a few times with different file names, different text, etc. so that you get used to the basic set of steps in using Vim.

Notice that when you are in insert mode, Vim displays -- INSERT -- at the bottom left corner. When you switch to normal mode, it will not display anything. This is because normal mode is the *default* mode in which Vim runs.

Take some time to soak in this information, this is probably the hardest lesson there is to learn about Vim, the rest is easy.

And don't worry, help is not too far away. Actually, it's just a `:help` command away. For example, run `:help:edit` and you'll see the documentation open up. Go ahead, try it.

5.5 Summary

We have now discussed the basic concepts and usage of Vim. See `:help notation` and `:help keycodes` also.

Be sure to understand these concepts well. Once you start "thinking in Vim", understanding the rest of Vim's features is easy.

Chapter 6

Vim Modes

6.1 Introduction

We had our first encounter with modes in the *previous chapter*. Now, let us explore the concept of modes and what we can do in each mode.

6.2 Types of modes

There are three basic modes in Vim – normal, insert and visual.

- Normal mode is where you can run commands. This is the default mode in which Vim starts up.
- Insert mode is where you insert, i.e., write the text.
- Visual mode is where you visually select a bunch of text so that you can run a command/operation only on that part of the text.

6.3 Normal mode

By default, you're in normal mode. Let's see what we can do in this mode.

Type `:echo "hello world"` and press enter. You should see the famous words `hello world` echoed back to you. What you just did was run a Vim command called `:echo` and you supplied some text to it which was promptly printed back.

Type `/hello` and press the enter key. Vim will search for that phrase and will jump to the first occurrence.

This was just two simple examples of the kind of commands available in the normal mode. We will see many more such commands in later chapters.

6.3.1 How to use the help

Almost as important as knowing the normal mode, is knowing how to use the `:help` command. This is where you learn more about the commands available in Vim.

Remember that you do not need to know every command available in Vim; it's better to simply know where to find them when you need them. For example, see `:help usr_toc` takes us to the table of contents of the reference manual. You can see `:help index` to search for the particular topic you are interested in, for example, run `/insert mode` to see the relevant information regarding insert mode.

If you can't remember these two help topics at first, just press `F1` or simply run `:help`.

6.4 Insert mode

When Vim starts up in normal mode, we have seen how to use `i` to get into insert mode. There are other ways of switching from normal mode to insert mode as well:

- Run `:e dapping.txt`
- Press `i`
- Type the following paragraph (including all the typos and mistakes, we'll correct them later):

  means being determined about being determined and being passionate about being passionate

- Press `<Esc>` key to switch back to normal mode.
- Run `:w`

Oops, we seem to have missed a word at the beginning of the line, and our cursor is at the end of the line, what do we do now?

What would be the most efficient way of going to the start of the line and insert the missing word? Should we use the mouse to move the cursor to the start of the line? Should we use arrow keys to travel all the way to the start of the line. Should we press home key and then press `i` to switch back to insert mode again?

It turns out that the most efficient way would be to press `I` (upper case i):

- Press `I`
- Type `Dappin`
- Press `<Esc>` key to switch back to the normal mode.

Notice that we used a different key to switch to insert mode. Its specialty is that it moves the cursor to the start of the line and then switches to the insert mode.

Also notice how important it is to *switch back to the normal mode as soon as you're done typing the text.* Making this a habit will be beneficial because most of your work (after the initial writing phase) will be in the normal mode – that's where the all-important rewriting/editing/polishing happens.

Now, let's take a different variation of the `i` command. Notice that pressing `i` will place your cursor before the current position and then switch to insert mode. To place the cursor 'a'fter the current position, press `a`.

- Press `a`
- Type `g` (to complete the word as "Dapping")
- Press `<Esc>` to switch back to normal mode

Similar to the relationship between `i` and `I` keys, there is a relationship between the `a` and `A` keys – if you want to append text at the end of the line, press the `A` key.

- Press `A`
- Type `.` (put a dot to complete the sentence properly)
- Press `<Esc>` to switch back to the normal mode

To summarize the four keys we have learnt so far:

Command	Action
i	insert text just before the cursor
I	insert text at the start of the line
a	append text just after the cursor
A	append text at the end of the line

Notice how the upper case commands are 'bigger' versions of the lower case commands.

Now that we are proficient in quickly moving in the current line, let's see how to move to new lines. If you want to 'o'pen a new line to start writing, press the `o` key.

- Press `o`
- Type `I'm a rapper.`
- Press `<Esc>` to switch back to the normal mode.

Hmmm, it would be more appealing if that new sentence we wrote was in a paragraph by itself.

- Press `O` (upper case 'O')
- Press `<Esc>` to switch back to the normal mode.

To summarize the two new keys we just learnt:

Command	Action
o	open a new line below
O	open a new line above

Notice how the upper and lower case 'o' commands are opposite in the direction in which they open the line.

Was there something wrong in the text that we just wrote? Aah, it should be 'dapper', not 'rapper'! It's a single character that we have to change, what's the most efficient way to make this change?

We *could* press `i` to switch to insert mode, press `<Esc>` key to delete the `r`, type `d` and then press `<Esc>` to switch back to normal mode. But that is four steps for such a simple change! Is there something better? You can use the `s` key – s for 's'ubstitute.

- Move the cursor to the character `r` (or simply press `b` to move 'b'ack to the start of the word)
- Press `s`
- Type `d`
- Press `<Esc>` to switch back to the normal mode

Well, okay, it may not have saved us much right now, but imagine repeating such a process over and over again throughout the day! Making such a mundane operation as fast as possible is beneficial because it helps us focus our energies to more creative and interesting aspects. As Linus Torvalds says, *"it's not just doing things faster, but because it is so fast, the way you work dramatically changes."*

Again, there is a bigger version of the `s` key, `S` which substitutes the whole line instead of the current character.

- Press `S`
- Type `Be a sinner.`
- Press `<Esc>` to switch back to normal mode.

Command	Action
s	substitute the current character
S	substitute the current line

Let's go back to our last action… Can't we make it more efficient since we want to 'r'eplace just a single character? Yes, we can use the `r` key.

- Move the cursor to the first character of the word `sinner`.
- Press `r`
- Type `d`

Notice we're already back in the normal mode and didn't need to press `<Esc>`.

There's a bigger version of `r` called `R` which will replace continuous characters.

- Move the cursor to the 'i' in `dinner`.
- Press `R`
- Type `app` (the word now becomes 'dapper')
- Press `<Esc>` to switch back to normal mode.

Command	Action
r	replace the current character
R	replace continuous characters

The text should now look like this:

> *Dapping means being determined about being determined and being passionate about being passionate.*
>
> *Be a dapper.*

Phew. We have covered a lot in this chapter, but I guarantee that this is the only step that is the hardest. Once you understand this, you've pretty much understood the heart and soul of how Vim works, and all other functionality in Vim, is just icing on the cake.

To repeat, understanding how modes work and how switching between modes works is the key to becoming a Vimmer, so if you haven't digested the above examples yet, please feel free to read them again. Take all the time you need.

If you want to read more specific details about these commands, see `:help inserting` and `:help replacing`.

6.5 Visual mode

Suppose that you want to select a bunch of words and replace them completely with some new text that you want to write. What do you do?

One way would be to use the mouse to click at the start of the text that you are interested in, hold down the left mouse button, drag the mouse till the end

of the relevant text and then release the left mouse button. This seems like an awful lot of distraction.

We could use the `<Del>` or `<Backspace>` keys to delete all the characters, but this seems even worse in efficiency.

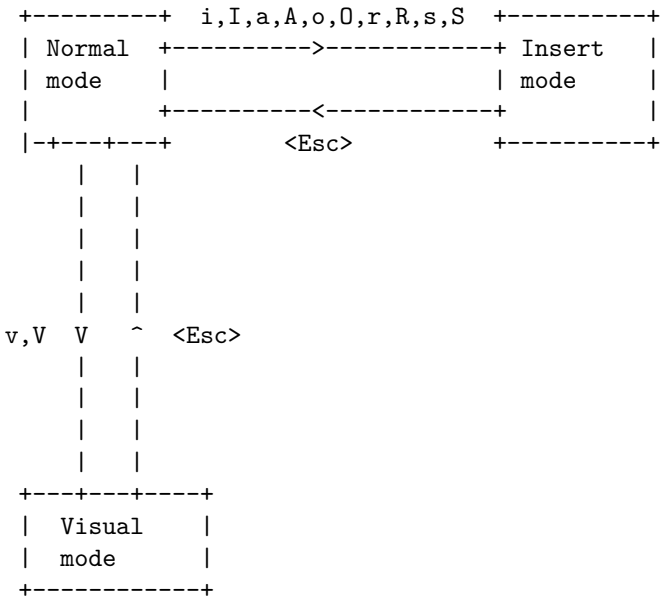
The most efficient way would be to position the cursor at the start of the text, press `v` to start the visual mode, use arrow keys or any text movement commands to move to the end of the relevant text (for example, press `5e` to move to the end of the 5th word counted from the current cursor position) and then press `c` to 'c'hange the text. Notice the improvement in efficiency.

In this particular operation (the `c` command), you'll be put into insert mode after it is over, so press `<Esc>` to return to normal mode.

The `v` command works on a character basis. If you want to operate in terms of lines, use the upper case `V`.

6.6 Summary

Here is a drawing of the relationship between the different modes:



See `:help vim-modes-intro` and `:help mode-switching` for details on the various modes and how to switch between them respectively.

If you remain unconvinced about why the concept of modes is central to Vim's power and simplicity, do read the articles on "Why Vi" and about the vi input model on why it is a better way of editing.

Chapter 7

Vim Typing Skills

7.1 Introduction

We just learned about how to switch between the three basic modes in Vim. Isn't it hard to remember all those keys? How do you remember which key is for which operation? Well, that's the "key" question. The answer is that you shouldn't "remember", *your fingers should automatically know what to do*! It must (literally) be on your finger tips.

So how do we do that? By making it a habit. Walking is not a nature that humans have at birth, but after trying a bit and by habit, we learn to walk. Same with Vim, although it requires less effort.

Vim was designed for people who are familiar with the keyboard. Why? Because we spend most of our time in editing tasks, which implicitly make extensive use of the keyboard, and the faster we are able to type, the faster we get the work done.

Let's start with a basic technique to get you comfortable with the keyboard.

7.2 Home Row Technique

Place your fingers on the home row of the keyboard by positioning your hands such that the fingers of your left hand are on the `ASDF` keys and the fingers of your right hand are on the `JKL;` keys as shown in the drawing (artist unknown).

Getting your hands to be comfortable in this position is one of the most important steps in learning how to use the keyboard effectively. The idea is that you should be able to type any key using the finger that is closest to that key and then your finger should automatically come back to its original position. It might seem difficult at first but try it a couple of times and you will see that you will type much faster this way.

Note that most keyboards have some home row markers on the `F` and `J` keys which serves as a reminder for you on where your fingers should be placed.

Now, try typing the alphabets A-Z using the home row technique.

Relatedly, there is also a free online typing tutorial available that explains the basics of typing skills. I would encourage you to try it for just ten minutes and explore.

7.3 Vim graphical keyboard cheat sheet

If you want to know how each key can map to something useful in Vim, see this Vim graphical cheat sheet by 'jng'.

Although a *lot* of commands are listed, for now you only need to learn the basic 'hjkl' keys which translates to left, down, up, right keys respectively. You'll learn more about this in the next chapter.

7.4 Summary

Notice that our efficiency in using Vim is directly proportional to the efficiency of using the keyboard.

Chapter 8

Vim Moving Around

8.1 Introduction

Once you've written the initial text, editing and rewriting requires a lot of movement between the various parts of the document. For example, you're writing a story and you suddenly get an idea for a new plot, but to develop this plot you need to go back to the part where the protagonist enters the new city (or something like that)… how do you quickly move around the text so that you don't lose your train of thought?

Let's see a few examples of how Vim makes this fast.

Note All the movements work *from the current cursor position.*

- Want to move the cursor to the next word? Press `w`.
- Want to move to to the next paragraph? Press `}`.
- Want to move to the 3rd occurrence of the letter 'h'? Press `3fh`.
- Want to move 35 lines downwards? Press `35j`.
- After one of the above movements, want to jump back to the previous location? Press `ctrl-o`.
- Want to learn how all these work? Let's dive in.

First, open a file called `chandrayaan.txt` and type the following text (taken from Wikipedia):

> Chandrayaan-1 is India's first mission to the moon. Launched by India's national space agency the Indian Space Research Organisation (ISRO). The unmanned lunar exploration mission includes a lunar orbiter and an impactor. The spacecraft was launched by a modified version of the PSLV XL on 22 October 2008 from Satish Dhawan Space Centre, Sriharikota, Andhra Pradesh at 06:23 IST (00:52 UTC). The vehicle was successfully inserted into lunar orbit on 8 November 2008. The Moon Impact Probe was successfully impacted at the lunar south pole at 20:31 hours on 14 November 2008.

> The remote sensing satellite had a mass of 1,380 kilograms (3,042 lb) at launch and 675 kilograms (1,488 lb) at lunar orbit and carries high resolution remote sensing equipment for visible, near infrared, and soft and hard X-ray frequencies. Over a two-year period, it is intended to survey the lunar surface to produce a complete map of its chemical characteristics and 3-dimensional topography. The polar regions are of special interest, as they might contain ice. The lunar mission carries five ISRO payloads and six payloads from other international space agencies including NASA, ESA, and the Bulgarian Aerospace Agency, which were carried free of cost.

8.2 Move your cursor, the Vim way

The most basic keys that you should use are the 'hjkl' keys. These 4 keys correspond to the left, down, up and right arrow keys respectively. Notice these keys are situated directly under your right hand when they are placed on the home row.

But why not use the arrow keys themselves? The problem is that they are located in a separate location in the keyboard and it requires as much hand movement as it requires to use a mouse.

Remember, that the right hand fingers should always be placed on `jkl;` keys (and the thumb on the space bar). Now, let's see how to use these 4 keys:

Using h,j,k,l instead of arrow keys

h	You have to stretch your index finger (which is on 'j') to the left to press the 'h'. This is the left-most key and signifies going left.
j	The drooping 'j' key signifies going down.
k	The upward pointing 'k' key signifies going up.
l	The right-most 'l' key signifies going right.

Note that we can repeat the operation by prefixing a count. For example, `2j` will repeat the `j` operation `2` times.

Open up the `chandrayaan.txt` text document and start practicing these keys:

- Position your cursor at the first letter 'C' of the document.
- Press `2j` and it should skip the current long line, the blank line and go to the second line, i.e., second paragraph.
- Press `2k` to get back to where we were. Or alternatively, press `ctrl-o` to jump back.
- Press `5l` to move 5 characters to the right.
- Press `5h` to move left by 5 characters. Or alternatively, press `ctrl-o` to jump back.

Make it a habit to use the `'hjkl'` keys instead of the arrow keys. Within a few tries, you'll notice how much faster you can be using these keys.

Similarly, there are more simple keys that replace the following special movements. Notice that this again is intended to reduce hand movement. In this particular case, people are prone to searching and hunting for these special keys, so we can avoid that altogether.

Traditional	Vim
'home' key moves to the start of the line	`^` key (think 'anchored to the start')
'end' key moves to the end of the line	`$` key (think 'the buck stops here')
'pgup' key moves one screen up	`ctrl-b` which means move one screen 'b'ackward
'pgdn' key moves one screen down	`ctrl-f` which means move one screen 'f'orward

If you know the absolute line number that you want to jump to, say line 50, press `50G` and Vim will jump to the 50th line. If no number is specified, `G` will take you to the last line of the file. How do you get to the top of the file? Simple, press `1G`. Notice how a single key can do so much.

- Move the cursor to the first line by pressing `1G`.
- Move 20 characters to the right by pressing `20l`.
- Move back to the first character by pressing `^`.
- Jump to the last character by pressing `$`.
- Press `G` to jump to the last line.

What if you wanted to move to the middle of the text that is currently being shown in the window?

- Press H to jump as 'h'igh as possible (first line of the window)
- Press M to jump to the 'm'iddle of the window
- Press L to jump as 'l'ow as possible (last line being displayed)

You must have started to notice the emphasis on touch-typing and never having to move your hands off the main area. That's a good thing.

8.3 Words, sentences, paragraphs

We have seen how to move by characters and lines. But we tend to think of our text as words and how we put them together – sentences, paragraphs, sections, and so on. So, why not move across such text parts, i.e., "text objects"?

Let's take the first few words from our sample text:

The polar regions are of special interest, as they might contain ice.

First, let's position the cursor on the first character by pressing `^`.

[T]he polar regions are of special interest, as they might contain ice.

Note that we are using the square brackets to mark the cursor position.

Want to move to the next 'w'ord? Press `w`. The cursor should now be at the 'p' in 'polar'.

The [p]olar regions are of special interest, as they might contain ice.

How about moving 2 words forward? Just add the prefix count to 'w': `2w`.

The polar regions [a]re of special interest, as they might contain ice.

Similarly, to move to the 'e'nd of the next word, press `e`.

The polar regions ar[e] of special interest, as they might contain ice.

To move one word 'b'ackward, press `b`. By prefixing a count, `2b` will go back by 2 words.

The polar [r]egions are of special interest, as they might contain ice.

See `:help word-motions` for details.

We have seen character motions and word motions, let's move on to sentences.

> [C]handrayaan-1 is India's first mission to the moon. Launched by India's national space agency the Indian Space Research Organisation (ISRO). The unmanned lunar exploration mission includes a lunar orbiter and an impactor. The spacecraft was launched by a modified version of the PSLV XL on 22 October 2008 from Satish Dhawan Space Centre, Sriharikota, Andhra Pradesh at 06:23 IST (00:52 UTC). The vehicle was successfully inserted into lunar orbit on 8 November 2008. The Moon Impact Probe was successfully impacted at the lunar south pole at 20:31 hours on 14 November 2008.

Position the cursor at the first character (`^`).

To move to the next sentence, press `)`.

> Chandrayaan-1 is India's first mission to the moon. [L]aunched by India's national space agency the Indian Space Research Organisation (ISRO). The unmanned lunar exploration mission includes a lunar orbiter and an impactor. The spacecraft was launched by a modified version of the PSLV XL on 22 October 2008 from Satish Dhawan Space Centre, Sriharikota, Andhra Pradesh at 06:23 IST (00:52 UTC). The vehicle was successfully inserted into lunar orbit on 8 November 2008. The Moon Impact Probe was successfully impacted at the lunar south pole at 20:31 hours on 14 November 2008.

Isn't that cool?

To move to the previous sentence, press `(`.

Go ahead, try it out and see how fast you can move. Again, you can prefix a count such as `3)` to move forward by 3 sentences.

Now, use the whole text and try out moving by paragraphs. Press `}` (right curly brace) to move to the next paragraph and `{` (left curly brace) to move to the previous paragraph.

Notice that the 'bigger' brackets is for the bigger text object. If you had already noticed this, then congratulations, you have already started to think like a winner, err, "think like a Vimmer".

Again, don't try to *remember* these keys, try to make it a *habit* such that your fingers naturally use these keys.

See `:help cursor-motions` for more details.

8.4 Make your mark

You are writing some text but you suddenly remember that you have to update a related section in the same document, but you do not want to forget where you are currently so that you can come back to this later. What do you do?

Normally, this would mean scrolling to that section, update it, and then scroll back to where you were. This is a lot of overhead and we may tend to forget where we were last at.

We can do things a bit smarter in Vim. Move the cursor to the 5th line in the following text (the words by John Lennon). Use `ma` to create a mark named 'a'. Move the cursor to wherever you want, for example `4j`.

> *I am eagerly awaiting my next disappointment.*
>
> — *Ashleigh Brilliant*
>
> *Every man's memory is his private literature.*
>
> — *Aldous Huxley*
>
> *Life is what happens to you while you're busy making other plans.*
>
> — *John Lennon*

> *Life is really simple, but we insist on making it complicated.*
>
> — *Confucius*
>
> *Do not dwell in the past, do not dream of the future, concentrate the mind on the present moment.*
>
> — *Buddha*
>
> *The more decisions that you are forced to make alone, the more you are aware of your freedom to choose.*
>
> — *Thornton Wilder*

Press `'a` (i.e., single quote followed by the name of the mark) and voila, Vim jumps (back) to the line where that mark was located.

You can use any alphabet (a-zA-Z) to name a mark which means you can have up to 52 named marks for each file.

8.5 Jump around

In the various movements that we have learned, we might often want to jump back to the previous location or to the next location after a movement. To do this, simply press `ctrl-o` to jump to the previous location and `ctrl-i` to jump forward to the next location again.

8.6 Parts of the text

There are various ways you can specify text objects in Vim so that you can pass them to a command. For example, you want to visually select a part of the text and then convert the case (from upper to lower or from lower to upper case) of the text using the `~` key.

Open the `dapping.txt` file that we created in previous chapters. Use the various keys to move to the first letter of the word 'dapper' in the second paragraph. Hint: Use `}`, `j`, `w`.

> *Dapping means being determined about being determined and being passionate about being passionate.*
>
> *Be a dapper.*

Press `v` to start the visual mode, and press `ap` to select 'a' 'p'aragraph. Press `~` to flip the case of the text. If you want to cancel the selection, simply press `<Esc>`.

> *Dapping means being determined about being determined and being passionate about being passionate.*
>
> *bE A DAPPER.*

Other text object mnemonics are `aw` which means 'a' 'w'ord, `a"` means a quoted string (like "this is a quoted string"), `ab` means 'a' 'b'lock which means anything within a pair of parentheses, and so on.

See `:help object-motions` and `:help text-objects` for more details.

8.7 Summary

We have seen the rich number of methods that Vim gives us to move around the text. It is not important to remember each of these movements, it is more important to make them a habit whenever you can, especially the ones that are most relevant to you, and when they become a habit they reduce the movement of your hands, you become faster, and ultimately spend more time on thinking about your writing rather than on the software you use to write.

See `:help various-motions` as well as `:help motion` for more interesting ways of movement.

Chapter 9

Vim Help

9.1 Introduction

Vim has such a diverse list of commands, keyboard shortcuts, buffers, and so on. It's impossible to remember how all of them work. In fact, it is not even useful to know all of them. The best situation is that you know how to look for certain functionality in Vim whenever you need it.

For example, you want to avoid having to type a long name every time, you suddenly remember there is an abbreviations feature in Vim that'll help you do just that, but don't remember how to use it. What do you do?

Let's look at the various ways of finding help about how to use Vim.

9.2 The :help command

The first and most important place to try to look for help is the built-in documentation and Vim has one of the most comprehensive user manuals that I've ever seen.

In our case, just run `:help abbreviation` and you'll be taken to the help for `abbreviations` and you can read about how to use the `:ab` and `:iab` commands.

Sometimes, it can be as simple as that. If you don't know what you're looking for, then you can run `:help user-manual` and browse through the list of contents of the entire user manual and read the chapter that you feel is relevant to what you're trying to do.

9.3 How to read the :help topic

Let us take some sample text from `:help abbreviate`:

```
:ab[breviate] [<expre>] {lhs} {rhs}
        add abbreviation for {lhs} to {rhs}.  If {lhs} already
        existed it is replaced with the new {rhs}.  {rhs} may
        contain spaces.
        See |:map-<expr>| for the optional  argument.
```

Notice that there is a standard way of writing help in Vim to make it easy for us to figure out the parts that are needed for us instead of trying to understand the whole command.

The first line explains the syntax, i.e., how to use this command.

The square brackets in `:ab[breviate]` indicate that the latter part of the full name is optional. The minimum you have to type is `:ab` so that Vim recognizes the command. You can also use `:abb` or `:abbr` or `:abbre` and so on till the full name `:abbreviate`. Most people tend to use the shortest form possible.

The square brackets in `[<expr>]` again indicate that the 'expression' is optional.

The curly brackets in `{lhs} {rhs}` indicate that these are placeholders for actual arguments to be supplied. The names are short for 'left hand side' and 'right hand side' respectively.

Following the first line is an indented paragraph that briefly explains what this command does.

Notice the second paragraph which points you to further information. You can position the cursor on the text between the two pipe symbols and press `ctrl-]` to follow the "link" to the corresponding `:help` topic. To jump back, press `ctrl-o`.

9.4 The :helpgrep command

If you do not know what the name of the topic is, then you can search the entire documentation for a phrase by using `:helpgrep`. Suppose you want to know how to look for the beginning of a word, then just run `:helpgrep beginning of a word`.

You can use `:cnext` and `:cprev` to move to the next and previous part of the documentation where that phrase occurs. Use `:clist` to see the whole list of all the occurrences of the phrase.

9.5 Quick help

Copy the following text into a file in Vim and then also run it:

```
:let &keywordprg=':help'
```

Now, position your cursor anywhere on the word `keywordprg` and just press `K`. You'll be taken to the help immediately for that word. This shortcut avoids having to type `:help keywordprg`.

9.6 Summary

There is a wealth of information on how to do things using Vim, and many Vimmers would gladly help you out as well. The Vim community is one of the greatest strengths of the Vim editor, so make sure to use the resources and do join the growing community as well.

> *The true delight is in the finding out rather than in the knowing.*
>
> *Isaac Asimov*

Chapter 10

Vim Editing Basics

10.1 Introduction

Let's learn the basic editing commands in Vim for reading/writing files, cut/copy/paste, undo/redo and searching.

10.2 Reading and writing files

10.2.1 Buffers

When you edit a file, Vim brings the text in the file on the hard disk to the computer's RAM. This means that a copy of the file is stored in the computer's memory and any changes you make is changed in the computer's memory and immediately displayed. Once you have finished editing the file, you can save the file which means that Vim writes back the text in the computer's memory to the file on the hard disk. The computer memory used here to store the text temporarily is referred to as a "buffer". Note that this same concept is the reason why we have to "save" files in all editors or word processors that we use.

Now open up Vim, write the words `Hello World` and save it as the file `hello.txt`. If you need to remember how to do this, please refer to the *First Steps* chapter.

10.2.2 Swap

Now you will notice that another file has been created in the same directory as this file, the file would be named something like `.hello.txt.swp`. Run the following command to find out the exact name of the file:

```
:swapname
```

What is this file? Vim maintains a backup of the buffer in a file which it saves regularly to the hard disk so that in case something goes wrong (like a computer crash or even Vim crashes), you have a backup of the changes you have made since you last saved the original file. This file is called a "swap file" because Vim keeps swapping the contents of the buffer in the computer memory with the contents of this file on the hard disk. See `:help swap-file` to know more details.

10.2.3 Save my file

Now that the file has been loaded, let's do a minor edit. Press the `~` key to change the case of the character on which the cursor is positioned. You will notice that Vim now marks the file having been changed (for example a `%2B` sign shows up in the title bar of the GUI version of Vim). You can open up the actual file in another editor and check that it has not changed yet, i.e., Vim has only changed the buffer and not yet saved it to the hard disk.

We can write the buffer back to the original file in the hard disk by running:

```
:write
```

> Note:
>
> To make saving easier, add the following lines to your *vimrc file*:
>
> ```
> " To save, ctrl-s.
> nmap <c-s> :w<CR>
> imap <c-s> <Esc>:w<CR>a
> ```
>
> Now you can simply press `ctrl-s` to save the file.

10.2.4 Working in my directory

Vim starts up with your home directory as the default directory and all operations will be done within that directory.

To open files located in other directories, you can use the full or relative paths such as:

```
:e ../tmp/test.txt :e C:shoppingmonday.txt
```

Or you can switch Vim to that directory:

```
:cd ../tmp
```

`:cd` is short for 'c'hange 'd'irectory.

To find out the current directory where Vim is looking for files:

```
:pwd
```

`:pwd` is short for 'p'rint 'w'orking 'd'irectory.

10.3 Cut, Copy and Paste

As Sean Connery says, in the movie *'Finding Forrester'*:

> *No thinking – that comes later. You must write your first draft with your heart. You rewrite with your head. The first key to writing is… to write, not to think!*

When we rewrite, we frequently rearrange the order of the paragraphs or sentences, i.e., we need to be able to cut/copy/paste the text. In Vim, we use a slightly different terminology:

desktop world	vim world	operation
cut	delete	d
copy	yank	y
paste	put	p

In normal desktop terminology, 'cut'ting text means removing the text and placing it on the clipboard. The same operation in Vim means it deletes the text from the file buffer and stores it in a 'register' (a part of the computer's memory). Since we can choose the register where we can store the text, it is referred to as the "delete" operation.

Similarly, in normal desktop terminology, 'copy' text means that a copy of the text is placed on the clipboard. Vim does the same, it "yanks" the text and places it in a register.

"Put" means the same as "Paste" in both terminologies.

We have seen how you can do cut/copy/paste operations in Vim. But how do you specify which text that these operations should work on? Well, we have already explored that in the previous *Text Objects* section.

Combining the operation notation and the text object notation means we have innumerable ways of manipulating the text. Let's see a few examples.

Write this text in Vim (exactly as shown):

```
This is the rthe first paragraph.
This is the second line.

This is the second paragraph.
```

Place the cursor at the topmost leftmost position, by pressing `1G` and `|` that moves to the first line and the first column respectively.

Let's see the first case: We have typed an extra 'r' which we have to remove. Press `3w` to move forward by 3 words.

Now, we need to delete one character at the current cursor position.

Note that there are two parts to this:

Operation	Text Object/Motion
Delete	d
One character at current cursor position	l (lower case L)

So, we have to just press `dl` and we delete one character! Notice that we can use `l` even though it is a motion.

Now we notice that the whole word is redundant because we have "the" twice. Now think carefully on what should be fastest key combination to fix this?

Take some time to think and figure this out for yourself. If not, please read on.

Operation	Text Object/Motion
Delete	d
Word	w

So, press `dw` and you delete a word. Voila! So simple and so beautiful. The beauty is that such simple concepts can be combined to provide such a rich range of possibilities.

How do we achieve the same operation for lines? Well, lines are considered special in Vim because lines are usually how we think about our text. As a shortcut, if you repeat the operation name twice, it will operate on the line. So, `dd` will delete the current line and `yy` will yank the current line.

Our example text in Vim should now look like this:

```
This is the first paragraph.
This is the second line.

This is the second paragraph.
```

Go to the second line by pressing `j`. Now press `dd` and the line should be deleted. You should now see:

```
This is the first paragraph.

This is the second paragraph.
```

Let's see a bigger case: How do we yank the current paragraph?

Operation	Text Object/Motion
Yank	y
Current paragraph	ap

So, `yap` will copy the current paragraph.

Now that we have done copying the text, how do we paste it? Just 'p' it.

You should now see:

```
This is the first paragraph.
This is the first paragraph.


This is the second paragraph.
```

Notice that the blank line is also copied when we do `yap`, so `p` adds that extra blank line as well.

There are two types of paste that can be done exactly like the two types of inserts we have seen before:

p (lower case)	P (upper case)
paste after current cursor position	paste before current cursor position

Taking the idea further, we can combine these into more powerful ways.

How to swap two characters? Press `xp`.

- `x` → delete one character at current cursor position
- `p` → paste after current cursor position

How to swap two words? Press `dwwP`.

- `d` → delete
- `w` → one word
- `w` → move to the next word
- `P` → paste before the current cursor position

The important thing is *not* to learn these operations by rote. These combinations of operations and text objects/motions should be automatic for your fingers, without you needing to put in mental effort. This happens when you make using these a habit.

10.4 Marking your territory

You are writing, and you suddenly realize you have to change sentences in a previous section to support what you are writing in this section. The problem is that you have to remember where you are right now so that you can come back to it later. Can't Vim remember it for me? This can be achieved using marks.

You can create a mark by pressing `m` followed by the name of the mark which is a single character from `a-zA-Z`. For example, pressing `ma` creates the mark called 'a'.

Pressing `'a` returns the cursor to line of the mark. Pressing "`a` will take you to the exact line and column of the mark.

The best part is that you can jump to this position using these marks any time thereafter.

See `:help mark-motions` for more details.

10.5 Time machine using undo/redo

Suppose you are rewriting a paragraph but you end up muddling up what you were trying to rewrite and you want to go back what you had written earlier. This is where we can "undo" what we just did. If we want to change back again to what we have now, we can "redo" the changes that we have made. Note that a change means some change to the text, it does not take into account cursor movements and other things not directly related to the text.

Suppose you have the text:

> *I have coined a phrase for myself – 'CUT to the G':*
>
> 1. *Concentrate*
> 2. *Understand*
> 3. *Think*
> 4. *Get Things Done*
>
> *Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing – It's all in the mind. That's why it's so hard.*

Now, start editing the first line:

1. Press `S` to 's'ubstitute the whole line.
2. Type the text `After much thought, I have coined a new phrase to help me solidify my approach:`.

3. Press `<Esc>`.

Now think about the change that we just did. Is the sentence better? Hmm, was the text better before? How do we switch back and forth?

Press `u` to undo the last change and see what we had before. You will now see the text `I have coined a phrase for myself - 'CUT to the G':`. To come back to the latest change, press `ctrl-r` to now see the line `After much thought, I have coined a new phrase to help me solidify my approach:`.

It is important to note that Vim gives unlimited history of undo/redo changes, but it is usually limited by the `undolevels` setting in Vim and the amount of memory in your computer.

Now, let's see some stuff that really shows off Vim's advanced undo/redo functionality, something that other editors will be jealous of: Vim is not only your editor, it also acts as a time machine.

For example,

```
:earlier 4m
```

will take you back by 4 minutes, i.e., the state of the text 4 minutes "earlier".

The power here is that it is superior to all undoes and redoes. For example, if you make a change, then you undo it, and then continue editing, that change is actually never retrievable using simple `u` again. But it is possible in Vim using the `:earlier` command.

You can also go forward in time:

```
:later 45s
```

which will take you later by 45 seconds.

Or if you want the simpler approach of going back by 5 changes:

```
:undo 5
```

You can view the undo tree using:

```
:undolist
```

See `:help:undolist` for the explanation of the output from this command.

See `:help undo-redo` and `:help usr_32.txt` for more details.

10.6 A powerful search engine but not a dotcom

Vim has a powerful built-in search engine that you can use to find exactly what you are looking for. It takes a little getting used to the power it exposes, so let's get started.

Let's come back to our familiar example:

> *I have coined a phrase for myself – 'CUT to the G':*
>
> 1. *Concentrate*
> 2. *Understand*
> 3. *Think*
> 4. *Get Things Done*
>
> *Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing – It's all in the mind. That's why it's so hard.*

Suppose we want to search for the word "Step". In normal mode, type `/Step<cr>` (i.e., `/Step` followed by enter key). This will take you to the first occurrence of those set of characters. Press `n` to take you to the 'n'ext occurrence and `N` to go in the opposite direction, i.e., the previous occurrence.

What if you knew only a part of the phrase or don't know the exact spelling? Wouldn't it be helpful if Vim could start searching and highlighting as you type the search phrase? You can enable this by running:

```
set incsearch
```

You can also tell Vim to ignore the case (whether lower or upper case) of the text that you are searching for:

```
set ignorecase
```

Or you can use:

```
set smartcase
```

When you have `smartcase` on:

- If you are searching for `/step`, i.e., the text you enter is in lower case, then it will search for any combination of upper and lower case text. For example, this will match all the following four – "Step", "Stephen", "step-brother", "misstep."

- If you are searching for `/Step`, i.e., the text you enter has an upper case, then it will search for ONLY text that matches the exact case. For example, it will match "Step" and "Stephen", but *not* "stepbrother" or "misstep."

  Note:

  I recommend that you put these two lines in your vimrc file (explained later, but see `:help vimrc-intro` for a quick introduction) so that this is enabled by default.

Now that we have understood the basics of searching, let's explore the real power of searching. The first thing to note that what you provide Vim can not only be a simple phrase, it can be a "expression". An expression allows you to specify the 'kinds' of text to search for, not just the exact text to look.

For example, you will notice that `/step` will take you to `steps` as well as `step` and even `footstep` if such a word is present. What if you wanted to look for the exact word `step` and not when it is part of any other word? Then you can search using `/`. The `\<\>` indicate the start and end positions of a "word" respectively.

Similarly, what if you wanted to search for any number? Searching for `/d` will look for a 'digit'. But a number is just a group of digits together. So we specify "one or more" digits together as `/d%2B`. If we were looking for zero or more characters, we can use the `*` instead of the `+`.

There are a variety of such magic stuff we can use in our search patterns. See `:help pattern` for details.

10.7 Summary

We have explored some of the basic editing commands that we will use in our everyday usage of Vim. *It is very important that you go through these concepts again and make them a habit.* It is not important to learn each and every option or nuances of these commands. If you know how to use the command and know how to find out more on it based on what you need, then you're a true Vimmer.

Now, go ahead and start editing!

Chapter 11

Vim More Editing

11.1 Introduction

Let's build upon what we learned in the last chapter to explore more editing tricks in Vim.

11.2 Viewing files and reading commands

We've already seen how to edit and write files, but Vim seems to think there are more things that you can do.

What if you wanted to open a file to read it and not edit it? This can also be done by running `vim -R` which starts Vim in read-only mode. Or if you have the file already open in Vim, then you can run `:set ro` to make the buffer read-only. The advantage of using this option is that viewing large files will be faster because Vim doesn't have to worry about making changes to it.

What if you wanted to insert the contents of another file into the current file? Just run `:r another_file.txt` and the contents of that file will be "read" and inserted into the current buffer. This is useful in many circumstances such as combining two different files or even making a copy of the file and making minor modifications, and so on.

A cool side-effect about the `:r` command is that you can use it to read the output of commands and not just files.

For example, install the *GCal* program and run:

```
:r !gcal -s1 -K
```

This inserts the calendar for the current month (`!gcal`) with the week starting from Monday (`s1`) and also displays the week number (`K`). The text will look something like this:

```
     April 2007
Mo Tu We Th Fr Sa Su CW
                   1 13
 2  3  4< 5> 6  7  8 14
 9 10 11 12 13 14 15 15
16 17 18 19 20 21 22 16
23 24 25 26 27 28 29 17
30                   18
```

Imagine the possibilities of how you can use external commands to add relevant information to your own text…

11.3 Register all these memories

Let's take our usual sample text:

> *I have coined a phrase for myself – 'CUT to the G':*
>
> 1. *Concentrate*
> 2. *Understand*
> 3. *Think*
> 4. *Get Things Done*
>
> *Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing – It's all in the mind. That's why it's so hard.*

Now, let's say you want to copy the 4 bullet points to another place, perhaps in the summary. You also want to cut the second sentence to put it somewhere else. Wouldn't this be easy if you can store these two separate pieces of text in two different places for now, continue our work, and then paste them later? This is achieved using registers, which are (again) parts of your computer's memory, using which you can quickly store and retrieve text.

For example, you can place the cursor on the line containing the text `1\. Concentrate`, press `"a4yy`:

- `"a` → use the register named 'a' for
- `4` → 4 times the operation of
- `yy` → yank a line

This translates to "copy the next 4 lines into the register named 'a' ".

For the next step, we can visually select the second sentence in the last paragraph, and press `"bd` to 'd'elete the text into the register named 'b'.

Now, that we have copied the appropriate text into the buffers, we can paste the text wherever required – just press `"ap` which means 'p'aste the text from the register named 'a' and similarly `"bp` pastes the text from the register named 'b' and so on.

To see the contents of all the registers, run:

```
:registers
```

Notice how Vim takes even the simple concept of clipboard and makes it so powerful!

See `:help registers` for the different types of registers in Vim.

11.4 Text formatting

Do you want to center some text? Let's say like this text:

```
   THIS IS THE HEADING
```

Just run:

```
:set textwidth=70
:center
```

You will get the following result:

```
                          THIS IS THE HEADING
```

Setting `:set textwidth=70` will cause all paragraphs to have a maximum width of 70, and if you write lines longer than 70 characters, Vim automatically moves the word exceeding the length to the next line.

For example, take the text:

Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally important. As Abraham Lincoln once said "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe." And to get to this stage, you need to do Step 1 which boils down to one thing - It's all in the mind. That's why it's so hard.

They've been written with a `textwidth` of 80. We want to reformat the paragraph now to fit into a maximum of 70 columns. So, just run

```
:set textwidth=70
gwap
```

The second command can be understood as:

- `gw` means 'g'o format this text and also go back 'w'here I was
- `ap` means 'a' 'p'aragraph

Voila! The text becomes:

```
Step 4 is eventually what gets you moving, but Steps 2 and 3 are
equally important. As Abraham Lincoln once said "If I had eight hours
to chop down a tree, I'd spend six hours sharpening my axe." And to
get to this stage, you need to do Step 1 which boils down to one thing
- It's all in the mind. That's why it's so hard.
```

See `:help formatting` and `:help formatoptions` for more details.

Also, similar to `:center`, there are `:left` and `:right` commands to left-align and right-align the text respectively.

11.5 Search and replace

We have seen how to search for text. What if we wanted to 'search and replace'? Then, use the `:s` command.

For example, say you have the text:

```
Setp 4 is eventually what gets you moving, but Setps 2 and 3 are equally
important. As Abraham Lincoln once said "If I had eight hours to chop down a
tree, I'd spend six hours sharpening my axe." And to get to this stage, you need
to do Setp 1 which boils down to one thing - It's all in the mind. That's why
it's so hard.
```

We want to replace all the spelling mistakes of `setp` to `step`. So, just run:

```
:s/setp/step/g
```

This command should be read as follows:

```
:s/pattern/replacement text/options
```

We have already seen how patterns can be as complex as we need it. The replacement patterns can also have special syntax to make use of the original search pattern. For example, if we want to swap two words, we can use:

```
:s/(bachchan) (amitabh)/2 1/g
```

This converts the text from `bachchan amitabh` to `amitabh bachchan`.

The options can specify how the replacement should work. By default, the search and replace works only on the first occurrence of the search pattern in a line. To make it work on all occurrences, we give the `g` option which means 'g'lobal. If we want a confirmation of each change, then specify the `c` option which means confirm every substitution.

11.6 Abbreviations

Sometimes you tend to write the same text over and over again. So why not use shortcuts? They're called abbreviations in Vim.

For example, if you repeatedly write the text `Highly Amazing Corporation Pvt. Ltd.`, then you can run:

```
:iab hac Highly Amazing Corporation Pvt. Ltd.
```

Now, when you are writing the text and enter `h`, `a`, `c`, `<space>`, it will be automatically expanded to the above text!

Run `:verbose abbreviate` to see the list of abbreviations currently set.

See `:help:ab` and `:help:unab` for details.

11.7 Spell checking

An important feature added in the latest version 7 of Vim is spell checking. This lets Vim look for spelling mistakes in your text and help you correct them.

More accurately, Vim looks for words that are present in a "good words list", i.e., a spell file and then flags the remaining words as possible mistakes.

Let's start with the following text:

```
Setp 4 is eventually what gets you moving, but Setps 2 and 3 are equally
important. As Abraham Lincoln once said "If I had eight hours to chop down a
tree, I'd spend six hours sharpening my axe." And to get to this stage, you need
to do Setp 1 which boils down to one thing - It's all in the mind. That's why
it's so hard.
```

To enable spell checking, run:

```
:setlocal spell spelllang=en_us
```

Here 'en' stands for 'English' and 'us' stands for USA. You can choose the language and locale of your choice depending on the text. However, the corresponding spell files must be installed in the `$VIMRUNTIME/spell/` directory. If not, Vim will prompt you on whether it should automatically download the spell file from the Vim website.

Vim should now display a red squiggly line below the "Setp" and similar mistakes. The exact color depends on your `colorscheme` setting.

Press `]s` to move to the next 'bad word', i.e., incorrect spelling.

Now press `z=` to ask Vim for suggestions on good words. It'll display a list of choices. You can type the number for the choice that you think is correct and

press enter to replace the word with the selected choice, or simply press enter to cancel.

If you want to see a score for each word on how 'good' a replacement choice the word is, run `:set verbose=1` and then run `z=`.

In our case, we can replace "Setp" with "Step", but we have the same word again. It would be better if we can make this substitution everywhere in the text. For that, we can run `:spellrepall`.

Consider the text:

```
Swaroop is a name.
```

In this case, Vim flags that 'Swaroop' is a "bad word". We know that it is a name and for our purposes, we want to add it to the good word list so that Vim doesn't mark this word everytime. For this, we can run `zg` to add to the 'g'ood words list.

To know more information about which spell file is being used, you can run `:spellinfo`.

I find spell-checking to be useful when I can toggle it on and off so that it doesn't hamper my normal editing. So, I've added the following lines to my *vimrc file* so that I can use F4 to do the toggling:

```
" Spell check
function! ToggleSpell()
    if !exists("b:spell")
        setlocal spell spelllang=en_us
        let b:spell = 1
    else
        setlocal nospell
        unlet b:spell
    endif
endfunction

nmap <F4> :call ToggleSpell()<CR>
imap <F4> <Esc>:call ToggleSpell()<CR>a
```

Spell-checking is a huge topic on its own, so if you're interested in how spell checking is implemented in Vim, and also about how you can add spellings or word lists for your language of choice, please see `:help spell`.

11.8 Rectangular selection

When we are editing tabular data, sometimes we would want to copy only a few columns from the text as opposed to a few lines. For this, we can use the rectangular block selection mode in Vim by pressing `ctrl-v`.

Consider the following sample text:

```
1. Concentrate
2. Understand
3. Think
4. Get Things Done
```

1. Place the cursor on the capital 'C' in the first line.
2. Press `ctrl-v`.
3. Press `3j` to travel down 3 lines.
4. Press `$` to move right towards the end of the line.
5. Press `y` to yank the text to the default register.
6. Run `:new` and press `p` to paste the rectangular selection.

The new file should have the following contents:

```
Concentrate
Understand
Think
Get Things Done
```

See `:help ctrl-v` for details.

11.9 Remote file editing

You can directly edit a file in a remote ftp site using Vim. Just run `vim ftp://ftp.foo.com/bar` command or run `:Nread ftp://ftp.foo.com/bar` in Vim.

This makes use of the built-in "netrw" plugin in Vim using which you can also edit remote files via scp, http, webdav and other protocols. See `:help netrw-urls` for details.

You can even provide the username and password in your `~/.netrc` file so that Vim can automatically login for you.

See `:help netrw-start` for more details.

11.10 Summary

We have dived a little deeper into the range of editing features that Vim provides. This should give you an idea of the wide range of things that you can do. Again, the important thing is not to know every feature but to learn what is important to you right now and make it a habit, and then learn the other features/options/plugins as and when required.

It might be a good idea to browse through the "Editing Effectively" section of `:help user-manual` and read any topics that you find interesting.

Chapter 12

Vim Multiplicity

12.1 Introduction

In this chapter, let's explore how Vim helps us work between different parts of the file, different files, different 'windows' and different tabs to help us to handle more simultaneously. After all, an important part about good editing is managing files.

12.2 Multiple Sections using Folds

If you're editing a long document, wouldn't it be easier if you can "close" all sections of the document and focus on only one at a time?

This is what we call "folding" in Vim.

Let us take the example where your document is structured such that each level of the text is indented one level higher such as the following piece of text:

```
Book I

    The Shadow of the Past

        Three Rings for the Elven-kings under the sky,
        Seven for the Dwarf-lords in their halls of stone,
        Nine for Mortal Men doomed to die,
        One for the Dark Lord on his dark throne
        In the Land of Mordor where the Shadows lie.
        One Ring to rule them all, One Ring to find them,
        One Ring to bring them all and in the darkness bind them
        In the Land of Mordor where the Shadows lie.

    Three is Company
```

```
The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with weary feet,
Until it joins some larger way,
Where many paths and errands meet.
And whither then? I cannot say.
```

Note:

This text was retrieved from *WikiQuote*

After writing this text, run `:set foldmethod=indent`, position your cursor on the text you want to indent, press `zc` and see how the text folds up. Press `zo` to open the fold.

Personally, I prefer a keyboard shortcut of using the spacebar for both opening and closing folds. To do that, add this to your *vimrc file*:

Basic commands are `zo` and `zc` where we can open and close the fold respectively. You can use `za` to 'a'lternate between opening and closing a fold respectively. You can make it even easier by using the space bar in normal mode to open and close a fold:

```
:nnoremap <space> za
```

Folding is a huge topic on its own with more ways of folding (manual, marker, expression) and various ways of opening and closing hierarchies of folds, and so on. See `:help folding` for details.

12.3 Multiple Buffers

Suppose you wanted to edit more than one file at a time using the same Vim, what do you do?

Remember that files are loaded into buffers in Vim. Vim can also load multiple buffers at the same time. So, you can have multiple files open at the same time and you can keep switching between them.

Let's say you have two files, `part1.txt` and `part2.txt`:

part1.txt

```
I have coined a phrase for myself - 'CUT to the G':

1. Concentrate
2. Understand
3. Think
4. Get Things Done
```

part2.txt

```
Step 4 is eventually what gets you moving, but Steps 2 and 3 are equally
important. As Abraham Lincoln once said "If I had eight hours to chop down a
tree, I'd spend six hours sharpening my axe." And to get to this stage, you need
to do Step 1 which boils down to one thing - It's all in the mind. That's why
it's so hard.
```

Now, run `:e part1.txt` and then run `:e part2.txt`. Observe that you have the second file now open for editing. How do you switch back to the first file? In this particular case, you can just run `:b 1` to switch to 'b'uffer number '1'. You can also run `:e part1.txt` to open the existing buffer into view.

You can see what buffers have been loaded and correspondingly, which files are being edited by running `:buffers` or a shorter form, `:ls` which stands for 'l'i's't buffers.

Buffers will be automatically removed when you close Vim, so you don't have to do anything special other than making sure you save your files. However, if you really want to remove a buffer, for example in order to use less memory, then you can use `:bd 1` to 'd'elete the 'b'uffer numbered '1', etc.

See `:help buffer-list` on the numerous things you can do with buffers.

12.4 Multiple Windows

We have seen how to edit multiple files at the same time, but what if we wanted to view two different files simultaneously. For example, you want to have two different chapters of your book open so that you can write the second chapter consistent with the wordings/description given in the first chapter. Or you want to copy/paste some stuff from the first file to the second file.

In the last section, we used the same "view" to edit multiple buffers. Vim calls these "views" as windows. This term "window" should *not* be confused with your desktop application window which usually means the entire application itself. Just think of 'windows' as 'views' on different files.

Let's take the same `part1.txt` and `part2.txt` sample files used in the previous section.

First, load the part1.txt using `:e part1.txt`. Now, let's open a new buffer by splitting the window to create a new window – run `:new`. You should now be able to do any normal editing in the new buffer in the new window, except that you can't save the text because you haven't associated a file name with the buffer. For that, you can use `:w test.txt` to save the buffer.

How do you switch between these two windows? Just use `ctrl-w` to switch between the windows. Motion keys can be one of h, j, k, l or even any of the arrow keys (in this example, only up and down keys make sense). Remember that `ctrl-w` operations work on 'w'indows.

As a quick shortcut, you can press ctrl-w twice, i.e., `ctrl-w ctrl-w` to cycle between all the open windows.

A particular situation where multiple windows are useful is when you want to view two different parts of the same file simultaneously. Just run `:sp` to create a 'sp'lit window and then you can scroll each window to a different position and continue your editing. Since they both are "windows" to the same buffer, changes in one window will be immediately reflected in the other window. You can also use `ctrl-w s` instead of `:sp`.

To create a vertical split, use `:vsp` or `ctrl-w v`. To close a window, just run `:q` as usual.

Now that we have seen how to open and use multiple windows, let's see how to further play around with the display.

- Suppose you have two split windows but want to reverse the windows, so that you can focus your eyes on the bottom part or top part of your computer screen, as per your preference? Press `ctrl-w r` to 'r'otate the windows.
- Want to move the current window to the topmost position? Press `ctrl-w K`.
- Want to resize a window to make it smaller or larger? Run `:resize 10` to make it 10 lines long display, etc.
- Want to make the current window as big as possible so that you can concentrate on it? Press `ctrl-w _`. Think of the underscore as an indication that the other windows should be as small as possible.
- Want to make the windows 'equal' in height again? Press `ctrl-w =`.

See `:help windows` on more details on what you can do with windows.

12.5 Multiple Tabs

If you use Firefox, you might have used the tabs feature which allows you to open multiple websites in a single Firefox window so that you can switch between them without having the headache of switching between multiple windows. Well, tabs work exactly the same way in Vim also. Except that they are called "tab pages."

```
" Shortcuts for moving between tabs.
" Alt-j to move to the tab to the left
noremap <A-j> gT
" Alt-k to move to the tab to the right
noremap <A-k> gt
```

To 'c'lose a 'tab', run `:tabc` or `:q`.

You can even open text that opens in a new window to open in a new tab instead. For example, `:help tabpage` opens the help in a horizontally split window. To view it in a new tab instead, use `:tab help tabpage`.

If you want to reorder the tabs, use `:tabmove`. For example, to move the current tab to the first position, use `:tabmove 0` and so on.

See `:help tabpage` for more details on tab pages and the other operations that you can run, such as `:tabdo` to operate on each of the tab pages which are open, and customizing the title of the tab pages (`:help setting-guitablabel`), etc.

12.6 Summary

Vim provides a number of ways to edit multiple files at the same time – buffers, windows and tabs. Using these features depends on your personal habit. For example, using multiple tabs may obviate the usage of multiple windows. It's important to use the one which is most convenient and comfortable.

Chapter 13

Vim Personal Information Management

13.1 Introduction

A chapter on 'personal information management' (PIM) in a book on an editor software seems strange, doesn't it? Well, there are lots of "professional software" that claim to do personal information management, so let us explore why can't we use a plain text editor like Vim for this purpose?

Personal information management is about organizing all your "information" – such as your todo lists, diary entries, your reference material (such as important phone numbers), scratchpad and so on. Putting all of this in one convenient location can be extremely handy, and we will explore this using Vim and a few plugins.

I tend to think that a PIM system is best organized as a wiki. A wiki is a quick way to link together various documents which are inter-related but are independent in their own right. Unsurprisingly, the word 'wiki' means 'quick' in the Hawaiian language. Think of a website – there is a home page, and there are related pages to which you see links, and each page will have its own content but can also inter-link to other pages. Isn't this an easy way of organizing websites? What if you could do the same for your own personal information? See this *LifeHack article titled 'Wikify Your Life: How to Organize Everything'* on some great examples on what you can do.

But does this really require a specialized Wiki software? What if you could do the same in just plain text files using Vim? Let's dive in.

13.2 Installing Viki

Note:

> The $vimfiles directory corresponds to `~/.vim` on Linux/Mac, `C:/Documents and Settings//vimfiles` on Windows and `C:Users//vimfiles` on Windows Vista. See `:help vimfiles` for specific details.

We're going to install Viki and its related plugins:

1. Download *multvals.vim* and store as `$vimfiles/plugin/multvals.vim`
2. Download *genutils.zip* and unzip this file to `$vimfiles`
3. Download *tlib.vba.gz*, open it with vim and run `:so%`
4. Download *Viki.vba* Open your vimrc file and set the following per the instructions for Viki.vba (for more information on vimrc use:help vimrc-intro)

   set nocompatible filetype plugin indent on syntax on

Open viki.vba with with vim and run `:so %`

13.3 Get Started

1. Open the GUI version of Vim
2. `:e test.txt`
3. `:set filetype=viki`
4. Type the following text: `[[http://deplate.sourceforge.net/Markup.html][Viki syntax]]`
5. `:w`
6. Position your cursor on the above text and press ctrl%2Benter, or alternatively press `vf`
7. You should see a web browser open up with the above website page open

Similarly, you can write down any file name (with a valid path) – whether it is a `.doc` file or a `.pdf` file and then you can ctrl%2Benter to open the file in the corresponding Word or Acrobat Reader programs!

The idea is that you can use plain text files to *hold* all your thinking together and you can ctrl%2Benter your way into everything else.

Now, notice that we had to type the square brackets in pairs above to identify the target of the link and the words that describe the link. This is basically the syntax of the markup language which we will explore next.

13.4 Markup language

The *Viki syntax* page (that you just opened in your web browser) explains how to write the text to allow Viki to syntax highlight portions of your text as

well as how to do the linking between 'wiki pages' and even write Viki-specific comments.

Learning the basics of the syntax highlighting is useful because you can visually see the parts of your text file. For example, use `* List of things to do` to make it a header, and then use dashes to create a list:

```
* List of things to do

 - Finish the blog post on Brahmagiri trek
 - Fix footer bug on IONLAB website
 - Buy some blank CDs
 - Get motorbike serviced
```

13.4.1 Disabling CamelCase

Writing CamelCase can create a wiki link in Viki, but I personally dislike this. I prefer that only explicit links like `_[CamelCase]]` be allowed to avoid situations where I have genuinely used a name which uses camel case but I don't want it to be a link (for example, the word "JavaScript"). To disable camel-case syntax, you will have to add the following line to the *vimrc file* (which is explained in the Plugins chapter):

```
let g:vikiNameTypes = "sSeuix"
```

13.5 Getting Things Done

One of the major reasons for creating this 'wiki' for myself is to maintain a 'Getting Things Done' system.

Getting Things Done ("GTD") is a system devised by David Allen to help manage your 'stuff' – which could mean anything from your career plans to the list of chores you have to do today. *A good introduction to GTD can be found on bnet.com.*

From David Allen's book:

> *"Get everything out of your head. Make decisions about actions required on stuff when it shows up – not when it blows up. Organize reminders of your projects and the next actions on them in appropriate categories. Keep your system current, complete, and reviewed sufficiently to trust your intuitive choices about what you're doing (and not doing) at any time."*

The GTD system basically consists of organizing your information into certain pages/folders:

1. Collection Basket

2. Projects List
3. Next Actions
4. Calendar
5. Someday/Maybe
6. Reference Material
7. Waiting For

I created a wiki to match this system by using the following method:

1. First, create a `StartPage` which is literally the start page to your personal organization system (hereby referred to as simply "your viki").
2. Then, create a list of main sections of your wiki:

```
* Getting Things Done

 1. [[Collect][In Basket]]
 2. [[Project][Projects List]]
 3. [[NextActions][Next Actions]]
 4. [[Calendar]]
 5. [[SomedayMaybe][Someday/Maybe]]
 6. [[Reference][Reference Material]]
 7. [[Waiting][Waiting For]]
```

3. Similarly, go to as much depth as you want, for example creating a `[[Reference.Career]]` to jot down your career plans, and `[[Project.TopSecret]]` to gather thoughts on your next project, and so on.
4. Every time you want to jot down something, use the `[[Collect]]` page and then process, organize, review and finally actually `do` your next-physical-actions.
5. It takes a while to get accustomed to using this system, but once you are comfortable, you can achieve clarity of mind, confidence that you're taking care of all the factors in your life, and most importantly, a sense of direction in knowing what are the important things in your life.

Notice how we are managing an entire system using just plain text!

13.6 Summary

We have just explored how Vim can help you in creating a personal information management system for yourself. It's fascinating how we don't need a complicated software for such a system, just plain text files and Vim will do.

See *Abhijit Nadgouda's article on using Vim as a personal wiki* for an alternative way of achieving the same using built-in Vim functionality.

Chapter 14

Vim Scripting

14.1 Introduction

If you want to customize any software, most likely you will change the various settings in the software to suit your taste and needs. What if you wanted to do more than that? For example, to check for conditions such as `if GUI version, then use this colorscheme else use this colorscheme`? This is where "scripting" comes in. Scripting basically means using a language where you can specify conditions and actions put together into 'scripts'.

There are two approaches to scripting in Vim – using the built-in Vim scripting language, or using a full-fledged programming language such as Python or Perl which have access to the Vim internals via modules (provided that Vim has been compiled with these options enabled).

This chapter requires some knowledge of programming. If you have no prior programming experience, you will still be able to understand although it might seem terse. If you wish to learn programming, please refer my other free book *A Byte of Python*.

There are two ways of creating reusable functionality in Vim – using macros and writing scripts.

14.2 Macros

Using macros, we can record sequences of commands and then replay it in different contexts.

For example, suppose you had some text like this:

```
tansen is the singer
daswant is the painter
todarmal is the financial wizard
abul fazl is the historian
birbal is the wazir
```

There are many things to correct here.

1. Change the first letter of the sentence to upper case.
2. Change 'is' to 'was'.
3. Change 'the' to 'a'.
4. End the sentence with "in Akbar's court."

One way would be to use a series of substitute commands, such as `:s/^w/u/` but this would require 4 substitution commands and it might not be simple if the substitute command changes parts of the text which we do not want to be changed.

A better way would be to use macros.

1. Position your cursor on the first letter of the first line: `tansen is the singer`
2. Type `qa` in normal mode to start recording the macro named as `a`.
3. Type `gUl` to switch the first letter to upper case.
4. Type `w` to move to the next word.
5. Type `cw` to change the word.
6. Type `was`.
7. Press `<Esc>`.
8. Type `w` to move to the next word.
9. Type `cw` to change the word.
10. Type `a`.
11. Press `<Esc>`.
12. Type `A` to insert text at the end of the line.
13. Type `in Akbar's court.`
14. Press `<Esc>`.
15. Type `q` to stop recording the macro.

That looks like a long procedure, but sometimes, this is much easier to do than cook up some complicated substitute commands!

At the end of the procedure, the line should look like this:

```
Tansen was a singer in Akbar's court.
```

Great. Now, let's move on to apply this to the other lines. Just move to the first character of the second line and press `@a`. Voila, the line should change to:

```
Daswant was a painter in Akbar's court.
```

This demonstrates that macros can record complicated operations and can be easily replayed. This helps the user to replay complicated editing in multiple places. This is one type of reusing the manipulations you can do to the text. Next, we will see more formal ways of manipulating the text.

> Note:
>
> If you want to simply repeat the last action and not a sequence of actions, you do not have to use macros, just press `.` (dot).

14.3 Basics of Scripting

Vim has a built-in scripting language using which you can write your own scripts to take decisions, "do" stuff, and manipulate the text.

14.3.1 Actions

How do you change the theme, i.e., colors used by Vim? Just run:

```
:colorscheme desert
```

Here, I am using the 'desert' color scheme, which happens to be my favorite. You can view the other schemes available by typing `:colorscheme` and then pressing " key to cycle through the available schemes.

What if you wanted to know how many characters are in the current line?

```
:echo strlen(getline("."))
```

Notice the names 'strlen' and 'getline'. These are "functions". *Functions* are pieces of scripts already written and have been given a name so that we can use them again and again. For example, the `getline` function fetches a line and we are indicating which line by the `.` (dot) which means the current line. We are passing the result returned by the getline function to the `strlen` function which counts the number of characters in the text and then we are passing the result returned by the strlen function to the `:echo` command which simply displays the result. Notice how the information flows in this command.

The `strlen(getline("."))` is called an expression. We can store the results of such expressions by using variables. *Variables* do what the name suggests – they are names pointing to values and the value can be anything, i.e., it can vary. For example, we can store the length as:

```
:let len = strlen(getline("."))
:echo "We have" len "characters in this line."
```

When you run this line on the second line above in this text, you will get the following output:

```
We have 46 characters in this line.
```

Notice how we can use variables in other 'expressions'. The possibilities you can achieve with the help of these variables, expressions and commands are endless.

Vim has many types of variables available via prefixes such as `$` for environment variables, `&` for options, and `@` for registers:

```
:echo $HOME
:echo &amp;filetype
:echo @a
```

See `:help function-list` for a huge list of functions available.

You can create your own functions as well:

```
:function CurrentLineLength()
:    let len = strlen(getline("."))
:    return len
:endfunction
```

Now position your cursor on any line and run the following command:

```
:echo CurrentLineLength()
```

You should see a number printed.

Function names have to start with an upper case. This is to differentiate that built-in functions start with a lower case and user-defined functions start with an upper case.

If you want to simply "call" a function to run but not display the contents, you can use `:call CurrentLineLength()`

14.3.2 Decisions

Suppose you want to display a different color schemes based on whether Vim is running in the terminal or is running as a GUI, i.e., you need the script to take decisions. Then, you can use:

```
:if has("gui_running")
:    colorscheme desert
:else
:    colorscheme darkblue
:endif
```

How it works:

- `has()` is a function which is used to determine if a specified feature is supported in Vim installed on the current computer. See `:help feature-list` to see what kind of features are available in Vim.
- The `if` statement checks the given condition. If the condition is satisfied, we take certain actions. "Else", we take the alternative action.
- Note that an `if` statement should have a matching `endif`.
- There is `elseif` also available, if you want to chain together multiple conditions and actions.

The looping statements 'for' and 'while' are also available in Vim:

```
:let i = 0
:while i < 5
:    echo i
:    let i += 1
:endwhile
```

Output:

```
0
1
2
3
4
```

Using Vim's built-in functions, the same can also be written as:

```
:for i in range(5)
:    echo i
:endfor
```

- `range()` is a built-in function used to generate a range of numbers. See `:help range()` for details.
- The `continue` and `break` statements are also available.

14.3.3 Data Structures

Vim scripting also has support for lists and dictionaries. Using these, you can build up complicated data structures and programs.

```
:let fruits = ['apple', 'mango', 'coconut']

:echo fruits[0]
" apple

:echo len(fruits)
" 3

:call remove(fruits, 0)
:echo fruits
" ['mango', 'coconut']

:call sort(fruits)
:echo fruits
" ['coconut', 'mango']
```

```
:for fruit in fruits
:   echo "I like" fruit
:endfor
" I like coconut
" I like mango
```

There are many functions available – see 'List manipulation' and 'Dictionary manipulation' sections in `:help function-list`.

14.4 Writing a Vim script

We will now write a Vim script that can be loaded into Vim and then we can call its functionality whenever required. This is different from writing the script inline and running immediately as we have done all along.

Let us tackle a simple problem – how about capitalizing the first letter of each word in a selected range of lines? The use case is simple – When I write headings in a text document, they look better if they are capitalized, but I'm too lazy to do it myself. So, I can write the text in lower case, and then simply call the function to capitalize.

We will start with the basic template script. Save the following script as the file `capitalize.vim`:

```
" Vim global plugin for capitalizing first letter of each word
"       in the current line.
" Last Change: 2008-11-21 Fri 08:23 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php

if exists("loaded_capitalize")
    finish
endif
let loaded_capitalize = 1

" TODO: The real functionality goes in here.
```

How It Works:

- The first line of the file should be a comment explaining what the file is about.
- There are 2-3 standard headers mentioned regarding the file such as 'Last Changed:' which explains how old the script is, the 'Maintainer:' info so that users of the script can contact the maintainer of the script regarding any problems or maybe even a note of thanks.
- The 'License:' header is optional, but highly recommended. A Vim script or plugin that you write may be useful for many other people as well, so you can specify a license for the script. Consequently, other people can improve your work and that it will in turn benefit you as well.

- A script may be loaded multiple times. For example, if you open two different files in the same Vim instance and both of them are `.html` files, then Vim opens the HTML syntax highlighting script for both of the files. To avoid running the same script twice and redefining things twice, we use a safeguard by checking for existence of the name 'loaded_capitalize' and closing if the script has been already loaded.

Now, let us proceed to write the actual functionality.

We can define a function to perform the transformation – capitalize the first letter of each word, so we can call the function as `Capitalize()`. Since the function is going to work on a range, we can specify that the function works on a range of lines.

```
" Vim global plugin for capitalizing first letter of each word
"       in the current line
" Last Change: 2008-11-21 Fri 08:23 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php

" Make sure we run only once
if exists("loaded_capitalize")
    finish
endif
let loaded_capitalize = 1

" Capitalize the first letter of each word
function Capitalize() range
    for line_number in range(a:firstline, a:lastline)
        let line_content = getline(line_number)
        " Luckily, the Vim manual had the solution already!
        " Refer ":help s%" and see 'Examples' section
       let line_content = substitute(line_content, "w+", "u0", "g")
        call setline(line_number, line_content)
    endfor
endfunction
```

How It Works:

- The `a:firstline` and `a:lastline` represent the arguments to the function with correspond to the start and end of the range of lines respectively.
- We use a 'for' loop to process each line (fetched using `getline()`) in the range.
- We use the `substitute()` function to perform a regular expression search-and-replace on the string. Here we are specifying the function to look for words which is indicated by 'w+' which means a word (i.e., a continuous set of characters that are part of words). Once such words are found, they are to be converted using `u` – the `u` indicates that the first character following this sequence should be converted to upper case. The " indicates the

match found by the substitute() function which corresponds to the words. In effect, we are converting the first letter of each word to upper case.
- We call the `setline()` function to replace the line in Vim with the modified string.

To run this command:

1. Open Vim and enter some random text such as 'this is a test'.
2. Run `:source capitalize.vim` – this 'sources' the file as if the commands were run in Vim inline as we have done before.
3. Run `:call Capitalize()`.
4. The line should now read 'This Is A Test'.

Running `:call Capitalize()` every time appears to be tedious, so we can assign a keyboard shortcut using leaders:

```
" Vim global plugin for capitalizing first letter of each word
" in the current line
" Last Change: 2008-11-21 Fri 08:23 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php

" Make sure we run only once
if exists("loaded_capitalize")
    finish
endif
let loaded_capitalize = 1

" Refer ':help using-<Plug>'
if !hasmapto('<Plug>Capitalize')
    map <unique> <Leader>c <Plug>Capitalize
endif
noremap <unique> <script> <Plug>Capitalize <SID>Capitalize
noremap <SID>Capitalize :call <SID>Capitalize()<CR>

" Capitalize the first letter of each word
function s:Capitalize() range
    for line_number in range(a:firstline, a:lastline)
        let line_content = getline(line_number)
        " Luckily, the Vim manual had the solution already!
        " Refer ":help s%" and see 'Examples' section
       let line_content = substitute(line_content, "\w+", "\u0", "g")
        call setline(line_number, line_content)
    endfor
endfunction
```

- We have changed the name of the function from simply 'Capitalize' to 's:Capitalize' – this is to indicate that the function is local to the script that it is defined in, and it shouldn't be available globally because we want to avoid interfering with other scripts.

- We use the `map` command to define a shortcut.
- The “ key is usually backslash.
- We are now mapping `c` (i.e., the leader key followed by the ‘c’ key) to some functionality.
- We are using `Capitalize` to indicate the `Capitalize()` function described within a plugin, i.e., our own script.
- Every script has an ID, which is indicated by ‘`. So, we can map the command`Capitalize`to a call of the local function`Capitalize()‘.

So, now repeat the same steps mentioned previously to test this script, but you can now run `c` to capitalize the line(s) instead of running `:call Capitalize()`.

This last set of steps may seem complicated, but it just goes to show that there's a wealth of possibilities in creating Vim scripts and they can do complex stuff.

If something does go wrong in your scripting, you can use `v:errmsg` to see the last error message which may give you clues to figure out what went wrong.

Note that you can use `:help` to find help on everything we have discussed above, from `:help w` to `:help setline()`.

14.5 Using external programming languages

Many people would not like to spend the time in learning Vim's scripting language and may prefer to use a programming language they already know and write plugins for Vim in that language. This is possible because Vim supports writing plugins in Python, Perl, Ruby and many other languages.

In this chapter, we will look at a simple plugin using the Python programming language, but we can easily use any other supported language as well.

As mentioned earlier, if you are interested in learning the Python language, you might be interested in my other free book *A Byte of Python*

First, we have to test if the support for the Python programming language is present.

```
:echo has("python")
```

If this returns 1, then we are good to go, otherwise you might want to install Python on your machine and try again.

Suppose you are writing a blog post. A blogger usually wants to get as many people to read his/her blog as possible. One of the ways people find such blog posts is by querying a search engine. So, if you're going to write on a topic (say ‘C V Raman’, the famous Indian physicist who has won a Nobel Prize for his work on the scattering of light), you might want to use important phrases that helps more people find your blog when they search for this topic. For example, if people are searching for ‘c v raman’, they might also search for the ‘raman effect’, so you may want to mention that in your blog post or article.

How do we find such related phrases? It turns out that the solution is quite simple, thanks to Yahoo! Search.

First, let us figure out how to use Python to access the current text, which we will use to generate the related phrases.

```
" Vim plugin for looking up popular search queries related
"       to the current line
" Last Updated: 2008-11-21 Fri 08:36 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php

" Make sure we run only once
if exists("loaded_related")
    finish
endif
let loaded_related = 1

" Look up Yahoo Search and show results to the user
function Related()
    python << EOF

    import vim

    print 'Length of the current line is', len(vim.current.line)

    EOF
endfunction
```

The main approach to writing plugins in interfaced languages is same as that for the built-in scripting language.

The key difference is that we have to pass on the code that we have written in the Python language to the Python interpreter. This is achieved by using the EOF as shown above – all the text from the `python <<EOF command` to the subsequent EOF is passed to the Python interpreter.

You can test this program, by opening Vim again separately and running `:source related.vim`, and then run `:call Related()`. This should display something like `Length of the current line is 54`.

Now, let us get down the actual functionality of the program. Yahoo! Search has something called a *RelatedSuggestion* query which we can access using a web service. This web service can be accessed by using a Python API provided by Yahoo! *Search pYsearch*:

```
" Vim plugin for looking up popular search queries related
" to the current line
" Last Updated: 2008-11-21 Fri 08:36 AM IST
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php
```

```
" Make sure we run only once
if exists("loaded_related")
    finish
endif
let loaded_related = 1

" Look up Yahoo Search and show results to the user
function Related()
    python >> EOF

    import vim
    from yahoo.search.web import RelatedSuggestion

   search = RelatedSuggestion(app_id='vimsearch', query=vim.current.line)
    results = search.parse_results()

    msg = 'Related popular searches are:n'
    for i, result in enumerate(results):
        msg += '%d.%sn' % (i + 1, result)
        print msg

    EOF
endfunction
```

Notice that we use the current line in Vim as the current text we are interested in, you can change this to any text that you want, such as the current word, etc.

We use the yahoo.search.web.RelatedSuggestion class to query Yahoo! Search for phrases related to the query that we specify. We get back the results by calling `parse_results()` on the result object. We then loop over the results and display it to the user.

```
1. Run :source related.vim
2. Type the text c v raman.
3. Run :call Related()
4. The output should look something like this:

    Related popular searches are:
    1. raman effect
    2. c v raman india
    3. raman research institute
    4. chandrasekhara venkata raman
```

14.6 Summary

We have explored scripting using the Vim's own scripting language as well as using external scripting/programming languages. This is important because the

functionality of Vim can be extended in infnite ways.

See `:help eval`, `:help python-commands`, `help perl-using` and `:help ruby-commands` for details.

Chapter 15

Vim Plugins

15.1 Introduction

As we have seen in the *previous chapter*, we can write scripts to extend the existing functionality of Vim to do more stuff. We call these scripts which extend or add functionality as "plugins."

There are various kinds of plugins that can be written:

- vimrc
- global plugin
- filetype plugin
- syntax highlighting plugin
- compiler plugin

Not only can you write your own plugins but also *download and use plugins written by others*.

15.2 Customization using vimrc

When I install a new Linux distribution or reinstall Windows, the first thing I do after installing Vim is fetch my latest `vimrc` file from my backups, and then start using Vim. Why is this important? Because the `vimrc` file contains various customizations/settings I like which makes Vim more useful and comfortable for me.

There are two files you can create to customize Vim to your taste:

1. vimrc – for general customizations
2. gvimrc – for GUI specific customizations

These are stored as:

- `%HOME%/_vimrc` and `%HOME%/_gvimrc` on Windows
- `$HOME/.vimrc` and `$HOME/.gvimrc` on Linux/BSD/Mac OS X

See `:help vimrc` on the exact location on your system.

These vimrc and gvimrc files can contain any Vim commands. The convention followed is to use only simple settings in the vimrc files, and advanced stuff are sourced from plugins.

For example, here's a portion of my vimrc file:

```
" My Vimrc file
" Maintainer: www.swaroopch.com/contact/
" Reference: Initially based on
"   http://dev.gentoo.org/~ciaranm/docs/vim-guide/
" License: www.opensource.org/licenses/bsd-license.php

" Enable syntax highlighting.
syntax on

" Automatically indent when adding a curly bracket, etc.
set smartindent

" Tabs should be converted to a group of 4 spaces.
" This is the official Python convention
" (http://www.python.org/dev/peps/pep-0008/)
" I didn't find a good reason to not use it everywhere.
set shiftwidth=4
set tabstop=4
set expandtab
set smarttab

" Minimal number of screen lines to keep above and below the cursor.
set scrolloff=999

" Use UTF-8.
set encoding=utf-8

" Set color scheme that I like.
if has("gui_running")
    colorscheme desert
else
    colorscheme darkblue
endif

" Status line
set laststatus=2
set statusline=
set statusline+=%-3.3n                      " buffer number
set statusline+=%f                          " filename
```

```
set statusline+=%h%m%r%w                     " status flags
set statusline+=[%{strlen(&ft)?&ft:'none'}] " file type
set statusline+=%=                          " right align remainder
set statusline+=0x%-8B                       " character value
set statusline+=%-14(%l,%c%V%)               " line, character
set statusline+=%<%P                         " file position

" Show line number, cursor position.
set ruler

" Display incomplete commands.
set showcmd

" To insert timestamp, press F3.
nmap <F3> a<C-R>=strftime("%Y-%m-%d%a%I:%M%p")<CR><Esc>
imap <F3> <C-R>=strftime("%Y-%m-%d%a%I:%M%p")<CR>

" To save, press ctrl-s.
nmap <c-s> :w<CR>
imap <c-s> <Esc>:w<CR>a

" Search as you type.
set incsearch

" Ignore case when searching.
set ignorecase

" Show autocomplete menus.
set wildmenu

" Show editing mode
set showmode

" Error bells are displayed visually.
set visualbell
```

Notice that these commands are not prefixed by colon. The colon is optional when writing scripts in files because it assumes they are normal mode commands.

If you want to learn detailed usage of each setting mentioned above, refer `:help`.

A portion of my gvimrc file is:

```
" Size of GVim window
set lines=35 columns=99

" Don't display the menu or toolbar. Just the screen.
set guioptions-=m
set guioptions-=T
```

```
" Font. Very important.
if has('win32') || has('win64')
    " set guifont=Monaco:h16
    " http://jeffmilner.com/index.php/2005/07/30/
    \   windows-vista-fonts-now-available/
    set guifont=Consolas:h12:cANSI
elseif has('unix')
    let &guifont="Monospace 10"
endif
```

There are *various examples vimrc files out there* that you should definitely take a look at and learn the various kinds of customizations that can be done, then pick the ones you like and put it in your own vimrc.

A few good ones that I have found in the past are:

It is a known fact that a person's vimrc usually reflects how long that person has been using Vim.

15.3 Global plugin

Global plugins can be used to provide global/generic functionality.

Global plugins can be stored in two places:

1. `$VIMRUNTIME/plugin/` for standard plugins supplied by Vim during its installation
2. To install your own plugins or plugins that you have downloaded from somewhere, you can use your own plugin directory:

- `$HOME/.vim/plugin/` on Linux/BSD/Mac OS X
- `%HOME%/vimfiles/plugin/` on Windows
- See `:help runtimepath` for details on your plugin directories.

Let's see how to use a plugin.

A useful plugin to have is `highlight_current_line.vim` by Ansuman Mohanty which does exactly as the name suggests. Download the latest version of the file `highlight_current_line.vim` and put it in your plugin directory (as mentioned above). Now, restart Vim and open any file. Notice how the current line is highlighted compared to the other lines in the file.

But what if you don't like it? Just delete the `highlight_current_line.vim` file and restart Vim.

Similarly, we can install our own `related.vim` or `capitalize.vim` from the *Scripting* chapter into our plugin directory, and this avoids us the trouble of having to use `:source` every time. Ultimately, any kind of Vim plugin that you write should end up somewhere in your `.vim/vimfiles` plugin directory.

15.4 Filetype plugin

Filetype plugins are written to work on certain kinds of files. For example, programs written in the C programming language can have their own style of indentation, folding, syntax highlighting and even error display. These plugins are not general purpose, they work for these specific filetypes.

15.4.1 Using a filetype plugin

Let's try a filetype plugin for XML. XML is a declarative language that uses tags to describe the structure of the document itself. For example, if you have a text like this:

```
Iron Gods
---------

Ashok Banker's next book immediately following the Ramayana is said to
be a novel tentatively titled "Iron Gods" scheduled to be published in
2007. A contemporary novel, it is an epic hard science fiction story
about a war between the gods of different faiths. Weary of the
constant infighting between religious sects and their deities, God
(aka Allah, Yahweh, brahman, or whatever one chooses to call the
Supreme Deity) wishes to destroy creation altogether.

A representation of prophets and holy warriors led by Ganesa, the
elephant-headed Hindu deity, randomly picks a sample of mortals, five
of whom are the main protagonists of the book--an American Catholic,
an Indian Hindu, a Pakistani Muslim, a Japanese Buddhist, and a
Japanese Shinto follower. The mortal sampling, called a 'Palimpsest'
is ferried aboard a vast Dyson's Sphere artifact termed The Jewel,
which is built around the sun itself, contains retransplanted cities
and landscapes brought from multiple parallel Earths and is the size
of 12,000 Earths. It is also a spaceship travelling to the end of
creation, where the Palimpsest is to present itself before God to
plead clemency for all creation.

Meanwhile, it is upto the five protagonists, aided by Ganesa and a few
concerned individuals, including Lucifer Morningstar, Ali Abu Tarab,
King David and his son Solomon, and others, to bring about peace among
the myriad warring faiths. The question is whether or not they can do
so before the audience with God, and if they can do so peacefully--for
pressure is mounting to wage one final War of Wars to end all war
itself.

(Excerpt taken from

http://en.wikipedia.org/w/index.php?title=Ashok_Banker&oldid=86219280

under the GNU Free Documentation License)
```

It can be written in XML form (specifically, in 'DocBook XML' format) as:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
        "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">
<article>

    <articleinfo>
      <author><firstname>Wikipedia Contributors</firstname></author>
        <title>Iron Gods</title>
    </articleinfo>

    <para>

      Ashok Banker's next book immediately following the Ramayana is
      said to be a novel tentatively titled "Iron Gods" scheduled to
      be published in 2007. A contemporary novel, it is an epic hard
        science fiction story about a war between the gods of
       different faiths.  Weary of the constant infighting between
        religious sects and their deities, God (aka Allah, Yahweh,
       brahman, or whatever one chooses to call the Supreme Deity)
        wishes to destroy creation altogether.

    </para>

    <para>

       A representation of prophets and holy warriors led by Ganesa,
        the elephant-headed Hindu deity, randomly picks a sample of
         mortals, five of whom are the main protagonists of the
         book--an American Catholic, an Indian Hindu, a Pakistani
        Muslim, a Japanese Buddhist, and a Japanese Shinto follower.
      The mortal sampling, called a 'Palimpsest' is ferried aboard a
       vast Dyson's Sphere artifact termed The Jewel, which is built
         around the sun itself, contains retransplanted cities and
        landscapes brought from multiple parallel Earths and is the
        size of 12,000 Earths.  It is also a spaceship travelling to
       the end of creation, where the Palimpsest is to present itself
         before God to plead clemency for all creation.

    </para>

    <para>

       Meanwhile, it is upto the five protagonists, aided by Ganesa
         and a few concerned individuals, including Lucifer
        Morningstar, Ali Abu Tarab, King David and his son Solomon,
         and others, to bring about peace among the myriad warring
       faiths. The question is whether or not they can do so before
       the audience with God, and if they can do so peacefully--for
       pressure is mounting to wage one final War of Wars to end all
```

```
        war itself.

    </para>

    <sidebar>
        <para>

            (Excerpt taken from
        http://en.wikipedia.org/w/index.php?title=Ashok_Banker&amp;oldid=86219280
            under the GNU Free Documentation License)

        </para>
    </sidebar>

</article>
```

Notice how the structure of the document is more explicit in the XML version. This makes it easier for the tools to convert the XML into other kinds of formats including PDF and print versions. The downside is that writing in XML is more difficult for the human who writes it. So let's see how ftplugins can help a Vim user who writes XML.

1. First, download the *xmledit ftplugin* and put it in your `~/.vim/ftplugin/` directory.
2. Add the following line to your `~/.vimrc`:

   ```
   autocmd BufNewFile,BufRead *.xml source ~/.vim/ftplugin/xml.vim
   ```

(Make sure to specify the correct directory as per your operating system) This enables the xmledit ftplugin everytime you open a file with extension `.xml`. 3. Open Vim and edit a file called `test.xml`. 4. Type ", and see how the xmledit ftplugin adds the closing tag automatically for you. So, your document should now look like:

```
<article></article>
```

6. Now, type another `>` and see how the tag expands so that you can enter more tags. The document should now look like this:

   ```
   <article>

   </article>
   ```

7. You'll notice the cursor is also indented, so that you can write the document in a neatly structured way to reflect the structure of the document.
8. Repeat the process until you have written the full document.

Notice how a special ftplugin for XML makes it much easier for you to write XML documents. This is what ftplugins are usually designed for.

15.4.2 Writing a filetype plugin

Let us try to write our own ftplugin.

In our previous example of using the xmledit.vim ftplugin and writing in the XML format, we saw that we had to write some standard header information at the top for every DocBook XML file (which is the specific format we used). Why not make this automatic in Vim via the use of a ftplugin?

Our xml ftplugin basically needs to add the following information at the top of a 'new' XML file:

```
<?xml version="1.0"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
         "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">
```

So, let's put our new ftplugin, say called 'xmlheader.vim' to work only on 'BufNewFile' event. So, add the following to your `~/.vimrc`:

```
autocmd BufNewFile *.xml source ~/.vim/ftplugin/xmlheader.vim
```

Now, all we need to do in the xmlheader.vim is to set the first and second lines of the file:

```
" Vim plugin to add XML header information to a new XML file
call setline(1, '<?xml version="1.0"?>')
call setline(2, '<!DOCTYPE article
\   PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
\   "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">')
```

So, now restart Vim, ensure that 'test.xml' file doesn't already exist, and run `:e test.xml`. You will see that the header is already filled in!

15.5 Syntax scheme

We just wrote a DocBook XML file in the previous section. It would have been helpful if there was some color-coding for the XML file for valid DocBook tags to make sure we're writing it correctly. It turns out that this is possible if we just run `:set filetype=docbkxml`. What happens is that Vim uses the syntax file located in `$VIMRUNTIME/syntax/docbkxml.vim`.

The syntax file defines how the parts of the file relate to each other. For example, the syntax file for XML defines what tags are and what colors should be given to tag names, etc.

15.5.1 Using a syntax scheme

Let us see a syntax file in action. Download the *mkd.vim][6] script which is a syntax file for the [Markdown syntax*. Markdown is basically a format in which plain text can be written so that the format can be converted to HTML later.

1. Open a new file in Vim called 'test_markdown.txt'.
2. Run `:set syntax=mkd`
3. Type the following text in the file:

```
# Bengaluru

The name **Bangalore** is an anglicised version of the city's name in
the Kannada language, Bengaluru.

> A popular anecdote (although one contradicted by historical
> evidence) recounts that the 11th-century Hoysala king Veera Ballala
> II, while on a hunting expedition, lost his way in the forest. Tired
> and hungry, he came across a poor old woman who served him boiled
> beans. The grateful king named the place _"benda kaal-ooru"_
> (literally, "town of boiled beans"), which was eventually
> colloquialised to "Bengaluru".

***

(This information has been retrieved from
[Wikipedia](http://en.wikipedia.org/wiki/Bangalore) under the GNU Free
Documentation License.)
```

4. Notice how different parts of the file such as the heading and emphasized words are automatically highlighted. This should hopefully make writing in Markdown syntax easier.

15.5.2 Writing a syntax scheme

Let us now try to write a syntax file of our own for the *AmiFormat* text format.

Syntax highlighting basically revolves around two steps – first is to define the kind of text format we are looking for and the second is to describe how it is to be displayed.

For example, suppose we want to find all instances of `any word` to be displayed in bold. First, we need to match such a pattern in our text and then link the name of this pattern to the kind of display needed:

```
:syntax match ourBold /<b>.*</b>/
:highlight default ourBold term=bold cterm=bold gui=bold
```

The first line says that we are creating a new type of syntax based on matching a pattern, with name 'ourBold' and the regex pattern as specified above.

The second line says that we want to highlight the 'ourBold' syntax. The scheme we are specifying is the default scheme which means it can be overridden by the user or by other color schemes. We can specify three different ways of representing 'ourBold' for the three different kinds of displays that Vim can run in – black and white terminals, color terminals and the GUI (graphical version).

Sometimes we want to specify some task in our text as a todo item and we usually write it in caps 'TODO', but if we want to make it stand out further?

```
:syntax keyword ourTodo TODO FIXME XXX
:hi def link ourTodo Todo
```

First, we define that 'ourTodo' consists of keywords – these are plain simple words that need to be highlighted, and we link this pattern group 'ourTodo' to a pre-existing group called 'Todo' in Vim. There are many such pre-existing groups in Vim which have predefined color schemes. It is best to link our syntax styles to the existing groups. See `:help group-name` for a list of the groups available.

Next, we can have specific blocks of code in the group enclosed in

```
 <tt>..</tt>
```

so how do we highlight this?

```
:syn region amiCode excludenl start=/[code]/ end=/[/code]/
:hi def link amiCode Identifier
```

First, we specify that we are defining a region of text that has a start pattern and end pattern (which is very simple in our case), and then link it to the pre-existing 'Identifier' class.

Similarly, we can proceed to add definitions to other parts of the text as defined in the *AmiFormat reference*, and the final script can look something like this:

```
" Vim syntax file for AmiFormat
" Language: AmiFormat
" Version: 1
" Last Change: 2006-12-28 Thu
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php
" Reference: http://orangoo.com/labs/AmiNation/AmiFormat/

"""""""""" Initial Checks """""""""""""""""""""""""""""""""""""""""""""""""

" To be compatible with Vim 5.8. See `:help 44.12`
if version < 600
```

```
    syntax clear
elseif exists("b:current_syntax")
    " Quit when a (custom) syntax file was already loaded
    finish
endif

"""""""""" Patterns """""""""""""""""""""""""""""""""""""""""""""""""""""""""

" Emphasis
syn match amiItalic /<i>.{-}</i>/
syn match amiBold /<b>.{-}</b>/

" Todo
syn keyword amiTodo TODO FIXME XXX

" Headings
syn match amiHeading /^h[1-6].s+.{-}$/

" Lists
syn match amiList /^s**s+/
syn match amiList /^s*d+.s+/

" Classes
syn match amiClass /^s*%(w+).*%/
syn match amiClass /^s*%{.*}.*%/

" Code
syn region amiCode excludenl start=/[code]/ end=/[/code]/

" HTML
syn region amiEscape excludenl start=/[escape]/ end=/[/escape]/

" Link
syn match amiLink /".{-}":(.{-})/

" Image
syn match amiImage /!.{-}(.{-})!/

"""""""""" Highlighting """""""""""""""""""""""""""""""""""""""""""""""""""""

hi def amiItalic term=italic cterm=italic gui=italic
hi def amiBold term=bold cterm=bold gui=bold

hi def link amiHeading Title
hi def link amiTodo Todo
hi def link amiList PreProc
hi def link amiClass Statement
hi def link amiCode Identifier
hi def link amiEscape Comment
hi def link amiLink String
```

```
hi def link amiImage String

""""""""""" Finish """""""""""""""""""""""""""""""""""""""""""""""""""""""""""

" Set syntax name
let b:current_syntax = "amifmt"
```

Now that the script actually works, I've uploaded it to the *Vim scripts section* already as I wrote this! Now anyone in the world can use the AmiFormat syntax highlighting in Vim.

To learn more about syntax highlighting scripts in Vim, refer:

- `:help syntax`
- `:help usr_44.txt`
- `:help group-name`
- `:help pattern-overview`
- `:help mysyntaxfile`
- `:help new-filetype`

> Note:
>
> If you want to redraw the screen in case the syntax file is causing the display to be improper, press `CTRL-L`.

> Note:
>
> You might have already guessed that when we have previously set the `filetype`, Vim in turn automatically sets the `syntax` to the same name also.

15.6 Compiler plugin

Compiler plugins are used for compiling programs written in different languages. It is useful anywhere a transformation is required from a source plain text language to a different format, even if you are writing a plain text file in Markdown want to convert the text to HTML using a transformation program.

Let us explore the use of a compiler plugin for Python.

1. Download the *compiler/python.vim* script and put it in your `~/.vim/compiler/` directory.
2. Put the following line in your `~/.vimrc`:

   ```
   autocmd BufNewFile,BufRead *.py compiler python
   ```

3. Restart Vim and open a Python file, say `test.py` and enter the following program:

```
#!python
print 'Hello World'
```

4. Run `:make` and you should see successful compilation.

5. Let us intentionally introduce an error in the program by changing the spelling of 'print' to 'pritn':

```
pritn 'Hello World'
```

Now run `:make` and notice that the error is displayed and Vim automatically moves the cursor to the error line!

6. Run `:clist` to see the full list of errors.

7. After fixing an error, you can run `:cnext` to move to the next error.

If you open up the `compiler/python.vim` script that we downloaded, you will notice that it is very simple – there are only two variables defined – one is `makeprg` which defines how to 'make' the file, i.e., how to compile it and the second is `errorformat` which defines the form of the error output of the compiler.

I've written a *compiler plugin for Adobe Flex* using the same two variables.

See `:help write-compiler-plugin` and `:help quickfix` for details on how to write your own compiler plugin.

15.7 Homework: Write a global plugin

In order to exercise your newly acquired plugin-writing skills, here's an exercise for you to try out:

> Write a plugin that deletes duplicate lines and deletes redundant blank lines in the document.

You can use either Vim's scripting language or any of the other languages that have an interface with Vim.

If you need "inspiration", see *this Vim Tip*.

How about another one?

> Write a script to fetch the meaning and related words for the current word under the cursor.

Again, if you need "inspiration", see my *lookup.vim plugin*.

15.8 Disabling plugins

Suppose you find Vim acting weirdly and suspect a plugin to be the cause, then you can allow Vim to do selective initialization using the `-u` command line argument.

For example, `vim -u NONE` will start up Vim without any initialization scripts being run. This is pure raw Vim running. Use `vim -u your-minimal-initialization.vim` to run only the specific initializations that you need. This option is useful for debugging if any problems you are facing are present in Vim or have been introduced by a plugin, etc.

See `:help -u` and `:help starting` for details.

15.9 Summary

We have seen the various kinds of plugins available for Vim, how to use such plugins and how to write such plugins. We now have some idea of how extensible Vim is, and how we can write plugins to make our life easier.

Chapter 16

Vim Programmers Editor

16.1 Introduction

Vim tends to be heavily used by programmers. The features, ease of use and flexibility that Vim provides makes it a good choice for people who write a lot of code. This should not seem surprising since writing code involves a lot of editing.

Let me reiterate that typing skills are critical for a programmer. If our *earlier discussion* didn't convince you, hope this *article by Jeff Atwood on 'We Are Typists First, Programmers Second'* will convince you.

If you do not have programming experience, you can skip this chapter.

For those who love programming, let's dive in and see Vim can help you in writing code.

16.2 Simple stuff

The simplest feature of Vim that you can utilize to help you in writing code is to use syntax highlighting. This allows you to visualize, i.e., "see" your code which helps you in reading and writing your code faster and also helps avoid making obvious mistakes.

16.2.1 Syntax highlighting

Suppose you are editing a vim syntax file, run `:set filetype=vim` and see how Vim adds color. Similarly, if you are editing a Python file, run `:set filetype=python`.

To see the list of language types available, check the `$VIMRUNTIME/syntax/` directory.

Tip:

If you want the power of syntax highlighting for any Unix shell output, just pipe it to Vim, for example `svn diff | vim -R -`. Notice the dash in the end which tells Vim that it should read text from its standard input.

16.2.2 Smart indentation

An experienced programmer's code is usually indented properly which makes the code look "uniform" and the structure of the code is more visually apparent. Vim can help by doing the indentation for you so that you can concentrate on the actual code.

If you indent a particular line and want the lines following it to be also indented to the same level, then you can use the `:set autoindent` setting.

If you start a new block of statements and want the next line to be automatically indented to the next level, then you can use the `:set smartindent` setting. Note that the behavior of this setting is dependent on the particular programming language being used.

16.2.3 Bounce

If the programming language of your choice uses curly brackets to demarcate blocks of statements, place your cursor on one of the curly brackets, and press the `%` key to jump to the corresponding curly bracket. This bounce key allows you to jump between the start and end of a block quickly.

16.2.4 Shell commands

You can run a shell command from within Vim using the `:!` command.

For example, if the `date` command is available on your operating system, run `:!date` and you should see the current date and time printed out.

This is useful in situations where you want to check something with the file system, for example, quickly checking what files are in the current directory using `:!ls` or `:!dir` and so on.

If you want access to a full-fledged shell, run `:sh`.

We can use this facility to run external filters for the text being edited. For example, if you have a bunch of lines that you want to sort, then you can run `:%!sort`, this passes the current text to the `sort` command in the shell and then the output of that command replaces the current content of the file.

16.3 Jumping around

There are many ways of jumping around the code.

- Position your cursor on a filename in the code and then press `gf` to open the file.
- Position your cursor on a variable name and press `gd` to move to the local definition of the variable name. `gD` achieves the same for the global declaration by searching from the start of the file.
- Use `]]` to move to the next `{` in the first column. There are many similar motions – see `:help object-motions` for details.
- See `:help 29.3`, `:help 29.4`, and `:help 29.5` for more such commands. For example, `[I` will display all lines that contain the keyword that is under the cursor!

16.4 Browsing parts of the code

16.4.1 File system

Use `:Vex` or `:Sex` to browse the file system within Vim and subsequently open the required files.

16.4.2 ctags

We have now seen how to achieve simple movements within the same file, but what if we wanted to move between different files and have cross-references between files? Then, we can use tags to achieve this.

For simple browsing of the file, we can use the `taglist.vim` plugin.

Taglist in action

1. Install the *Exuberant ctags* program.
2. Install the *taglist.vim* plugin. Refer the "install details" on the script page.
3. Run `:TlistToggle` to open the taglist window. Voila, now you can browse through parts of your program such as macros, typedefs, variables and functions.
4. You can use `:tag foo` to jump to the definition of `foo`.
5. Position your cursor on any symbol and press `ctrl-]` to jump to the definition of that symbol.

- Press `ctrl-t` to return to the previous code you were reading.

6. Use `ctrl-w ]` to jump to the definition of the symbol in a split window.
7. Use `:tnext`, `:tprev`, `:tfirst`, `:tlast` to move between matching tags.

Note that exuberant Ctags supports 33 programming languages as of this writing, and it can easily be extended for other languages.

See `:help taglist-intro` for details.

16.4.3 cscope

To be able to jump to definitions across files, we need a program like cscope. However, as the name suggests, this particular program works only for the C programming language.

cscope in action

1. Install cscope. Refer `:help cscope-info` and `:help cscope-win32` regarding installation.
2. Copy `cscope_maps.vim` to your `~/.vim/plugin/` directory.
3. Switch to your source code directory and run `cscope -R -b` to 'b'uild the database 'r'ecursively for all subdirectories.
4. Restart Vim and open a source code file.
5. Run `:cscope show` to confirm that there is a cscope connection created.
6. Run `:cscope find symbol foo` to locate the symbol `foo`. You can shorten this command to `:cs f s foo`.

You can also:

- Find this definition – `:cs f g`
- Find functions called by this function – `:cs f d`
- Find functions calling this function – `:cs f c`
- Find this text string – `:cs f t`
- Find this egrep pattern – `:cs f e`

See `:help cscope-suggestions` for suggested usage of cscope with Vim.

Also, the *Source Code Obedience* plugin is worth checking out as it provides easy shortcut keys on top of cscope/ctags.

While we are on the subject of C programming language, the *c.vim* plugin can be quite handy.

16.5 Compiling

We have already seen in the *previous chapter* regarding `:make` for the programs we are writing, so we won't repeat it here.

16.6 Easy writing

16.6.1 Omnicompletion

One of the most-requested features which was added in Vim 7 is "omnicompletion" where the text can be auto-completed based on the current context. For example, if you are using a long variable name and you are using the name

repeatedly, you can use a keyboard shortcut to ask Vim to auto-complete and it'll figure out the rest.

Vim accomplishes this via ftplugins, specifically the ones by the name `ftplugin/complete.vim` such as `pythoncomplete.vim`.

Let's start the example with a simple Python program:

```
def hello():
    print 'hello world'

def helpme():
    print 'help yourself'
```

After typing this program, start a new line in the same file, type 'he' and press `ctrl-x ctrl-o` which will show you suggestions for the autocompletion.

If you get an error like `E764: Option 'omnifunc' is not set`, then run `:runtime! autoload/pythoncomplete.vim` to load the omnicompletion plugin.

To avoid doing this every time, you can add the following line to your `~/.vimrc`:

```
autocmd FileType python runtime! autoload/pythoncomplete.vim
```

Vim automatically uses the first suggestion, you can change to the next or previous selection using `ctrl-n` and `ctrl-p` respectively.

In case you want to abort using the omnicompletion, simply press `esc`.

Refer `:help new-omni-completion` for details on what languages are supported (C, HTML, JavaScript, PHP, Python, Ruby, SQL, XML, …) as well as how to create your own omnicompletion scripts.

> Note:
>
> If you are more comfortable in using the arrow keys to select your choice from the omnicompletion list, see *Vim Tip 1228* on how to enable that.

I prefer to use a simple `ctrl-space` instead of the unwieldy `ctrl-x ctrl-o` key combination. To achieve this, put this in your `vimrc`:

```
imap <c-space> <c-x><c-o>
```

Relatedly, the *PySmell plugin* may be of help to Vim users who code in Python.

16.6.2 Using Snippets

Code snippets are small pieces of code that you repetitively tend to write. Like all good lazy programmers, you can use a plugin that helps you to do that. In our case, we use the amazing SnippetsEmu plugin.

1. Download the *snippetsEmu* plugin.
2. Create your `~/.vim/after/` directory if it doesn't already exist.
3. Start Vim by providing this plugin name on the command line. For example, start Vim as `gvim snippy_bundles.vba`
4. Run `:source%`. The 'vimball' will now unpack and store the many files in the appropriate directories.
5. Repeat the same process for `snippy_plugin.vba`

Now, let's try using this plugin.

1. Open a new file called, say, `test.py`.
2. Press the keys `d`, `e`, `f` and then ".
3. Voila! See, how snippetsEmu has created a structure of your function already. You should now see this in your file:

   ```
   def <{fname}>(<{args}>):
   """
   <{}>
   <{args}>"""
   <{pass}>
   <{}>
   ```

   Note:

   In case you see `def` and nothing else happened, then perhaps the snippets plugin is not loaded. Run `:runtime! ftplugin/python_snippets.vim` and see if that helps.

4. Your cursor is now positioned on the function name, i.e., `fname`.
5. Type the function name, say, `test`.
6. Press " and the cursor is automatically moved to the arguments. Tab again to move to the next item to be filled.
7. Now enter a comment: `Just say Hi`
8. Tab again and type `print 'Hello World'`
9. Press tab
10. Your program is complete!

You should now see:

```
def test():
    """
    Just say Hi
    """
    print 'Hello World'
```

The best part is that SnippetsEmu enables a standard convention to be followed and that nobody in the team 'forgets' it.

16.6.3 Creating Snippets

Let's now see how to create our own snippets.

Let us consider the case where I tend to repeatedly write the following kind of code in ActionScript3:

```
private var _foo:Object;

public function get foo():Object
{
    return _foo;
}

public function set foo(value:Object)
{
    _foo = value;
}
```

This is a simple getter/setter combination using a backing variable. The problem is that's an awful lot of boilerplate code to write repeatedly. Let's see how to automate this.

The SnippetsEmu language snippets plugins assume `st` as the start tag and `et` as the end tag – these are the same arrow-type symbols you see in-between which we enter our code.

Let's start with a simple example.

```
exec "Snippet pubfun public
\   function ".st.et.":".st.et."<CR>{<CR>".st.et."<CR>}<CR>"
```

Add the above line to your `~/.vim/after/ftplugin/actionscript_snippets.vim`.

Now open a new file, say, `test.as` and then type `pubfun` and press " and see it expanded to:

```
public function <{}>:<{}>
{

}
```

The cursor will be positioned for the function name, tab to enter the return type of the function, tab again to type the body of the function.

Going back to our original problem, here's what I came up with:

```
exec "Snippet getset private var _".st."name".et.";<CR><CR>public
\   function get
\   ".st."name".et."():".st."type".et."<CR>{<CR>
\   <tab>return _".st."name".et.";<CR>}<CR><CR>public
\   function set ".st."name".et."(value:".st."type".et.")<CR>
\   {<CR><tab>_".st."name".et." = value;<CR>}<CR>"
```

> Note:
>
> All snippets for this plugin *must* be entered on a *single line*. It is a technical limitation.

Follow the same procedure above to use this new snippet:

1. Add this line to your `~/.vim/after/ftplugin/actionscript_snippets.vim`.
2. Open a new file such as `test.as`.
3. Type `getset` and press " and you will see this:

   ```
   private var _<{name}>;

   public function get <{name}>():<{type}>
   {
           return _<{name}>;
   }

   public function set <{name}>(value:<{type}>)
   {
           _<{name}> = value;
   }
   ```

4. Type `color` and press '. `Notice that the variable name`color' is replaced everywhere.
5. Type `Number` and press ". The code now looks like this:

   ```
   private var _color;

   public function get color():Number
   {
           return _color;
   }

   public function set color(value:Number)
   {
           _color = value;
   }
   ```

Notice how much of keystrokes we have reduced! We have replaced writing around 11 lines of repetitive code by a single Vim script line.

We can keep adding such snippets to make coding more lazier and will help us concentrate on the real work in the software.

See `:help snippets_emu.txt` for more details (this help file will be available only after you install the plugin).

16.7 IDE

Vim can be actually used as an IDE with the help of a few plugins.

16.7.1 Project plugin

The Project plugin is used to create a Project-manager kind of usage to Vim.

1. Download the *project* plugin.
2. Unarchive it to your `~/.vim/` directory.
3. Run `:helptags ~/.vim/doc/`.
4. Download Vim source code from
5. Run `:Project`. A sidebar will open up in the left which will act as your 'project window'.
6. Run `c` (backslash followed by 'c')
7. Give answers for the following options

- Name of entry, say 'vim7_src'
- Directory, say `C:repovim7src`
- CD option, same as directory above
- Filter option, say `*.h *.c`

8. You will see the sidebar filled up with the list of files that match the filter in the specified directory.
9. Use the arrow keys or `j/k` keys to move up and down the list of files, and press the enter key to open the file in the main window.

This gives you the familiar IDE-kind of interface, the good thing is that there are no fancy configuration files or crufty path setups in IDEs which usually have issues always. The Project plugin's functionality is simple and straightforward.

You can use the standard fold commands to open and close the projects and their details.

You can also run scripts at the start and end of using a project, this helps you to setup the PATH or set compiler options and so on.

See `:help project.txt` for more details.

16.7.2 Running code from the text

You can run code directly from Vim using plugins such as *EvalSelection.vim* or simpler plugins like *inc-python.vim*

16.7.3 SCM integration

If you start editing a file, you can even make it automatically checked out from Perforce using the *perforce plugin*. Similarly, there is a *CVS/SVN/SVK/Git integration plugin*.

16.7.4 More

To explore more plugins to implement IDE-like behavior in Vim, see:

There are more language-specific plugins that can help you do nifty things. For example, for Python, the following plugins can be helpful:

- *SuperTab* allows you to call omni-completion by just pressing tab and then use arrow keys to choose the option.
- *python_calltips* shows a window at the bottom which gives you the list of possibilities for completion. The cool thing about this compared to omni-completion is that you get to view the documentation for each of the possibilities.
- *VimPdb* helps you to debug Python programs from within Vim.

16.7.5 Writing your own plugins

You can write your own plugins to extend Vim in any way that you want. For example, here's a task that you can take on:

> Write a Vim plugin that takes the current word and opens a browser with the documentation for that particular word (the word can be a function name or a class name, etc.).

If you really can't think of how to approach this, take a look at *"Online documentation for word under cursor"* tip at the Vim Tips wiki.

I have extended the same tip and made it more generic:

```
" Add the following lines to your ~/.vimrc to enable online documentation
" Inspiration:
"  http://vim.wikia.com/wiki/Online_documentation_for_word_under_cursor

function Browser()
    if has("win32") || has("win64")
     let s:browser = "C:Program FilesMozilla Firefoxfirefox.exe -new-tab"
```

```
    elseif has("win32unix") " Cygwin
        let s:browser =
     \ "'/cygdrive/c/Program Files/Mozilla Firefox/firefox.exe' -new-tab"
    elseif has("mac") || has("macunix") || has("unix")
        let s:browser = "firefox -new-tab"
    endif

    return s:browser
endfunction

function Run(command)
    if has("win32") || has("win64")
        let s:startCommand = "!start"
        let s:endCommand = ""
    elseif has("mac") || has("macunix") " TODO Untested on Mac
        let s:startCommand = "!open -a"
        let s:endCommand = ""
    elseif has("unix") || has("win32unix")
        let s:startCommand = "!"
        let s:endCommand = "&"
    else
        echo "Don't know how to handle this OS!"
        finish
    endif

    let s:cmd = "silent " . s:startCommand . " " .
    \   a:command . " " . s:endCommand
    " echo s:cmd
    execute s:cmd
endfunction

function OnlineDoc()
    if &filetype == "viki"
        " Dictionary
     let s:urlTemplate = "http://dictionary.reference.com/browse/<name>"
    elseif &filetype == "perl"
     let s:urlTemplate = "http://perldoc.perl.org/functions/<name>.html"
    elseif &filetype == "python"
        let s:urlTemplate =
        \ "http://www.google.com
        \ /search?q=<name>&domains=docs.python.org&
        \ sitesearch=docs.python.org"
    elseif &filetype == "ruby"
     let s:urlTemplate = "http://www.ruby-doc.org/core/classes/<name>.html"
    elseif &filetype == "vim"
        let s:urlTemplate =
     \ "http://vimdoc.sourceforge.net/search.php?search=<name>&docs=help"
    endif

    let s:wordUnderCursor = expand("<cword>")
```

```
   let s:url = substitute(s:urlTemplate, '<name>', s:wordUnderCursor, 'g')

    call Run(Browser() . " " . s:url)
endfunction

noremap <silent> <M-d> :call OnlineDoc()<CR>
inoremap <silent> <M-d> <Esc>:call OnlineDoc()<CR>a
```

16.8 Access Databases

You can even talk to some 10 different databases from Oracle to MySQL to PostgreSQL to Sybase to SQLite, all from Vim, using the *dbext.vim plugin*. The best part is that this plugin helps you to edit SQL written within PHP, Perl, Java, etc. and you can even directly execute the SQL even though it is embedded within another programming language and even asking you for values for variables.

16.9 Summary

We have learned how Vim can be used for programming with the help of various plugins and settings. If we need a feature, it can be solved by writing our own Vim plugins (as we have discussed in the *Scripting* chapter).

A good source of related discussions is at *Stack Overflow* and *Peteris Krumins's blog*.

Chapter 17

Vim More

17.1 Introduction

We've explored so many features of Vim, but we still haven't covered them all, so let's take a quick wild ride through various topics that are useful and fun.

17.2 Modeline

What if you wanted to specify that a certain file should always use pure tabs and no spaces while editing. Can we enforce that within the file itself?

Yes, just put `vim: noexpandtab` within the first two lines or last two lines of your file.

An example:

```
# Sample Makefile
.cpp:
    $(CXX) $(CXXFLAGS) $< -o $@

# vim: noexpandtab
```

This line that we are adding is called a "modeline."

17.3 Portable Vim

If you keep switching between different computers, isn't it a pain to maintain your Vim setup exactly the same on each machine? Wouldn't it be useful if you could just carry Vim around in your own USB disk? This is exactly what *Portable GVim* is.

Just unzip into a directory on the portable disk, then run `GVimPortable.exe`. You can even store your own `vimrc` and other related files on the disk and use it anywhere you have a Microsoft Windows computer.

17.4 Upgrade plugins

Any sufficiently advanced Vim user would be using a bunch of plugins and scripts added to their `~/.vim` or `~/vimfiles` directory. What if we wanted to update them all to the latest versions? You could visit the script page for each of them, download and install them, but there's a better way – just run `:GLVS` (which stands for 'G'et 'L'atest 'V'im 'S'cripts).

See `:help getscript` for details.

There are scripts to even *twitter from Vim*!

17.5 Dr. Chip's plugins

"Dr. Chip" has written some amazing *Vim plugins* over many years. My favorite ones are the `drawit.vim` which help you to draw actual text-based drawings such as all those fancy ASCII diagrams that you have seen before.

Another favorite is *Align.vim* which helps you to align consecutive lines together. For example, say you have the following piece of program code:

```
a = 1
bbbbb = 2
ccccccccc = 3
```

Just visually select these three lines and press `t=`, and voila, it becomes like this:

```
a          = 1
bbbbb      = 2
ccccccccc = 3
```

This is much easier to read than before and makes your code look more professional.

Explore Dr. Chip's page to find out about many more interesting plugins.

17.6 Blog from Vim

Using the *Vimpress plugin*, you can blog to your WordPress blog right within Vim.

17.7 Make Firefox work like Vim

Use the *Vimperator add-on* to make Firefox behave like Vim, complete with modal behavior, keyboard shortcuts to visit links, status line, tab completion and even marks support!

17.8 Bram's talk on the seven habits

Bram Moolenaar, the creator of Vim, had written an article long ago titled *Seven habits of effective text editing* that explained how you should use a good editor (such as Vim).

Bram recently gave a talk titled *Seven habits for effective text editing, 2.0* where he goes on to describe the newer features of Vim as well as how to effectively use Vim. This talk is a good listen for any regular Vim user.

17.9 Contribute to Vim

You can contribute to Vim in various ways such as *working on development of Vim itself*, *writing plugins and color schemes*, contributing *tips* and helping with the *documentation*.

If you want to help in the development of Vim itself, see `:help development`.

17.10 Community

Many Vim users hang out at the *vim@vim.org mailing list* where questions and doubts are asked and answered. The best way to learn more about Vim and to help other beginners learn Vim is to frequently read (and reply) to emails in this mailing list.

You can also ask questions at *Stack Overflow* by tagging the question as 'vim' and you'll find useful discussions there, such as the one on *"What are your favorite vim tricks?"*.

You can also find articles and discussions at *delicious* and *reddit*.

17.11 Summary

We've seen some wide range of Vim-related stuff and how it can be beneficial to us. Feel free to explore these and many more *Vim scripts* to help you ease your editing and make it even more convenient.

Chapter 18

Vim What Next

18.1 Introduction

We have explored so much about Vim, so what next?

Well, if you have learned, understood and made Vim *a habit*, then you are officially a Vimmer. Congratulations!

Now, immediately *send me a mail* thanking me for this book;-) . This step is optional but recommended.

Also please consider *making a donation* to support the continued development of this book, or donate to the Vim project to [help children in Uganda][3].

Next, I recommend to regularly follow the *Vim mailing list* to see the kind of queries and answers that appear there. You would be surprised to see the amount of activity and the range of flexibility of Vim that is demonstrated by all the discussion. And you never know, you might be able to answer a few queries too.

Two other important resources that should be good to explore are the *Best of Vim Tips* page and the all-important `:help user-manual` that is an index to everything possible in Vim. Make sure to refer the user-manual from time to time and familiarize yourself on how to find the relevant information so that you can look up for commands as and when required.

Lastly, if you want to know what are the latest cool features in Vim, then see `:help new-7`.

18.2 Summary

We have gone through a very broad overview of the range of things possible in Vim. The most important thing that we have stressed on again and again is to not learn every feature possible, but to make the features, most useful to you,

a habit, and to make sure to use the `:help` to look for features as and when needed.

So, go forth and skyrocket your editing skills to be more efficient and effective than you ever thought possible.

Happy Vimming!

Contents