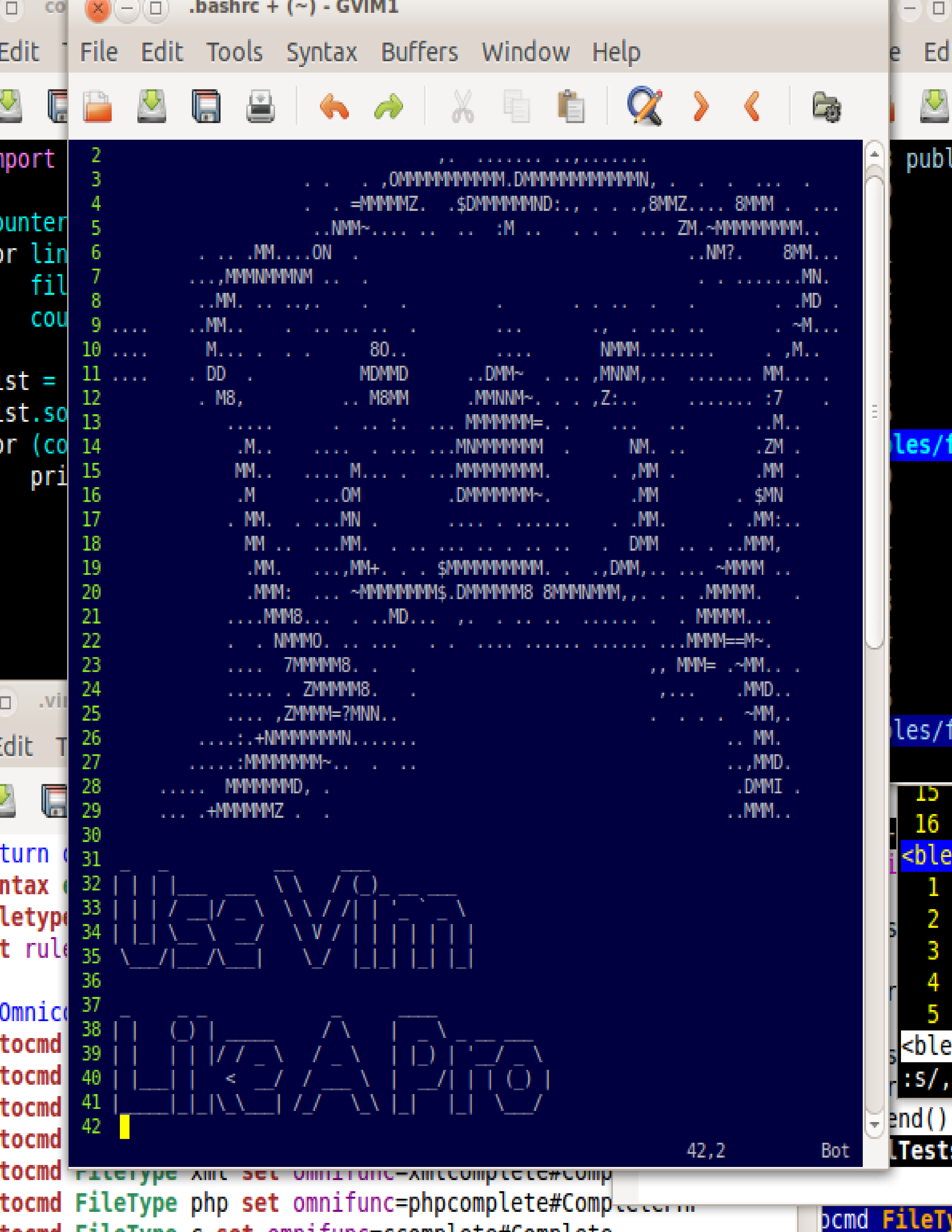

.bashrc + (~) - GVIM1
File Edit Tools Syntax Buffers Window Help
Use Vim
Like A Pro
42,2
Bot

Use Vim Like A Pro

Go from noob to pro

Tim Ottinger

This book is for sale at http://leanpub.com/VimLikeAPro

This version was published on 2014-06-22

Tweet This Book!

Please help Tim Ottinger by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I am ready to #UseVimLikeAPro

The suggested hashtag for this book is #vimLikeAPro.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#vimLikeAPro

Contents

Why Bother? (reasons)

There are many other editors. Several are excellent. There is no reason why you cannot use all of them.

You might want to learn vim for any of these reasons:

- With the sudden rise in Unix use (Linux and Mac OS X, in particular) the text editor known as *vim* ("vi improved") has become ubiquitous
- *vim* has a small footprint in RAM and on the CPU. A given system can support a great many *vim* users at once.
- *vim* has a lot of "superpowers", which make editing quite efficient.
- *vim* has "geek appeal".
- *vim* has a very active user/developer community. It always has.
- Learning new stuff is good for your brain.

Why Write This Tutorial (approach)

Some other tutorials are very good, and google/yahoo/bing/whatever can help you find them all.

There are also some great books that have been written since I started this tutorial. Those books are far more comprehensive and have had a lot of investment from their publishers and editors.

This little tutorial has been around for a long time and has been strangely popular. I like to think it is because I have taken a slightly different approach.

I wrote this for the impatient developer.

There is a certain mental model that makes mastering *vim* much faster. I don't know any other materials that use the same approach, or which teach as deeply in such a small space.

I've agonized and organized (and re-agonized, and reorganized) the tutorial for top-to-bottom learning, so that anyone who emerges from the other end of this tutorial will have professional-grade editing skills, probably better than many of their more experienced colleagues.

When you are done here, you may want to invest in a much more comprehensive book. I am keenly, painfully aware of how much material I have intentionally left out.

You'll be pleased to know that I continue to look for things to leave out, or faster ways to shrink the content. Well, that is, other than this apologetic section.

I think this is one of the fastest ways to improve your use of *vim*, and a pretty good way to start using *vim* from scratch.

This work started out as a web page, for free. Now I am using leanpub because I like the formats and styling I can get with their system.

I stil have a “suggested price” of zero dollars. Free. Gratis.

Leanpub has a nice system that allows people to give me small monetary gifts if they want to. I have had some pizza and scotch money that I would not otherwise have had. Thank you for the kindness you have shown. It is more than I expected.

How should one use the tutorial? (usage)

Look at each subsection heading as the beginning of a separate lesson, and spend a little time with it before moving onward. Maybe spend a day with each bit of knowledge, and maybe a several days when the lesson is particularly meaty.

Don’t be in a hurry. Don’t rush your brain, lest you forget old things as fast as you learn new ones. Consider doing a few lessons a week. People have used *vim* for 10 years and still don’t know half as much as you’ll learn in the first major section; you can take a few months to work through this.

You can’t learn *vim* without using *vim*, so you should have some text files (preferably open source program code) to work with. It is better yet if you are using *vim* at work. It also helps if you work with a partner who is also reading this tutorial, so that you can reinforce each other.

What can I do with this tutorial? (license)

This work is licensed under a Creative Commons Attribution 3.0 License[1].

Copy it, share it, paste it into your web page. Don’t pretend it is your own stuff, and please give me some attribution. As a courtesy, if you find it worth distributing, I wouldn’t mind getting a copy or a link. Just let me know[2].

[1]http://creativecommons.org/licenses/by/3.0/

[2]mailto:tottinge@gmail.com

Master The Basics

A little reassurance first.

Nobody knows all of *vim*. Nobody needs to know it all. You only need to know how to do your own work. The secret is to not settle for crummy ways of doing work.

vim has word completion, and undo, and shortcuts, and abbreviations, and keyboard customization, and macros, and scripts. You can turn this into *your* editor for *your* environment.

That is cool, but it may be reassuring to know that you can probably will not need to. You can be far more productive without touching any of deeply advanced features.

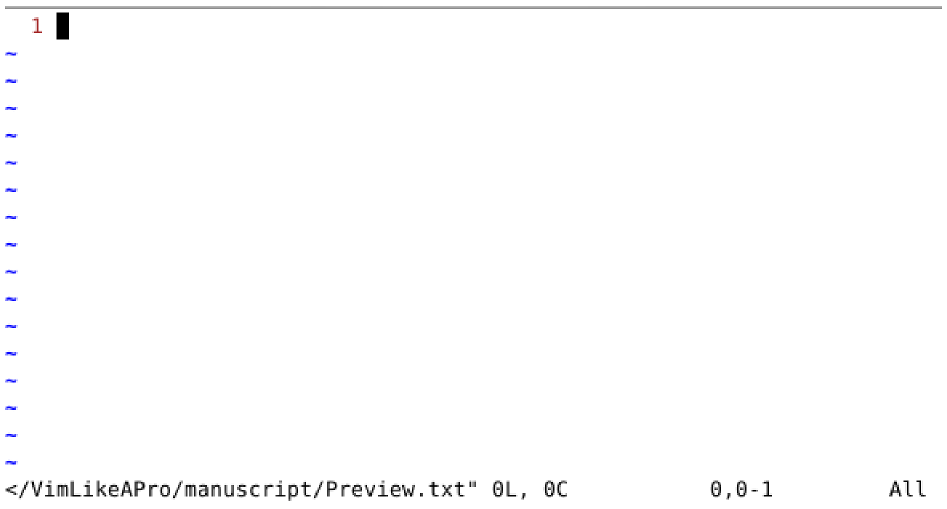
As Bram Moolenaar (*vim*'s primary author) says, the best way to learn *vim* is to use it and ask questions. This little tutorial is full of questions you might not have thought to ask. That's the main value I can give you.

vim has a built-in tutorial. You might want to try it, especially if you don't like my tutorial. All you have to do is type "vimtutor" at the command line. It is a very nice tutorial, and is rather complete (compared to mine, which is fairly nice but not very complete at all).

Finally, please consider GVIM. It will make your experience much more pleasant. If you only have *vim*, then you can still use it and learn, but GVIM has a much nicer look, lets you use your mouse and scroll wheel, and has menus and icons for those of you who are used to such things.

What does it look like?

It does not look like much. It was not built for beauty. *vim* uses the default terminal appearance. GVim adds menu bars and stuff, but *vim* looks like this:



```
  1
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
</VimLikeAPro/manuscript/Preview.txt" 0L, 0C          0,0-1         All
```

a screenshot of *vim* in action

As far as visible features, there is:

- the line number (I have line numbers turned on by default, you might not),
- a bunch of tildes (∼) marking empty lines
- a line of status at the bottom of the screen.

You can usually tell by the blank line markers that you are in *vim*.

The status line can be turned off, but in this case it shows:

- the rightmost ∼n∼ characters of the filename,
- the number of lines and characters in the file,
- the location of the cursor.
- where you are in the file ("All" because its all showing).

It is not exciting, and that is good.

Modality

The original vi was invented back when "green screen" ascii terminals were the UI innovation of the day (ask your dad about ascii terminals). There were not so many shift-like keys (shift, alt, ctrl, windows, fn) and there was no such thing as a pointing device. Pretend that there was only a "ctrl" key and a "shift" key, whether it is true or not.

Programming and all other computer use)was done with your eyes on the screen and two hands on the keyboard. Vi made it possible to do so **quickly**, because vi is a bit like a video game, where any little gesture on the keyboard causes something to happen.

If you are using *vim* and pressing Whatkeys causes either cool or unfortunate things to happen, you know you are in the `command mode`, which is the default state of the editor. Commands are assigned to the ordinary everyday keys like 'p' and 'y' and 'g', not chords like Control-Alt-Shift-Escape.

vim has combinations and sequences to get the special power-ups like navigating between functions in separate files and reformatting entire lists in the middle of a document, code completion, abbreviations, templates and the like but that is for later.

There has to also be a way to type text into a document, but most of the keys already have special meanings! The only reasonable option was for the developers to create an "insert `mode`" which would make the 'a' key type an 'a' character, just like a typewriter (ask your dad what a typewriter is). This is called "insert `mode`". Not much happens in "insert `mode`" except normal, old, boring typing. You only want to use insert `mode` when you must do typing, but all the cool stuff happens in the normal (control) `mode`.

You will learn many convenient ways to get into insert `mode`, but for now you should know that the way out of insert `mode`, back to the video-game-like control `mode`, is to press the ESCAPE key.

Understanding that you have basically two `modes` of operation will make your stay in *vim* less confusing, and starts you on your way to *vim* guruhood.

Know the *vim* command pattern

Most of the time you will either get an immediate result from a keystroke, or you will type a command and a movement command (often repeating the same keystroke: the "double-jump"). When you start to learn the other bits and pieces (registers, repeats, etc) then you might think *vim* is inconsistent, and this is not so. The command pattern is rather consistent, but some parts are optional.

```
register repeats operation movement
```

item	**meaning**
`register`	Register name (optional, with default cut/paste register used if not otherwise specified)
`repeats`	Repeats (optional): 13
`operation`	Operation: y (for yank)
`movement`	Movement (depending on the operation): yy (repeated to take current line, a convention used in vi)

vim commands work with the pattern shown above. There are some commands that don't use register and some that don't take movement, but for the most part this is the way it goes.

A register is essencially a cut-n-paste buffer. In most editors you get only one. In *vim* you have too many, but you don't have to use them, so don't worry about it untl you get to the lesson on *registers*.

A repeat is a number of times you want to do something. If you don't type in a number, the default is 1.

An operation is a keystroke that tells *vim* to do something. These are mostly normal keypresses, and most operators do not require shifts or alts or controls.

Movement is a command that takes the cursor somewhere. There are a lot of them, because there are lots of ways you need to move. don't panic, though, because you can use the arrow keys if you really have to. There is a whole section of this tutor on moving around.

Lets try an example to clarify how the pattern works. If I want to copy 13 lines into my copy/paste register, I can skip specifying a register name, type 13 for a repeat count, press 'y' for yank, and then press one more 'y' as a movement command (meaning current line). That yanks 13 lines into the default cut-n-paste register. If I press 'p' (choosing to use no register name and no repeat, recognizing that put has no movement command), then those lines are pasted back into my document just after my current line.

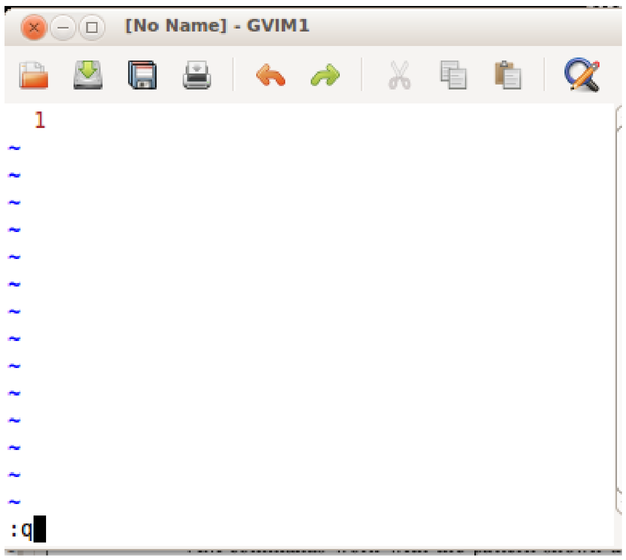
If you know this pattern, then you will know how to leverage everything else you learn about vi. Learn all the convenient ways to move, and you will know how to cut, paste, reformat, shift, and many other things you want to do.

GET OUT!

You should be able to get out of a *vim* session once you are in it. There are a few ways to do so. Try these:

What to Type	What it does
`:q`	Quit the current window (or editor if you're out of windows) if there are no unsaved changes.
`:q!`	Quit the current window even if there are unsaved changes.
`:qa`	Quit all windows unless there are unsaved changes.
`:qa!`	Quit all windows even if there are unsaved changes.
`:wq`	Save changes and quit the current window.
`ZZ`	Save changes and quit current window

When you type a colon, the cursor drops to the lower left corner of the screen. Later you will know why. For now, it is enough to know that it is supposed to do that, and that these :q commands will work. Notice that there is no : in front of ZZ.



```
1
:q
```

how it looks when you quit

If you can't get out of *vim*, you should check to be sure the caps lock is OFF, and press the escape button. If it feels good, press it a couple of times. If it beeps, you know that you've escaped enough. Then these exit commands should work.

Mnemonics

Not all commands are mnemonic. They tried, but there are more than 26 things you might want to do in a text editor, and the distribution of letters means that not that many words start with a 'q' and happen to be meaningful in editing. However, many commands are mnemonic. There are commands for moving Forward, Back, a Word at a time, etc.

A great many are mnemonic if you know the jargon. Since "copy" and "cut" both start with "c", we have the vernacular of "yank" (for copy), "delete" (for cut), and "put" (for paste). Y, D, P. It seems a little funky but it is possible to remember these. Remember, eventually it becomes muscle memory, but the authors of VI and *vim* tried not to be arbitrary when it was totally up to them. Sometimes, there wasn't much of an option.

Invocation

Now that you know how to get out of *vim*, maybe it is time to learn how to get into *vim*. We typically start *vim* from the command line, though you may have menues or other ways.

There are a few ways you can start *vim*.

What to Type	What it does
`vim`	start with an empty window
`vim file.txt`	start with an file.txt loaded and ready to edit
`vim +23 file.txt`	start with an file.txt loaded and ready to edit at line 23.
`vimtutor`	Start in tutorial mode. This is a good idea.
`vimdiff oldfile.txt newfile.txt`	Start *vim* as a really fancy code merge tool.
`vimdiff .`	Start *vim* as a file explorer.

There is more, not shown. For now, knowing these will help you to get started. DO try out the vimtutor and the vimdiff. Some of these won't work until you set up a .vimrc, but that is explained later.

```
>vim Book.txt
```

starting vim

If you type `gvim` instead of `vim` (mvim on OS X) then you will get the gee-whiz, cool, gui version of *vim* (if it is installed). It has some extra powers. You'll typically like it better than the plain *vim*. It is like *vim* with chocolate icing. Everything we say about *vim* here is also true of GVIM, so you can use the same tutorial with either.

You don't have to edit one file at a time. You can start (g)*vim* with multiple filename arguments. When you do, there are a few options you can pass to get some fun additional effects. Of course, these are more fun after you learn how to work with split windows, so you can refer back to it later.

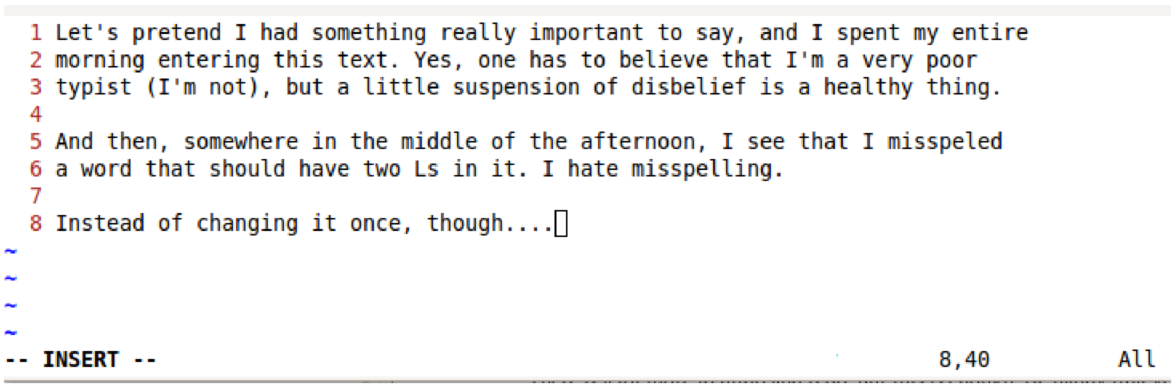
Option	What it does
`-o`	Open multiple files in horizontally tiled windows .
`-O`	Open multiple files in vertically tiled windows .
`-p`	Open multiple files in separate tabs (I hate this).

Don't panic. You have undo/redo

The command for undo is `u`. That is not too hard to remember, is it? A lot of *vim* commands are pretty mnemonic-friendly.

The redo would be the `r` key, but the `r` is used for "replace" (we'll talk later about this). We're stuck with `control-R` instead. Ah, well. You can't have everything.

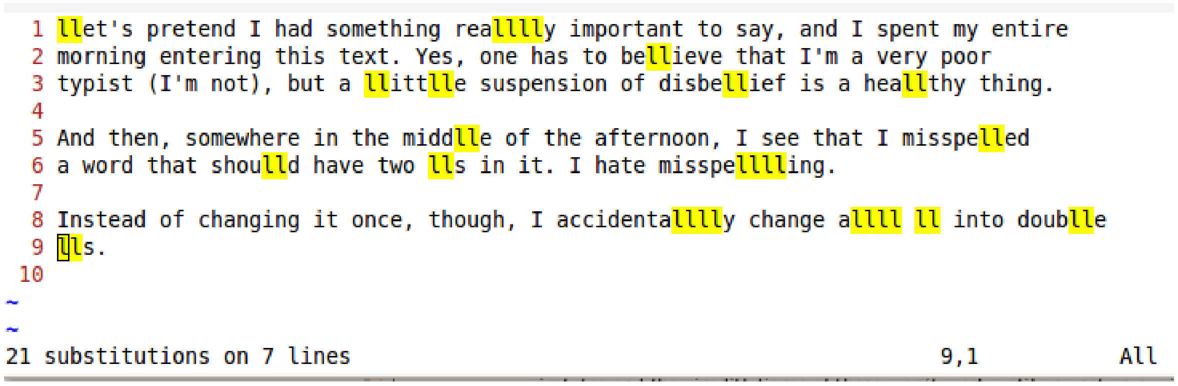
This is a good place for an example, so lets start with some precious, text that cost me a whole morning of text (well, a couple of minutes at least)



```
Let's pretend I had something really important to say, and I spent my entire
morning entering this text. Yes, one has to believe that I'm a very poor
typist (I'm not), but a little suspension of disbelief is a healthy thing.

And then, somewhere in the middle of the afternoon, I see that I misspeled
a word that should have two Ls in it. I hate misspelling.

Instead of changing it once, though....
~
~
~
~
-- INSERT --                                                 8,40         All
```

before making the error

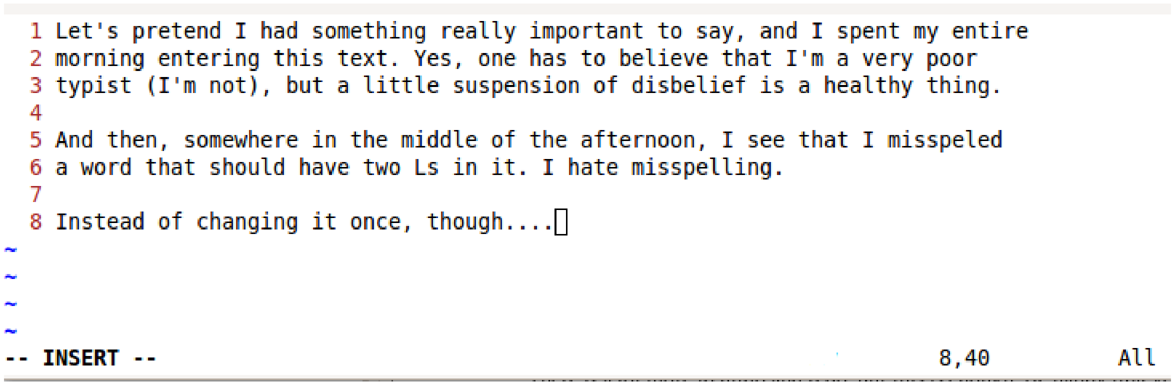
Ewww. Misspelling. Yuck, Lets change that L into a double-L.



```
llet's pretend I had something realllly important to say, and I spent my entire
morning entering this text. Yes, one has to bellieve that I'm a very poor
typist (I'm not), but a llittlle suspension of disbellief is a heallthy thing.

And then, somewhere in the middlle of the afternoon, I see that I misspelled
a word that shoulld have two lls in it. I hate misspellllling.

Instead of changing it once, though, I accidentallllly change allll ll into doublle
lls.

~
~
21 substitutions on 7 lines                                  9,1          All
```

after the error

Wow. That is far worse. As a pro *vim* user, I press the `u` button for undo.



```
Let's pretend I had something really important to say, and I spent my entire
morning entering this text. Yes, one has to believe that I'm a very poor
typist (I'm not), but a little suspension of disbelief is a healthy thing.

And then, somewhere in the middle of the afternoon, I see that I misspeled
a word that should have two Ls in it. I hate misspelling.

Instead of changing it once, though....
~
~
~
~
-- INSERT --                                                 8,40         All
```

starting vim

There is a lot more to undo and redo, but this is enough. Be happy that you can revert changes, and un-revert them. *vim* isn't as powerless and unforgiving as you feared it might be, though you might still not like it very much. Just wait for that muscle memory to kick in.

If you get into a real mess, then exit the editor without saving.

If you are really afraid, or really cautious, then you should have version control for your text files. I recommend you start editing with junk files in a junk directory anyway, but when you are working on something important, you should not be afraid to make changes. Version control is a good security blanket and a useful backup strategy. Consider using Git or Mercurial, both of which are easy and powerful.

Move by context, not position

The poor soul who is using *vim* for the first time will be found pressing up and down arrows and executing key repeats, moving horribly inefficiently through any body of code. He will be scrolling or paging (btw: `\^f` moves forward one page, `\^b` moves backward one page) and searching with his poor eyeballs through piles of code. This poor soul is slow and clueless, and probably considers *vim* to be a really bad version of windows notepad instead of seeing it as the powerful tool it is.

By the way, the arrow keys do not always work for *vim*, but do not blame *vim*. It is actually an issue with the way your terminal is set up. *vim* can't tell that your arrow keys are arrow keys. If you have the problem, you have more research to do.

To use *vim* well, it is essential that you learn how to move well.

Do not search and scroll. Do not use your eyes to find text. They have computers for that now. Here are a handful of the most important movement commands. The best way to move is by searching:

What to Type	What it does
`/`	search forward: will prompt for a pattern
`?`	search backward: will prompt for a pattern
`n`	repeat last search (like dot for searches!)
`N`	repeat last search but in the opposite direction.
`tx`	Move "to" letter 'x' (any letter will do), stopping just before the 'x'. Handy for change/delete commands.
`fx`	"Find" letter 'x' (any letter will do), stopping **on** the letter 'x'. Also handy for change/delete commands

If you're not searching, at least consider jumping

What to Type	What it does
`gg`	Move to beginning of file
`G`	Move to end of file
`0`	Jump to the very start of the current line.
`w`	Move forward to the beginning of the next word.
`W`	Move forward to the beginning of the next space-terminated word (ignore punctuation).
`b`	Move backward to the beginning of the current word, or backward one word if already at start.
`B`	Move backward to the beginning of the current space-terminated word, ignoring punctuation.
`e`	Move to end of word, or to next word if already at end.
`E`	Move to end of space-terminated word, ignoring punctuation

The following commands are handy, and are even sensible and memorable if you know regex:

What to Type	What it does
`\ˆ`	Jump to start of text on the current line. Far superior to leaning on left-arrow or h key.
`$`	Jump to end of the current line. Far superior to leaning on right-arrow or k key.

Here is some fancy movement

What to Type	What it does
`%`	move to matching brace, paren, etc
`}`	Move to end of paragraph (first empty line).
`{`	Move to start of paragraph.
`(`	Move to start of sentence (separator is both period and space).
`)`	Move to start of next sentence (separator is both period and space).
`''`	Move to location of your last edit in the current file.
`]]`	Move to next function (in c/java/c++/python)
`[[`	Move to previous function/class (in c/java/c++/python)

Finally, if you can't move by searching, jumping, etc, you can still move with the keyboard, so put your mouse down.

What to Type	What it does
`h`	move cursor to the left
`l`	move cursor to the right
`k`	move cursor up one line
`j`	move cursor down one line
`\ˆf`	move forward one page
`\ˆb`	move backward one page

You want to use the option `hls` (for “highlight search”) in your vimrc. You will learn about that soon enough. In the short term you can type “`:set hls`” and press enter.

Help is on its way.

There is an online help mechanism in *vim*. You should know how to use it.

Type `:help` and you will get a split window with help text in it. You can move around with the arrow keys, or with any of the *vim* movement commands you will learn.

You can always enter funky keys by pressing ˆv first, and then the keystroke. This is most useful in help. You can type `:help \^v\^t` to get help for the keystroke ˆt. By convention you can usually get what you want by typing `:help CTRL-T` also. Do not underestimate how handy this is.

Most distributions of *vim* will install a program called `vimtutor`. This program will teach you to use *vim*. It will do so by using *vim*. It is a handy piece of work (props to the author!).

Help has links. If you see one you like, you can move the cursor to the link (lets not just beat on the arrow keys, here!) and press ˆ]. Yeah, it is an odd and arbitrary-looking command. That will not only navigate to the link, but also push it on a stack. If you want to go back, you can press ˆt (yes, also pretty arbitrary) to pop the current link off the stack and return to the previous location in the help. The commands `\^]` and `\^t` aren’t very memorable, but we’ll use them for code navigation later, so learning them is not a total waste of mental energy.

Shifted letters and DEATH BY CAPS!

For a number of commands, shift will either reverse the direction of a command (so N is the opposite of n, see next bullet) or will modify how the command works. When moving forward by one word at a time (pressing `w`), one may press `W` to move forward by one word but with W the editor will consider punctuation to be part of the word. The same is true when moving backward with `b` or `B`.

Because a shifted letter may mean something very different from the same letter unshifted, you must be very careful not to turn on the capslock! Sometimes a poor unwary soul will accidentally hit the capslock. When he intends to move left with ‘j’, he instead joins the current line with the next. Many other unwanted edits can take place as his fingers make a quick strafing run for some complex edit. It is ugly.

If you encounter DEATH BY CAPS, you should turn off the capslock, and then try pressing ‘u’ repeatedly to get rid of unwanted edits. If you feel that it is a lost cause, press “:e!” followed by pressing the enter key. That will reload the file from disk, abandoning all changes. It is a troublesome thing that will eventually happen to you. Some people turn off their capslock key entirely for this reason.

Quoting Your Regex Metacharacters

If you do not know what a regex is, skip this section. For those who understand what a regex is, and who realize that the "/" command takes a regex rather than just normal text, this will be important. For the rest of you, it will seem totally out of place and should be skipped for now.

You should know how to use regular expressions, because a few tricks in regex will make your whole Unix/Linux/Mac experience a little better. It is too large a topic to expose fully here, but you might try looking at on of the good references or [3]tutorials[4] elsewhere on the web.

The main thing to remember is that *vim* will side with convenience when it comes to regex. Since you search a lot, *vim* will assume that `/+` means that you want to search for the nearest + character. As a result, all the metacharacters have to be quoted with the backslash ("") character. It is sometimes a pain, but if you really want to find a plus sign followed by a left-parenthesis, it is very easy.

Insert, Overwrite, Change

In *vim* you have a variety of ways to start entering text, as mentioned above in the section on Modality.

You are normally in `command mode`. When you type certain keys, you are placed in insert mode or overtype mode. In insert mode, the text you type goes before the cursor position, and everything after the cursor is pushed to the right or to the next line.

In ‘ overtype mode‘ your keystrokes are input, just as they are in insert mode, but instead of inserting the keystrokes *vim* will replace the next character in the document with the character you type. You get to overtype mode by pressing an overtype key command while in command mode.

In `ex mode` you are typing a string of commands to run into a little window at the bottom of the screen. We'll talk about this later on, because it is powerful stuff. It is also a little cryptic, so we will wait. You get into ex mode by typing ":" in command mode.

You always return to command mode from overtype, insert, or command mode by pressing escape. That is one handy key.

[3]http://www.geocities.com/volontir/%3E

[4]http://larc.ee.nthu.edu.tw/~cthuang/vim/files/vim-regex/vim-regex.htm

What to Type	What it does
`i`	insert before the current cursor position
`I`	insert at the beginning of the current line. Far better than pressing ˆ and then i.
`a`	insert after the current cursor position
`A`	insert/append at the end of the current line. Far better than pressing $ and then i.
`r`	retype just the character under the cursor
`R`	Enter overtype (replace) mode, where you destructively retype everything until you press ESC.
`s`	(substitute) delete the character (letter, number, punctuation, space, etc) under the cursor, and enter insert mode
`c`	the 'change' (retype) command. *Follow with a movement command.* `cw` is a favorite, as is `cc`
`C`	Like 'c', but for the entire line.
`o`	insert in a new line below the current line
`O`	insert in a new line above the current line
`:`	Enter command mode (for the advanced student)
`!`	Enter shell filter mode (for the very advanced student)

Consider the value of the `c` command. If you use it with the ' `t or f commands, it becomes very powerful. If you were at the C at the beginning of the previous sentence, you could type` ct. `and retype the whole first sentence, preserving the period. The same is true with other commands, such as the` d' for delete. The movement commands add a lot of power to the change command, and that is one reason why it is important to learn to move well.

NEVER PARK IN INSERT MODE.

vim is set up to do more navigating and editing than typing. It rewards you for working in the same way, mostly in control mode with spurts of time in insert mode.

If you try to use *vim* as a weak form of notepad, modality and navigation will ensure that you are never really efficient. If you want to sail, you have to get in the boat, and if you want to get good at *vim*, you need to get good in command mode.

So, if you are stopping to think, hit `<esc>`. If you aren't in the middle of text typing, you should be in command mode. If you are wanting to move up or down a line, or to some other place, hit

Epilog

If you have followed the tutorial this far, you have a good start. There is much more to learn, and much further to go.

As a graduate of the Vim Like A Pro school of editing, you must uphold the standards.

- Do Not Park In Insert
- Avoid Death By Caps
- Don’t mindlessly tap-tap-tap
- Make your work easier
- Learn always!
- Remember there are other tools. Use them, too.