

FREE eBook

LEARNING vim

Free unaffiliated eBook created from
Stack Overflow contributors.

#vim

www.dbooks.org

Table of Contents

About 1

Chapter 1: Getting started with vim 2

Remarks 2

Versions 3

Examples 4

Installation 4

Installation on Linux/BSD 4

Arch and Arch-based distributions 4

Debian and Debian-based distributions 4

Gentoo and Gentoo-based distributions 4

RedHat and RedHat-based distributions 4

Fedora 4

Slackware and Slackware-based distributions 5

OpenBSD and OpenBSD-based distributions 5

FreeBSD and FreeBSD-based distributions 5

Installation on Mac OS X 5

Regular install 5

Package manager 5

Installation on Windows 5

Chocolatey 6

Building Vim from source 6

Exiting Vim 6

Explanation: 7

Interactive Vim Tutorials (such as vimtutor) 7

Saving a read-only file edited in Vim 7

Command Explanation 8

Suspending vim 8

Basics 9

What to do in case of a crash 10

Chapter 2: :global 12

Syntax 12

Remarks ... 12
Examples ... 12
Basic usage of the Global Command ... 12
Yank every line matching a pattern ... 12
Move/collect lines containing key information ... 12
Chapter 3: Advantages of vim ... 14
Examples ... 14
Customization ... 14
Lightweight ... 14
Chapter 4: Ask to create non-existant directories upon saving a new file ... 15
Introduction ... 15
Examples ... 15
Prompt to create directories with :w, or sliently create them with :w! ... 15
Chapter 5: Autocommands ... 16
Remarks ... 16
Examples ... 16
Automatically source .vimrc after saving ... 16
Refresh vimdiff views if a file is saved in another window ... 17
Chapter 6: Auto-Format Code ... 18
Examples ... 18
In normal mode: ... 18
Chapter 7: Buffers ... 19
Examples ... 19
Managing buffers ... 19
Hidden buffers ... 19
Switching buffer using part of the filename ... 19
Quickly switch to previous buffer, or to any buffer by number ... 20
Chapter 8: Building from vim ... 21
Examples ... 21
Starting a Build ... 21
Chapter 9: Command-line ranges ... 22

www.dbooks.org

Examples ... 22
Absolute line numbers ... 22
Relative line numbers ... 22
Line shortcuts ... 22
Marks ... 22
Search ... 23
Line offsets ... 23
Mixed ranges ... 23
Chapter 10: Configuring Vim ... 24
Examples ... 24
The vimrc file ... 24
Which options can I use? ... 24
Files and directories ... 25
Options ... 26
Setting boolean options ... 26
Setting string options ... 26
Setting number options ... 26
Using an expression as value ... 26
Mappings ... 27
Recursive mappings ... 27
Non-recursive mappings ... 27
Executing a command from a mapping ... 27
Executing multiple commands from a mapping ... 27
Calling a function from a mapping ... 28
Mapping a <Plug>mapping ... 28
Variables ... 28
Commands ... 28
Examples ... 28
Functions ... 29
Example ... 29
Script functions ... 29
Using s:functions from mappings ... 29

Autocommand groups ... 30
Example ... 30
Conditionals ... 30
Setting Options ... 30
Syntax Highlighting ... 31
Color Schemes ... 31
Changing Color Schemes ... 31
Installing Color Schemes ... 31
Toggle line enumerating ... 32
Plugins ... 32
Chapter 11: Converting text files from DOS to UNIX with vi ... 33
Remarks ... 33
Examples ... 33
Converting a DOS Text file to a UNIX Text file ... 33
Using VIm's fileformat ... 33
Chapter 12: Differences between Neovim and Vim ... 35
Examples ... 35
Configuration Files ... 35
Chapter 13: Easter Eggs ... 36
Examples ... 36
Help! ... 36
When you're feeling down ... 36
The Answer ... 36
Looking for the Holy Grail ... 36
Ceci n'est pas une pipe ... 36
When a user is getting bored ... 37
Spoon ... 37
Knights who say Ni! ... 37
nunmap ... 37
Chapter 14: Enhanced undo and redo with a undodir ... 39
Examples ... 39
Configuring your vimrc to use a undodir ... 39

www.dbooks.org

Chapter 15: Exiting Vim 40
Parameters 40
Remarks 40
Examples 40
Exit with save 40
Exit without save 40
Exit forcefully (without save) 40
Exit forcefully (with save) 41
Exit forcefully from all opened windows (without save) 41
if multiple files are opened 41
Chapter 16: Extending Vim 42
Remarks 42
Examples 42
How plugins work 42
The principle 42
The manual method 43
Single file plugin 43
Bundle 43
VAM 43
Vundle 44
Installing Vundle 44
Supported Plugin Formats 44
The future: packages 45
Pathogen 45
Installing Pathogen 45
Using Pathogen 45
Benefits 46
Chapter 17: Filetype plugins 47
Examples 47
Where to put custom filetype plugins? 47
Chapter 18: Find and Replace 48

Examples ... 48
Substitute Command ... 48
Replace with or without Regular Expressions ... 49
Chapter 19: Folding ... 51
Remarks ... 51
Examples ... 51
Configuring the Fold Method ... 51
Creating a Fold Manually ... 51
Opening, Closing and Toggling Folds ... 51
Showing the Line Containing the Cursor ... 52
Folding C blocks ... 52
Chapter 20: Get :help (using Vim's built-in manual) ... 53
Introduction ... 53
Syntax ... 53
Parameters ... 53
Examples ... 53
Getting started / Navigating help files ... 53
Searching the manual ... 54
Chapter 21: How to Compile Vim ... 55
Examples ... 55
Compiling on Ubuntu ... 55
Chapter 22: Indentation ... 56
Examples ... 56
Indent an entire file using built-in indentention engine ... 56
Indent or outdent lines ... 56
Chapter 23: Inserting text ... 58
Examples ... 58
Leaving insert mode ... 58
Different ways to get into insert mode ... 58
Insert mode shortcuts ... 59
Running normal commands from insert mode ... 59
Example ... 59

www.dbooks.org

Insert text into multiple lines at once ... 60
Paste text using terminal "paste" command ... 60
Pasting from a register while in insert mode ... 61
Advanced Insertion Commands and Shortcuts ... 61
Disable auto-indent to paste code ... 62
Chapter 24: Key Mappings in Vim ... 63
Introduction ... 63
Examples ... 63
Basic mapping ... 63
map Overview ... 63
map Operator ... 63
map Command ... 64
Examples ... 64
Map leader key combination ... 64
Illustration of Basic mapping (Handy shortcuts) ... 65
Chapter 25: Macros ... 66
Examples ... 66
Recording a macro ... 66
Editing a vim macro ... 66
Recursive Macros ... 67
What is a macro? ... 67
Record and replay action (macros) ... 68
Chapter 26: Manipulating text ... 70
Remarks ... 70
Examples ... 70
Converting text case ... 70
In normal mode: ... 70
In visual mode: ... 70
Incrementing and decrementing numbers and alphabetical characters ... 70
Incrementing and decrementing numbers ... 70
Incrementing and decrementing alphabetical characters ... 71
Incrementing and decrementing numbers when alphabetical increment/decrement is enabled ... 71

Formatting Code 72
Using "verbs" and "nouns" for text editing 72
Chapter 27: Modes - insert, normal, visual, ex 74
Examples 74
The basics about modes 74
Normal mode (or Command mode) 74
Insert mode 74
Visual mode 74
Select mode 74
Replace mode 74
Command-line mode 75
Ex mode 75
Chapter 28: Motions and Text Objects 76
Remarks 76
Examples 76
Changing the contents of a string or parameter list 76
Chapter 29: Movement 77
Examples 77
Searching 77
Jumping to characters 77
Searching for strings 77
Basic Motion 77
Remarks 77
Arrows 78
Basic motions 78
Searching For Pattern 80
Navigating to the beginning of a specific word 81
Using Marks to Move Around 82
TLDR 82
Set a mark 82
Jump to a mark 83

www.dbooks.org

Global Marks ... 83
Special marks ... 83
Jump to specific line ... 84
Chapter 30: Normal mode commands ... 86
Syntax ... 86
Remarks ... 86
Examples ... 86
Sorting text ... 86
Normal sorting ... 86
Reverse sorting ... 86
Case insensitive sorting ... 86
Numerical sorting ... 86
Remove duplicates after sorting ... 86
Combining options ... 87
Chapter 31: Normal mode commands (Editing) ... 88
Examples ... 88
Introduction - Quick Note on Normal Mode ... 88
Basic Undo and Redo ... 88
Undo ... 88
Redo ... 88
Repeat the Last Change ... 89
Copy, Cut and Paste ... 89
Registers ... 89
Motions ... 90
Copying and Cutting ... 90
Pasting ... 90
So, How Do I Perform A Really Simple Cut and Paste? ... 91
Completion ... 92
Chapter 32: Plugins ... 94
Examples ... 94
Fugitive Vim ... 94

NERD Tree ... 94
Chapter 33: Regular expressions ... 95
Remarks ... 95
Examples ... 95
Word ... 95
Chapter 34: Regular expressions in Ex Mode ... 96
Examples ... 96
Edit a regular expression in Ex mode ... 96
Chapter 35: Saving ... 99
Examples ... 99
Saving a buffer in a non-existent dir ... 99
Chapter 36: Scrolling ... 100
Examples ... 100
Scrolling downwards ... 100
Scrolling upwards ... 100
Scrolling relative to cursor position ... 100
Chapter 37: Searching in the current buffer ... 102
Examples ... 102
Searching for an arbitrary pattern ... 102
Searching for the word under the cursor ... 102
execute command on lines that contain text ... 103
Chapter 38: Solarized Vim ... 104
Introduction ... 104
Examples ... 104
.vimrc ... 104
Chapter 39: Spell checker ... 105
Examples ... 105
Spell Checking ... 105
Set Word List ... 105
Chapter 40: Split windows ... 106
Syntax ... 106

www.dbooks.org

Remarks ... 106
Examples ... 106
Opening multiple files in splits from the command line ... 106
Horizontally ... 106
Vertically ... 106
Opening a new split window ... 106
Changing the size of a split or vsplit ... 107
Shortcuts ... 107
Close all splits but the current one ... 107
Managing Open Split Windows (Keyboard Shortcuts) ... 107
Move between splits ... 108
Sane split opening ... 108
Chapter 41: Substitution ... 109
Syntax ... 109
Parameters ... 109
Remarks ... 109
Example ... 109
Examples ... 109
Simple replacement ... 109
Quickly refactor the word under the cursor ... 110
Replacement with interactive approval ... 110
Keyboard short-cut to replace currenlty highlighted word ... 110
Chapter 42: The dot operator ... 111
Examples ... 111
Basic Usage ... 111
Set indent ... 111
Chapter 43: Tips and tricks to boost productivity ... 113
Syntax ... 113
Remarks ... 113
Examples ... 113
Quick "throwaway" macros ... 113
Using the path completion feature inside Vim ... 113

Turn On Relative Line Numbers ... 114
Viewing line numbers ... 115
Mappings for exiting Insert mode ... 115
jk ... 115
Caps Lock ... 116
Linux ... 116
Windows ... 116
macOS ... 116
How to know the directory and/or the path of the file you are editing ... 117
Search within a function block ... 117
Copy, move or delete found line ... 118
Write a file if you forget to `sudo` before starting vim ... 119
Automatically reload vimrc upon save ... 119
Command line completion ... 119
Chapter 44: Useful configurations that can be put in .vimrc ... 121
Syntax ... 121
Examples ... 121
Move up/down displayed lines when wrapping ... 121
Enable Mouse Interaction ... 121
Configure the default register to be used as system clipboard ... 121
Chapter 45: Using ex from the command line ... 123
Examples ... 123
Substitution from the command line ... 123
Chapter 46: Using Python for Vim scripting ... 124
Syntax ... 124
Examples ... 124
Check Python version in Vim ... 124
Execute Vim normal mode commands through Python statement ... 124
Executing multi-line Python code ... 124
Chapter 47: vglobal: Execute commands on lines that do not match globally ... 126
Introduction ... 126
Examples ... 126

www.dbooks.org

:v/pattern/d 126
Chapter 48: Vim Options 128
Syntax 128
Remarks 128
Examples 128
Set 128
Indentation 128
Width 128
Spaces 128
Tabs 129
Automatic Indentation 129
Instruction descriptions 129
Invisible characters 129
Show or hide invisible characters 129
Default symbol characters 130
Customize symbols 130
Chapter 49: Vim Registers 131
Parameters 131
Examples 131
Delete a range of lines into a named register 131
Paste the filename while in insert mode using the filename register 132
Copy/paste between Vim and system clipboard 132
Append to a register 132
Chapter 50: Vim Resources 133
Remarks 133
Examples 133
Learning Vimscript the Hard Way 133
Chapter 51: Vim Text Objects 134
Examples 134
Select a word without surrounding white space 134
Select a word with surrounding white space 134

Select text inside a tag ... 134
Chapter 52: Vimscript ... 136
Remarks ... 136
Examples ... 136
Hello World ... 136
Using Normal Mode Commands in Vimscript ... 136
Chapter 53: Whitespace ... 137
Introduction ... 137
Remarks ... 137
Examples ... 137
Delete trailing spaces in a file ... 137
Delete blank lines in a file ... 137
Convert tabs to spaces and spaces to tabs ... 138
Credits ... 139

www.dbooks.org

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: vim

It is an unofficial and free vim ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official vim.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with vim

Remarks

Vim (or "Vi IMproved") is a console-based multi-mode (*modal*) text editor. It is widely used and available by default on all Unix, Linux, and Apple OS X systems. Vim has a large active community and a wide user base. The editor supports all popular programming languages, and many plugins are available to extend its features.

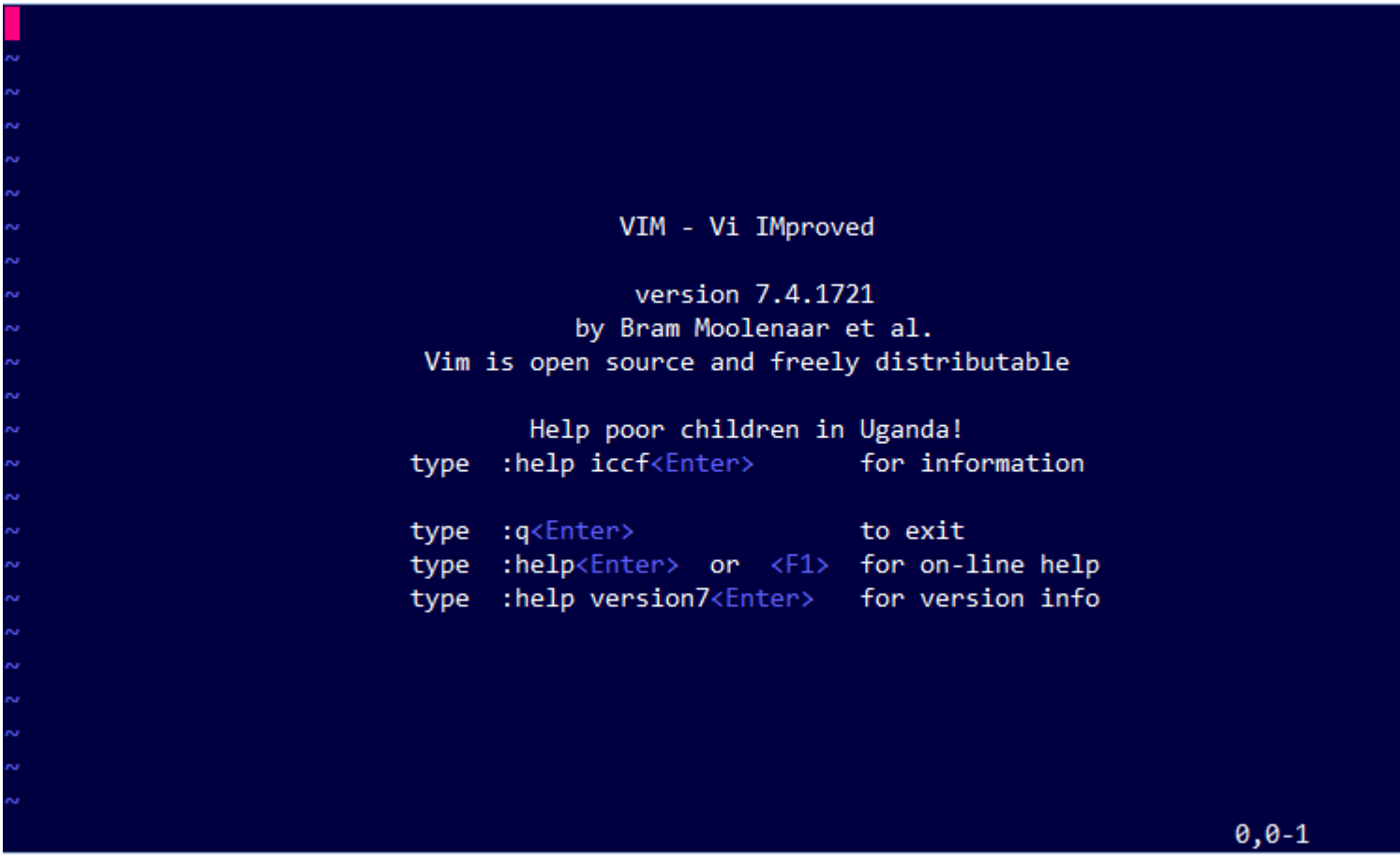
Developers like the editor for its speed, many configuration options, and powerful expression based editing. In "command" mode the editor is controlled by keyboard commands, so the user is not distracted by a GUI or mouse pointer.

Vim is based on the earlier Unix "vi" editor created in the seventies and it has been in continuous development since 1991. With macros and plugins the editor offers most of the features of a modern IDE. It is also uniquely capable of processing large amounts of text with its scripting language (vimscript) and regular expressions.

Main Topics:

- installation
- editing modes
- navigation
- basic editing
- advanced editing
- configuration
- plugins
- vimscript
- macros
- community
- related projects

www.dbooks.org



Versions

Version	Release Date
8.0	2016-09-12
7.4	2013-08-10
7.3	2010-08-15
7.2	2008-08-09
7.1	2007-05-12
7.0	2006-05-07
6.0	2001-09-26
5.0	1998-02-19
4.0	1996-05-29
3.0	1994-08-12
2.0	1993-12-14

Version	Release Date
1.14	1991-11-02

Examples

Installation

The Vim on your machine—if there is one—is very likely to be a "small" build that lacks useful features like clipboard support, syntax highlighting or even the ability to use plugins.

This is not a problem if all you need is a quick way to edit config files but you will soon hit a number of walls if you intend to make Vim your main editor.

It is therefore generally recommended to install a complete build.

Installation on Linux/BSD

On those systems, the trick is simply to install the GUI version which comes with both a `gvim` command for starting the GUI and a `vim` command for starting the TUI.

Arch and Arch-based distributions

```
$ sudo pacman -R vim
$ sudo pacman -S gvim
```

Debian and Debian-based distributions

```
$ sudo apt-get update
$ sudo apt-get install vim-gtk
```

Gentoo and Gentoo-based distributions

```
$ sudo emerge --sync
$ sudo emerge app-editors/gvim
```

RedHat and RedHat-based distributions

```
$ sudo yum check-update
$ sudo yum install vim-X11
```

Fedora

```
$ sudo dnf check-update
$ sudo dnf install vim-X11
```


www.dbooks.org

Slackware and Slackware-based distributions

```
$ sudo slackpkg update
$ sudo slackpkg install-new vim-gvim
```

OpenBSD and OpenBSD-based distributions

```
$ sudo pkg_add vim-x11
```

FreeBSD and FreeBSD-based distributions

```
$ sudo pkg install editors/vim
```

Installation on Mac OS X

The strategy is similar to Mac OS X: we install the GUI version to get both the GUI and the TUI. In the end, we should be able to:

- double-click the MacVim icon in the Finder,
- click on the MacVim icon in the Dock,
- issue `$ mvim` in the shell to open the MacVim GUI,
- issue `$ mvim -v` in the shell to open the MacVim TUI.

Regular install

Download and install an official snapshot like you would with any other Mac OS X application.

Place the `mvim` script that comes bundled with MacVim somewhere in your `$PATH`.

Package manager

MacPorts:

```
$ sudo port selfupdate
$ sudo port install macvim
```

Homebrew:

```
$ brew install macvim
```

To make MacVim the default console Vim:

```
$ brew install macvim --with-override-system-vim
```

Installation on Windows

There is no Vim on Windows systems by default. You can download and install Vim from the Tuxproject site for more up-to-date and complete builds or you can download and install Vim from the official Vim site.

Chocolatey

```
> choco install vim
```

Building Vim from source

If the methods above don't suit your needs it is still possible to build Vim yourself, with *only* the options you need.

This topic will be discussed in its own section (currently in draft).

Exiting Vim

In order to exit Vim, first make sure you are in *Normal* mode by pressing `Esc`.

- `:q Enter` (will prevent you from exiting if you have unsaved changes - short for :quit)

To *discard* changes and exit Vim:

- `:q! Enter` to force exit and discard changes (short for `:quit!`, not to be confused with `:!q`),
- `ZQ` is a shortcut that does the same as `:q!`,
- `:cq Enter` quit and return error (discard all changes so the compiler will not recompile this file)

To *save* changes and exit Vim:

- `:wq Enter` (shorthand for `:write` and `:quit`),
- `:x Enter` (same as `:wq`, but will not write if the file was not changed),
- `ZZ` is a shortcut that does the same as `:x` (Save workspace and quit the editor),
- `:[range]wq! Enter` (write the lines in [range])

To close multiple buffers at once (even in multiple windows and/or tabs), append the letter `a` to any of the *Commands* above (the ones starting with `:`). For example, to write and quit all windows you can use:

- `:wqa Enter` or
- `:xa Enter` — Write all changed buffers and exit Vim. If there are buffers without a file name, which are readonly or which cannot be written for another reason, Vim will not quit
- `:xa! Enter` — Write all changed buffers, even the ones that are readonly, and exit Vim. If there are buffers without a file name or which cannot be written for another reason, Vim will not quit
- `:qa Enter` — try to quit, but stop if there are any unsaved files;
- `:qa! Enter` — quit *without saving* (discard changes in *any* unsaved files)

If you have opened Vim without specifying a file and you want to save that file before exiting, you

www.dbooks.org

will receive `E32: No file name` message. You can save your file and quit using:

- `:wq filename Enter` or;
- `:x filename Enter`

Explanation:

The `:` keystroke actually opens *Command* mode. The command `q` is an abbreviation of `quit`, `w`, of `write` and `x`, of `exit` (you can also type `:quit`, `:write` and `:exit` if you want). Shortcuts *not* starting with `:` such as `ZZ` and `ZQ` refer to *Normal* mode key mappings. You can think of them as shortcuts.

The `!` keystroke is sometimes used at the end of a command to force its execution, which allows to discard changes in the case of `:q!`. Placing the `!` at the beginning of the command has a different meaning. For example, one can mistype `:!q` instead of `:q!` and vim would terminate with a 127 error.

An easy way to remember this is to think of `!` as a way of insisting on executing something. Just like when you write: "I want to quit!"

Interactive Vim Tutorials (such as vimtutor)

`vimtutor` is an interactive tutorial covering the most basic aspects of text editing.

On UNIX-like system, you can start the tutorial with:

```
$ vimtutor
```

On Windows, “Vim tutor” can be found in the “Vim 7.x” directory under “All Programs” in the Windows menu.

See `:help vimtutor` for further details.

Other interactive tutorials include these browser-based ones:

- Vim Adventures – An interactive game version of vimtutor available on the web. Only the first few levels are free.
- Open Vim – An interactive terminal which teaches you the basic commands with feedback.
- Vim Genius – Another interactive terminal which also includes intermediate and advanced lessons including macros and arglist.

Saving a read-only file edited in Vim

Sometimes, we may open a file which we do not have permission to write in Vim without using `sudo`.

Use this command to save a read-only file edited in Vim.

```
:w !sudo tee > /dev/null %
```

Which you could map to `:w!!` in your `.vimrc`:

```
cmap w!! w !sudo tee > /dev/null %
```

You will be presented a prompt as shown in the image.



Press `O` and the file will be saved. It remains open in vi/vim for more editing or reading and you can exit normally by typing `:q!` since the file is still open as read-only.

Command Explanation

```
:w ........................... isn't modifying your file in this case,
   ........................... but sends the current buffer contents to
   ........................... a substituted shell command
   !sudo ...................... call the shell 'sudo' command
         tee .................. the output of the vi/vim write command is redirected
                                using the 'tee' command
             > /dev/null ...... throws away the standard output, since we don't need
                                to pass it to other commands
                          % .... expands to the path of the current file
```

Sources:

- Adam Culp's Tech Blog
- Stackoverflow, How does the vim "write with sudo" trick work

Suspending vim

When using `vim` from the command line, you can suspend `vim` and get back to your prompt, without actually quitting `vim`. Hence you will later be able to get back your `vim` session from the same prompt.

When in *Normal mode* (if not, press `esc` to get there), issue either of these commands:

```
:stenter

:susenter

:stopenter

:suspendenter
```

Alternatively, on some systems, when in *Normal* or *Visual* mode, issuing `Ctrl`+`Z` will have the same effect.

www.dbooks.org

Note: If `autowrite` is set, buffers with changes and filenames will be written out. Add a `!` before `enter` to avoid, eg. `:st!enter`.

Later, when you want to return to your `vim` session, if you haven't suspended any other jobs, issuing the following will restore vim as your foreground job.

```
fgenter
```

Otherwise you will need to find your `vim` sessions's job ID by issuing `jobsenter` and then foregrounding the matching jobs `fg %[job ID]enter` eg. `fg %1enter`.

Basics

Run interactive vim tutorials as many times as needed to feel comfortable with the basics.

Vim features several modes, e.g. **normal mode**, **insert mode** and **command-line mode**.

Normal mode is for editing and navigating text. In this mode `h`, `j`, `k` and `l` correspond to the cursor keys ←, ↓, ↑ and →. Most commands in normal mode can be prefixed with a "count", e.g. `3j` moves down 3 lines.

Insert mode is for inserting the text directly, in this mode vim is similar to other more simple text editors. To enter insert mode press `i` in normal mode. To leave it press `<ESC>` (escape key).

Command-line mode is for running more complex commands like saving the file and exiting vim. Press `:` to start the command-line mode. To leave this mode you can also press `<ESC>`. To save the changes to the file use `:w` (or `:write`). To exit vim without saving your changes use `:q!` (or `:quit!`).

These are some of the more useful commands in vim:

Command	Description
`i`	(insert) enters insert mode *before* the current cursor position
`I`	enters insert mode *before* the first printable character of the current line
`a`	(append) enters insert mode *after* the current cursor position
`A`	enters insert mode *after* the last printable character of the current line
`x`	delete character at the current cursor position
`X`	delete character at the left to the current cursor position
`w`	move to next word
`b`	move to previous word
`0`	move to the beginning of line
`$`	move to the end of line

Command	Description
`r`	replace – enters replace mode for one character. The next character you type will replace the character under the cursor.
`R`	enters replace mode indefinitely. Every character you type will replace the character under the cursor and advance the cursor by one.
`s`	substitute – deletes the character at the current cursor position and then enters insert mode
`S`	delete the current line that the cursor is currently on and enter insert mode
`<Esc>`, `<C-c>`	exit insert mode and returns to normal mode
`u`	undo
`<C-r>`	redo
`dd`, `dw`, `dl`, `d$`	cut the current line, from the cursor to next word, or the character, current position to end of current line respectively, note: `D` is the equivalent of `d$`
`cc`, `cw`, `cl`	change the current line, from the cursor to next word, or the character, respectively
`yy`, `yw`, `yl`, `y$`	yank ("copy") the current line, from the cursor to next word, or the character, current position to end of current line respectively
`p`, `P`	put ("paste") after, or before current position, respectively
`o`, `O`	to create a new empty line, after or before the current one and enter insert mode
`:w`	write the current buffer to disk
`:q!`, `ZQ`	quit without writing
`:x`, `:wq`, `ZZ`	write and quit
`:help`	open a window with help file
`:help {subject}`	show help for a specific subject
`qz`	begin recording actions to register `z`, `q` to end recording, `@z` to play back the actions. `z` can be any letter: `q` is often used for convenience. Read more: Macros

What to do in case of a crash

www.dbooks.org

Vim saves all your unsaved edits in a *swap file*, an extra file that gets deleted once the changes are committed by saving. The name of the swap file is usually the name of the file being edited preceded by a `.` and with a `.swp` suffix (you can see it with `:sw`).

So in case your vim process terminates before you've had the chance to save your edits you can recover your work by applying the changes contained in the swap file to your current file by using the command-line option `-r`. For instance if `myFile` is the file you were editing, use:

```
$ vi -r myFile
```

to recover the uncommitted changes.

If a swap file exists, vim should prompt you anyway for recovery options

```
$ vi myFile
E325: ATTENTION
Found a swap file by the name ".myFile.swp"
...
Swap file ".myFile.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (D)elete it, (Q)uit, (A)bort:
```

If you choose (R)ecover then the changes from the `swp` file are applied but the swap file won't be deleted, so don't forget to delete the swap file afterwards if you're satisfied with the recovery.

Read Getting started with vim online: https://riptutorial.com/vim/topic/879/getting-started-with-vim

Chapter 2: :global

Syntax

- `:[<range>]g[lobal]/{<pattern>}/[<command>]`
- `:[<range>]g[lobal]!/{<pattern>}/[<command>]` (inverted)
- `:[<range>]v[global]/{<pattern>}/[<command>]` (inverted)

Remarks

Vim's "global" command is used to apply an ex command to every line where a regex matches.

Examples

Basic usage of the Global Command

```
:g/Hello/d
```

Will delete every line containing the text "Hello". **Important note**: This is not the normal mode command `d`, this is the ex command `:d`.

You can use the global command to apply normal mode keystrokes instead of ex commands by prepending `normal` or `norm` to the command. For example:

```
:g/Hello/norm dw
```

Will delete the first word from every line that contains the text "Hello".

The global command also supports visual mode and ranges.

Yank every line matching a pattern

The command

```
:g/apples/y A
```

will yank all lines containing the word *apples* into the `a` register, which can be pasted with `"ap`. Any regular expression can be used.

Note the space before the `A`, and the capitalization of the register letter. If a capital letter is used as the yank register, matches will be *appended* to that register. If a lowercase letter is used, only the *last* match will be placed in that register.

Move/collect lines containing key information

www.dbooks.org

a simple yet very useful command:

```
:g/ending/m$
```

moves lines containing `ending` to the end of the buffer.

`m` means move
`$` means end of buffer, while `0` means beginning of buffer.

Read :global online: https://riptutorial.com/vim/topic/3861/-global

Chapter 3: Advantages of vim

Examples

Customization

The advantage of using **vim** over a simple text editor like **notepad** or **gedit** is that it allows the user to customize it's almost everything about itself. If you ever find yourself doing some kind of action over and over again, vim has a multitude of features that will help you do this action faster and easier.

Most of the popular IDEs such as **MS Visual Studio** or **IntelliJ IDEA** provide their users with useful shortcuts and even some amount of customization, but they are usually related to specific actions that are common in a particular context, whereas vim allows one to customize for different situations, without conflicting each other. It might be comfortable to develop c++ programs in Visual Studio and Java in IntelliJ, but you wouldn't write python code in there, and for that there is another IDE of course, but in vim you can pretty much edit whatever language you want without losing the convenience.

There are of course other customizable editors, and I'm not the one to say that vim is the best for everybody. This is a question of personal preference. I don't think someone will argue that **emacs** allows the level of customization inferior to that of vim's (and some would say otherwise), but you really have to try it out for yourself, to find what suits you best.

Some people say, they don't want to spend months learning how to use an editor, just to be able to work in it. But those who do, mostly agree, that it was worth it. For me personally it was never a problem, learning new stuff about vim and getting more efficient with it is just fun. And there is a lot to learn.

Lightweight

Vim is (like GNU Nano or GNU emacs) lightweight. It does not need any kind of graphical interface (like x11, wayland &co).

This makes vim to a system maintainers best friend. You can use it using ssh and, this is really important, on really small devices that do not have some kind of graphical interface.

Programming on and maintaining remote servers got more and more important during the last years and using vim (or emacs) is the best way to do so.

Unlike many IDEs vim brings the capability to work with many kinds of files out of the box and writing your own commands and syntax hl is easy.

And last but not least, a vim user should be able to use vi, that is preinstalled on most UNIX systems.

Read Advantages of vim online: https://riptutorial.com/vim/topic/9653/advantages-of-vim

www.dbooks.org

Chapter 4: Ask to create non-existant directories upon saving a new file

Introduction

If you edit a new file: `vim these/directories/dont/exist/newfile`, you won't be able to save the file as the directory `vim` is trying to save into does not exist.

Examples

Prompt to create directories with :w, or sliently create them with :w!

This code will prompt you to create the directory with `:w`, or just do it with `:w!`:

```
augroup vimrc-auto-mkdir
  autocmd!
  autocmd BufWritePre * call s:auto_mkdir(expand('<afile>:p:h'), v:cmdbang)
  function! s:auto_mkdir(dir, force)
    if !isdirectory(a:dir)
          \   && (a:force
          \       || input("'" . a:dir . "' does not exist. Create? [y/N]") =~? '^y\%[es]$')
      call mkdir(iconv(a:dir, &encoding, &termencoding), 'p')
    endif
  endfunction
augroup END
```

Read Ask to create non-existant directories upon saving a new file online:
https://riptutorial.com/vim/topic/9470/ask-to-create-non-existant-directories-upon-saving-a-new-file

Chapter 5: Autocommands

Remarks

Surround `autocmd` commands

`autocmd` is an additive command, and you probably don't want this behaviour by default.

For example, if you re-source your `.vimrc` a few times while editing it, vim can slow down.

Here's proof:

```
:autocmd BufWritePost * if &diff | diffupdate | endif " update diff after save
:autocmd BufWritePost * if &diff | diffupdate | endif " update diff after save
```

If you now type `:autocmd BufWritePost *`, you'll see **both** lines in the output, not just one. Both get executed.

To avoid this behaviour, surround all your `autocmd`s as follows:

```
if has ('autocmd')       " Remain compatible with vi which doesn't have autocmd
  augroup MyDiffUpdate   " A unique name for the group.  DO NOT use the same name twice!
      autocmd!           " Clears the old autocommands for this group name
      autocmd BufWritePost * if &diff | diffupdate | endif   " Update diff after save
      " ... etc ...
  augroup END
endif
```

Examples

Automatically source .vimrc after saving

Add this to your `$MYVIMRC`:

```
" Source vim configuration file whenever it is saved
if has ('autocmd')          " Remain compatible with earlier versions
 augroup Reload_Vimrc       " Group name.  Always use a unique name!
    autocmd!                " Clear any preexisting autocommands from this group
    autocmd! BufWritePost $MYVIMRC source % | echom "Reloaded " . $MYVIMRC | redraw
    autocmd! BufWritePost $MYGVIMRC if has('gui_running') | so % | echom "Reloaded " .
$MYGVIMRC | endif | redraw
  augroup END
endif " has autocmd
```

Features:

- `echom` tells the user what has happened (and also logs to `:messages`).
- `$MYVIMRC` and `$MYGVIMRC` handle platform-specific names for the configuration files,
- and only match the actual configuration files (ignoring copies in other directories, or a

www.dbooks.org

`fugitive://` diff)

- `has()` will prevent an error if using incompatible versions, such as `vim-tiny`.
- `autocmd!` avoids buildup of multiple identical autocommands if this file is sourced again. (It clears all commands in the named group, so the group name is important.)

Refresh vimdiff views if a file is saved in another window

```
:autocmd BufWritePost * if &diff | diffupdate | endif
```

Read Autocommands online: https://riptutorial.com/vim/topic/4887/autocommands

Chapter 6: Auto-Format Code

Examples

In normal mode:

In normal mode:

`gg` *go to top* `=` then `G`

Read Auto-Format Code online: https://riptutorial.com/vim/topic/7931/auto-format-code

www.dbooks.org

Chapter 7: Buffers

Examples

Managing buffers

You can use buffers to work with multiple files. When you open a file using

```
:e path/to/file
```

it opens in a new buffer (the command means edit the file). New buffer that holds a temporary copy of the file.

You can go to previous buffer with `:bp[rev]` and next buffer with `:bn[ext]`.

You can go to a particular buffer with `b{n}` to go to nth buffer. `b2` goes to second buffer.

Use `:ls` or `:buffers` to list all buffers

Hidden buffers

Moving away from a buffer with unsaved changes will cause this error:

```
E37: No write since last change (add ! to override)
```

You can disable this by adding `set hidden` to your .vimrc file. With this option set your changes will persist in the buffer, but will not be saved to disk.

Switching buffer using part of the filename

To easily select a buffer by filename, you can use:

```
:b [part_of_filename]<Tab><Tab><Tab>...<Enter>
```

The first `Tab` will expand the word to a full filename, and subsequent `Tab` presses will cycle through the list of possible matches.

When multiple matches are available, you can see a list of matches *before* the word expansion by setting this option:

```
:set wildmode=longest:full:list,full
```

This allows you to refine your word if the list of matches is too long, but it requires an extra `Tab` press to perform the expansion. Add the setting to your `$MYVIMRC` if you want to keep it.

Some people like to kick-start this process with a keymap that first lists the buffers:

```
:nnoremap <Leader>b :set nomore <Bar> :ls <Bar> :set more <CR>:b<Space>
```

That makes it easy to select a buffer by its number:

```
:b [buffer_num]
```

Quickly switch to previous buffer, or to any buffer by number

`<C-^>` will switch to and from the previous edited file. On most keyboards `<C-^>` is CTRL-6.

`3<C-^>` will switch to buffer number 3. This is very quick, but only if you know the buffer number.

You can see the buffer numbers from `:ls` or from a plugin such as MiniBufExplorer.

Read Buffers online: https://riptutorial.com/vim/topic/2317/buffers

www.dbooks.org

Chapter 8: Building from vim

Examples

Starting a Build

`:mak[e][!] [arguments]` will start the program referred to by the `makeprg` option. By default, `makeprg` is set to "make," but can be configured to invoke any appropriate program.

All `[arguments]` (can be several) are passed to `makeprg` just as if it had been invoked with `:!{makeprg} [arguments]`.

The output of the invoked program is parsed for errors according to the `'errorformat'` option. If any errors are found, the quickfix window is opened to display them.

`:cnext :cprev` can be used to cycle between errors displayed in the quickfix window. `:cc` will jump to the error under the cursor.

It should be noted that on systems where gnumake is installed and properly configured, there is generally no need to define `&makeprg` to anything but its default value to compile mono-file projects. Thus, in C, C++, Fortran... just type `:make %<` to compile the current file. According the source file is in the current directory, `:!./%<` will execute it. Compilation options can be controlled through `$CFLAGS`, `$CXXFLAGS`, `$LDFLAGS`, etc. Consult the documentation of `make` regarding *implicit rules*.

Read Building from vim online: https://riptutorial.com/vim/topic/3321/building-from-vim

Chapter 9: Command-line ranges

Examples

Absolute line numbers

The following command executes `:command` on lines `23` to `56`:

```
:23,56command
```

NB: Ranges are *inclusive* by default.

Relative line numbers

In the following command the range starts 6 lines above the current line and ends 3 lines below:

```
:-6,+3command
```

Line shortcuts

- `.` represents *the current line* but it can also be omitted entirely.
- `$` represents *the last line*.
- `%` represents *the whole buffer*, it is a shortcut for `1,$`.

The two commands below execute `:command` on every file from the current line to the last line:

```
:.,$command
:,$command
```

The command below executes `:command` on the whole buffer:

```
:%command
```

Marks

The command below executes `:command` on every line from the one containing the `f` manual mark to the one containing the `t` manual mark:

```
:'f,'tcommand
```

Automatic marks can be used too:

```
:'<,'>command    " covers the visual selection
:'{,'}command    " covers the current paragraph
:'[,']command    " covers the last changed text
```


www.dbooks.org

See `:help mark-motions`.

Search

The commands below execute `:command` on every line from the first matching `from` to the first matching `to`:

```
:/from/,/to/command    " from next 'from' to next 'to'
:?from?,/to/command    " from previous 'from' to next 'to'
:?from?,?to?command    " from previous 'from' to previous 'to'
```

See `:help search-commands`.

Line offsets

Line offsets can be used to adjust the start and end lines:

```
:/foo/-,/bar/+4command    " from the line above next 'foo' to 4 lines below next 'bar'
```

See `:help search-offset`.

Mixed ranges

It's possible to combine all of the above into expressive ranges:

```
:1267,/foo/-2command
:'{,command
:'f,$command
```

Be creative and don't forget to read `:help cmdline-ranges`.

Read Command-line ranges online: https://riptutorial.com/vim/topic/3383/command-line-ranges

Chapter 10: Configuring Vim

Examples

The vimrc file

The `.vimrc` file (pronounced Vim-wreck) is a Vim configuration file. It holds commands that will be executed by Vim every time it starts.

By default the file is empty or non-existent; you can use it to customize your Vim environment.

To find out where Vim expects the vimrc file to be stored, open Vim and run:

```
:echo $MYVIMRC
```

Unix: on a Unix system such as Mac or Linux your vimrc will be called `.vimrc` and usually be located in your home directory (`$HOME/.vimrc`).

Windows: on Windows it will be called `_vimrc` and located in your home directory (`%HOMEPATH%/_vimrc`).

On startup, Vim will search in multiple places for a vimrc file. The first that exists is used, the others are ignored. For a full reference see the `:h $MYVIMRC` documentation article.

Which options can I use?

If you don't know which options you should use, you may be interested in the `:options` command.

This will open a split with all Vim options listed and with their current value displayed. There are 26 sections to display all options you can try.

e.g.

```
4 displaying text

scroll    number of lines to scroll for CTRL-U and CTRL-D
    (local to window)
     set scr=20
scrolloff    number of screen lines to show around the cursor
     set so=5
wrap    long lines wrap
     set nowrap    wrap

...
```

On a value line (e.g. `set nowrap`) you can press `CR` to toggle the value (if it's a binary value). On an option line (e.g. `wrap long line wrap`) you can press `CR` to access the documentation for this option.

www.dbooks.org

Files and directories

Whatever you do to customize Vim, it should NEVER happen outside of `$HOME`:

- on Linux, BSD and Cygwin, `$HOME` is usually `/home/username/`,
- on Mac OS X, `$HOME` is `/Users/username/`,
- on Windows, `$HOME` is usually `C:\Users\username\`.

The canonical location for your `vimrc` and your `vim` directory is at the root of that `$HOME` directory:

- on Unix-like systems

```
$HOME/.vimrc       <-- the file
$HOME/.vim/        <-- the directory
```

- on Windows

```
$HOME\_vimrc       <-- the file
$HOME\vimfiles\    <-- the directory
```

The layout above is guaranteed to work, now and in the future.

Vim 7.4 made it possible to keep your lovely `vimrc` **inside** your `vim` directory. It is really a good idea, if only because it makes it easier to move your config around.

If you use 7.4 exclusively, the following will be enough:

- on Unix-like systems

```
$HOME/.vim/vimrc
```

- on Windows

```
$HOME\vimfiles\vimrc
```

If you want the benefits of a self-contained `vim/` but use both 7.4 and an older version, or only an older version, the simplest, future-proof, solution is to put this line *and only this line:*

```
runtime vimrc
```

in this file:

- on Unix-like systems

```
$HOME/.vimrc
```

- on Windows

```
$HOME\_vimrc
```

and do your configuration in `$HOME/.vim/vimrc` or `$HOME\vimfiles\vimrc`.

Options

There are three kinds of options:

- boolean options,
- string options,
- number options.

To check the value of an option,

- use `:set option?` to check the value of an option,
- use `:verbose set option?` to also see where it was last set.

Setting boolean options

```
set booloption      " Set booloption.
set nobooloption    " Unset booloption.

set booloption!     " Toggle booloption.

set booloption&     " Reset booloption to its default value.
```

Setting string options

```
set stroption=baz   " baz

set stroption+=buzz " baz,buzz
set stroption^=fizz " fizz,baz,buzz
set stroption-=baz  " fizz,buzz

set stroption=      " Unset stroption.

set stroption&      " Reset stroption to its default value.
```

Setting number options

```
set numoption=1     " 1

set numoption+=2    " 1 + 2 == 3
set numoption-=1    " 3 - 1 == 2
set numoption^=8    " 2 * 8 == 16
```

Using an expression as value

www.dbooks.org

- using concatenation:

```
execute "set stroption=" . my_variable
```

- using `:let`:

```
let &stroption = my_variable
```

See `:help :set` and `:help :let`.

Mappings

- Don't put comments after mappings, it will break things.
- Use `:map <F6>` to see what is mapped to `<F6>` and in which mode.
- Use `:verbose map <F6>` to also see where it was last mapped.
- `:map` and `:map!` are too generic. Use `:n[nore]map` for normal mode mappings, `:i[nore]map` for insert mode, `:x[nore]map` for visual mode, etc.

Recursive mappings

Use *recursive* mappings **only** if you intend to use other mappings in your mappings:

```
nnoremap b     B
nmap     <key> db
```

In this example, `b` is made to work like `B` in normal mode. Since we use `b` in a *recursive* mapping, pressing `<key>` will work like `dB`, not like `db`.

Non-recursive mappings

Use *non-recursive* mappings **only** if you intend to use default commands in your mappings, which is almost always what you want:

```
nnoremap <key> db
```

In this example, we use `b` in a *non-recursive* mapping so pressing key will always work like `db`, whether we remapped `b` or not.

Executing a command from a mapping

```
nnoremap <key> :MyCommand<CR>
```

Executing multiple commands from a mapping

```
nnoremap <key> :MyCommand <bar> MyOtherCommand <bar> SomeCommand<CR>
```

Calling a function from a mapping

```
nnoremap <key> :call SomeFunction()<CR>
```

Mapping a `<Plug>`mapping

```
map <key> <Plug>name_of_mapping
```

See `:help map-commands`, `:help key-notation` and `:help <plug>`.

see Key Mappings in Vim for futher read

Variables

Like most scripting languages, vimscript has variables.

One can define a variable with the `:let` command:

```
let variable = value
```

and delete it with `:unlet`:

```
unlet variable
```

In Vim, variables can be scoped by prepending a single letter and a colon to their name. Plugin authors use that feature to mimic options:

```
let g:plugin_variable = 1
```

See `:help internal-variables`.

Commands

- Don't forget the bang to allow Vim to overwrite that command next time you reload your vimrc.
- Custom commands must start with an uppercase character.

Examples

```
command! MyCommand call SomeFunction()
command! MyOtherCommand command | Command | command
```

- See `:help user-commands`.

www.dbooks.org

Functions

- Don't forget the bang to allow Vim to overwrite that function next time you reload the script where the function is defined.
- Custom functions must start either with an uppercase character (global functions), or with `s:` (script local functions), or they must be prefixed with the name associated to the autoload plugin where they are defined (e.g. in `{&rtp}/autoload/foo/bar.vim` we could define `foo#bar#functionname()`).
- To be able to use the parameters in the function, use `a:parameter_name`. Variadic functions can be defined with the ellipsis `...`, to access the parameters use `a:000` (list of all parameters), or `a:0` (number of parameters equal to `len(a:000)`), `a:1` first unnamed parameters, and so on.
- Functions can be called like so: `:call MyFunction(param1, param2)`
- Every line in a function implicitly begins with a `:`, thus all the commands are colon commands
- To prevent the function from continuing its execution in case of error, it's best to annotate the function signature with `abort`

Example

```
function! MyFunction(foo, bar, ... ) abort
    return a:foo . a:bar . (a:0 > 0 ? a:1 : '')
endfunction
```

Script functions

If you only plan on using your function in the file where it's defined (either because you've broken a bigger function in smaller parts, or because you'll use it in a command, a mapping, ...), you can prefix it with `s:`, avoiding littering your global namespace with useless internal functions:

```
function! s:my_private_function() " note we don't need to capitalize the first letter this
time
    echo "Hi!"
endfunction
```

Using s:functions from mappings

If your script local function is going to be used in a mapping, you need to reference it using the special `<SID>` prefix:

```
nnoremap <your-mapping-key> :call <SID>my_private_function()<CR>
```

See `:help user-functions`.

Note however, that since Vim 7, it's considered a best practice to define mappings abbreviations, commands and menus in (ft)plugins, and defining functions in autoload plugins -- except the functions the plugins need to use when they're loaded. This means that nowadays the need to call

scripts local functions from mappings is not as pertinent as what it used to be.

Autocommand groups

- Autocommand groups are good for organization but they can be useful for debugging too. Think of them as small namespaces that you can enable/disable at will.

Example

```
augroup MyGroup
    " Clear the autocmds of the current group to prevent them from piling
    " up each time you reload your vimrc.
    autocmd!

    " These autocmds are fired after the filetype of a buffer is defined to
    " 'foo'. Don't forget to use 'setlocal' (for options) and '<buffer>'
    " (for mappings) to prevent your settings to leak in other buffers with
    " a different filetype.
    autocmd FileType foo setlocal bar=baz
    autocmd FileType foo nnoremap <buffer> <key> :command<CR>

    " This autocmd calls 'MyFunction()' everytime Vim tries to create/edit
    " a buffer tied to a file in /'path/to/project/**/'.
    autocmd BufNew,BufEnter /path/to/project/**/* call MyFunction()
augroup END
```

See `:help autocommand`.

Conditionals

```
if v:version >= 704
    " Do something if Vim is the right version.
endif

if has('patch666')
    " Do something if Vim has the right patch-level.
endif

if has('feature')
    " Do something if Vim is built with 'feature'.
endif
```

See `:help has-patch` and `:help feature-list`.

Setting Options

Commonly you would use `:set` to set options to your liking in your `.vimrc`. There are many options that can be changed.

For example, in order to use spaces for indentation:

```
:set expandtab
```


www.dbooks.org

```
:set shiftwidth=4
:set softtabstop=4
```

Syntax Highlighting

Switch syntax highlighting on, when the terminal has colors

```
if &t_Co > 2 || has("gui_running")
    syntax on
end
```

Show trailing whitespace and tabs. Showing tabs can be especially useful when looking for errors in Makefiles.

```
set list listchars=tab:\|_,trail:.
highlight SpecialKey ctermfg=DarkGray
```

Color Schemes

Vim comes with several pre-installed color schemes. In Linux, the color schemes that come with Vim are stored in `/usr/share/vim/vim74/colors/` (where 74 is your version number, sans periods); MacVim stores them in `/Applications/MacVim.app/Contents/Resources/vim/runtime/colors`.

Changing Color Schemes

The `colorscheme` command switches the current color scheme.

For instance, to set the color scheme to "robokai":

```
:colorscheme robokai
```

The default color scheme is creatively named `default`, so, to return to it use

```
:colorscheme default
```

To view all of the currently installed color schemes, type `:colorscheme` followed by `space` and then either `tab` or `ctrld`.

Installing Color Schemes

User-installed color schemes can be placed in `~/.vim/colors/`. Once a color scheme is added to this directory, it will appear as an option to the `colorscheme` command.

To find new color schemes, there are sites like vimcolors which contain a variety of color schemes. There are also tools like vim.ink and Vivify to aid you in creating your own color schemes, or you

can create them by hand.

Toggle line enumerating

To enable - type:

`:set number` or `:set nu`.

To disable - type:

`:set nonumber` or `:set nonu`.

To enable enumerating *relative* to the cursor location - type:

`:set relativenumber`.

To disable enumerating *relative* to the cursor location - type:

`:set norelativenumber`.

Note: To change whether the current line shows the actual line number or `0`, use the `:set number` and `:set nonumber` commands while the `relativenumber` attribute is active.

Plugins

Vim plugins are addons that can be used to change or enhance functionality of vim.

There is a good list of plugins at vimawesome

Read Configuring Vim online: https://riptutorial.com/vim/topic/2235/configuring-vim

www.dbooks.org

Chapter 11: Converting text files from DOS to UNIX with vi

Remarks

The `^M` character stands for a carriage return in Vim (`<c-m>` or just `<CR>`). Vim displays this character when at least on line in the file uses `LF` line endings. In other words, when Vim consider a file to have `fileformat=unix` but some lines do have carriage returns (`CR`), the carriage returns are displayed as `^M`.

A file that has a single line with `LF` line ending and several lines with `CRLF` line endings is most often created by wrongly editing a file created on a MSDOS based system. For example, by creating a file under an MSDOS operating system, copying it to a UNIX based system, and then prepending a hash-bang sting (e.g. `#!/bin/sh`) using tools on the UNIX based operating system.

Examples

Converting a DOS Text file to a UNIX Text file

Quite often you have a file which was edited within DOS or Windows and you are viewing it under UNIX. This can look like the following when you view the file with vi.

```
First line of file^M
Next Line^M
And another^M
```

If you wish to remove the ^M, it can be that you delete each ^M by hand. Alternatively, in vi after hitting `Esc` you can enter the following at the command mode prompt:

```
:1,$s/^M//g
```

Where ^M is entered with `Ctrl` and `v` together and then `Ctrl` and `m` together.

This executes the command from the first line '1' to the last line '$', the command is to substitute 's' the '^M' for nothing '' and to this globally 'g'.

Using VIm's fileformat

When Vim opens a file with `<CR><NL>` line endings (common on MSDOS based operating systems, also called `CRLF`) it will set `fileformat` to `dos`, you can check what with:

```
:set fileformat?
  fileformat=dos
```

Or just

```
:set ff?
  fileformat=dos
```

To convert it to `<NL>` line endings (common on most UNIX based operating systems, also called `LF`) you can change the `fileformat` setting and Vim will change the buffer.

```
:set ff=unix
```

Read Converting text files from DOS to UNIX with vi online:
https://riptutorial.com/vim/topic/3827/converting-text-files-from-dos-to-unix-with-vi

www.dbooks.org

Chapter 12: Differences between Neovim and Vim

Examples

Configuration Files

In Vim, your configuration file is `~/.vimrc`, with further configuration files in `~/.vim`.

In Neovim, configuration files are located in `~/.config/nvim`. There is also no `~/.nvimrc` file. Instead, use `~/.config/nvim/init.vim`.

You can import your Vim configuration directly into Neovim using this Unix command:

```
ln -s ~/.vimrc ~/.config/nvim/init.vim
```

Read Differences between Neovim and Vim online:
https://riptutorial.com/vim/topic/7848/differences-between-neovim-and-vim

Chapter 13: Easter Eggs

Examples

Help!

For the distressed user, vim provides words of wisdom.

```
:help!
```

When you're feeling down

Problem: Vim users are not always happy.

Solution: Make them happy.

7.4

```
:smile
```

Note: Requires patch version ≥7.4.1005

The Answer

Vim knows The Answer:

```
:help 42
```

Vim will open the `usr_42.txt` document from the manual and show the following text:

> What is the meaning of life, the universe and everything? **42**
>
> Douglas Adams, the only person who knew what this question really was about is now dead, unfortunately. So now you might wonder what the meaning of death is...

Looking for the Holy Grail

Check this out:

```
:help holy-grail
```

Ceci n'est pas une pipe

If you look for the help section of `|` or `bar` : `:h bar` you can see:

```
                                                bar
|                       To screen column [count] in the current line.
```


www.dbooks.org

```
                        exclusive motion.  Ceci n'est pas une pipe.
```

This is a reference to the painting *La trahison des images* by René Magritte.



When a user is getting bored

Search for `:h UserGettingBored`

```
                        *UserGettingBored*
UserGettingBored        When the user presses the same key 42 times.
                        Just kidding! :-)
```

Spoon

Instead of looking for the `fork` help, you can search for the `spoon` help:

```
:h spoon

                                                        fork spoon
For executing external commands fork()/exec() is used when possible, otherwise
system() is used, which is a bit slower.  The output of ":version" includes ...
```

Knights who say Ni!

Check this out:

```
:Ni!
```

Monty Python and the Holy Grail

nunmap

```
:help map-modes
```

```
:nunmap can also be used outside of a monastery.
```

Read Easter Eggs online: https://riptutorial.com/vim/topic/4656/easter-eggs

www.dbooks.org

Chapter 14: Enhanced undo and redo with a undodir

Examples

Configuring your vimrc to use a undodir

Since vim version 7.3 the feature 'persistent_undo' is supported, which makes it possible do undo/redo changes, even after closing vim or restarting your computer.

It's possible to configure it by adding the following to your vimrc, but first create a directory, where your undofiles should be saved. You can create the file anywhere, but I recommend using the ".vim" directory.

```
if has('persistent_undo')         "check if your vim version supports
  set undodir=$HOME/.vim/undo     "directory where the undo files will be stored
  set undofile                    "turn on the feature
endif
```

After adding this to your vimrc and sourcing the vimrc again, you can use the feature by using the basic undo/redo commands

Read Enhanced undo and redo with a undodir online:
https://riptutorial.com/vim/topic/7875/enhanced-undo-and-redo-with-a-undodir

Chapter 15: Exiting Vim

Parameters

Parameter	Details
`:`	Enter command-line mode
`w`	Write
`q`	Quit
`a`	All
`!`	Override

Remarks

Command-line mode is entered through normal mode. You will know you are in command-line mode when there is a `:` in the bottom left corner of your terminal window.

Normal mode is the default mode of vi/vim and can be switched to by pressing the `ESC`.

Vi/Vim have built-in checks to prevent unsaved work from being lost and other useful features. To bypass these checks, use the override `!` in your command.

In Vi/Vim it is possible to have more than one file displayed (in different windows) at the same time. Use `a` to close all the opened windows.

Examples

Exit with save

```
:wq
```

```
ZZ
```

Exit without save

```
:q!
```

Exit forcefully (without save)

```
:q!
```

```
ZQ
```


www.dbooks.org

Exit forcefully (with save)

```
:wq!
```

Exit forcefully from all opened windows (without save)

```
:qa!
```

if multiple files are opened

```
:wqall
```

Exiting multiple files with saving contents

```
:qall!
```

Exiting multiple files without saving contents

Read Exiting Vim online: https://riptutorial.com/vim/topic/5074/exiting-vim

Chapter 16: Extending Vim

Remarks

A plugin is a script or set of scripts that changes Vim's default behavior, either by adding non-existing features or by extending existing features.

Often added "non-existing features" include:

- commenting,
- indentation detection,
- autocompletion,
- fuzzy-matching,
- support for a specific language,
- etc.

Often extended "existing features" include:

- omni-completion,
- text-objects & motions,
- yanking & putting,
- status line,
- search & replace,
- buffer/window/tab page switching,
- folding,
- etc.

Examples

How plugins work

A plugin could present itself as a single file containing 30 lines of vimscript or as 20MB of vimscript/python/ruby/whatever split into many files across a dozen of directories that depends on a number of external tools.

The former is obviously easy to install and manage but the latter could pause quite a challenge.

The principle

The `'runtimepath'` option tells Vim where to look for runtime scripts. The default value makes Vim look for scripts into the following directories *in order*:

- on UNIX-like systems
    - **`$HOME/.vim/`**
    - `$VIM/vimfiles/`

www.dbooks.org

    - `$VIMRUNTIME/`
    - `$VIM/vimfiles/after/`
    - **`$HOME/.vim/after/`**

- on Windows

    - **`$HOME/vimfiles/`**
    - `$VIM/vimfiles/`
    - `$VIMRUNTIME/`
    - `$VIM/vimfiles/after/`
    - **`$HOME/vimfiles/after/`**

Of the directories above, only install plugins into the ones in bold. The others will cause instability for no good reason. Installing a plugin boils down to placing each of its components in the right directory under `$HOME/.vim/` or `$HOME/vimfiles/`.

The manual method

Single file plugin

Put the file under `$HOME/.vim/plugin` or `$HOME/vimfiles/plugin`

This would source the plugin on startup of Vim. Now the user could use everything defined in it. If the plugin however needs activation, the user either has to execute the command themselves whenever they want to use it, or add the command to `.vimrc`

Bundle

A bundle is a directory structure that the plugin uses. It consists of all the files of the plugin under the appropriate sub-directories.

To install such a plugin the sub-directories should be merged with their counterparts in `$HOME/.vim/plugin`. This approach however leads to mixing of the files of different plugins in the same directories and could possibly lead to namespace problems.

Another approach is to copy the entire directory into `$HOME/.vim/bundle`.

When using this approach there should be at least one `.vim` file under the `$HOME/.vim/bundle/autoload` directory. These files would be sourced by vim on startup.

Note: Depending on the operating system of the user the prefix of all paths might be `$HOME/vimfiles`. For more details see How plugins work

VAM

https://github.com/MarcWeber/vim-addon-manager

Vundle

Vundle is a plugin manager for Vim.

Installing Vundle

(Full installation details can be found in the Vundle Quick Start)

1. Install Git and clone Vundle into `~/.vim/bundle/Vundle.vim`.
2. Configure plugins by adding the following to the top of your `.vimrc`, adding or removing plugins as necessary (the plugins in the list are merely for illustration purposes)

```
set nocompatible              " be iMproved, required
filetype off                  " required

" set the runtime path to include Vundle and initialize
set rtp+=~/.vim/bundle/Vundle.vim
call vundle#begin()
" alternatively, pass a path where Vundle should install plugins
"call vundle#begin('~/some/path/here')

" let Vundle manage Vundle, required
Plugin 'VundleVim/Vundle.vim'

" All of your Plugins must be added before the following line
call vundle#end()            " required
filetype plugin indent on    " required
" To ignore plugin indent changes, instead use:
"filetype plugin on

"place non-Plugin stuff after this line
```

3. Install Plugins: by launching Vim and running `:PluginInstall`.

Supported Plugin Formats

The following are examples of different formats supported. Keep Plugin commands between `vundle#begin` and `vundle#end`.

Plugin Location	Usage
plugin on GitHub	`Plugin 'tpope/vim-fugitive'`
plugin from http://vim-scripts.org/vim/scripts.html	`Plugin 'L9'`
Git plugin not hosted on GitHub	`Plugin 'git://git.wincent.com/command-t.git'`
git repos on your local machine (i.e. when working	`Plugin 'file:///home/gmarik/path/to/plugin'`

www.dbooks.org

Plugin Location	Usage
on your own plugin)	
The sparkup vim script is in a subdirectory of this repo called vim. Pass the path to set the runtimepath properly.	`Plugin 'rstacruz/sparkup', {'rtp': 'vim/'}`
Install L9 and avoid a Naming conflict if you've already installed a different version somewhere else.	`Plugin 'ascenator/L9', {'name': 'newL9'}`

Working on a shared account, for example, on cluster head node can raise issues from the point of disk usage by `.vim` directory. There are a couple of packages which take considerable amount of disk space, for example YCM. So please choose your `Vundle` plugin directory wisely, and its very easy to do so by setting `rtp`. And also if you are planning to install any vim plugin, don't directly do `git clone` in the `bundle` directory. Use the Vundle way.

The future: packages

See `:help packages`.

Pathogen

vim-pathogen is a `runtimepath` manager created by Tim Pope to make it easy to install plugins and runtime files in their own private directories.

Installing Pathogen

1. Put pathogen in `~/.vim/bundle` (here with Git, but it's not mandatory):

```
git clone https://github.com/tpope/vim-pathogen.git
```

2. Add the following lines to the top of your `.vimrc`:

```
" enable vim-pathogen
runtime bundle/vim-pathogen/autoload/pathogen.vim
execute pathogen#infect()
```

- the `runtime` directive specifies the path to the autoload script of `vim-pathogen`;
- `execute pathogen#infect()` initiates it.

Once initiated, Pathogen will automatically start a sweep through the folders in `~/.vim/bundle` and load the plugin from each of them.

Using Pathogen

1. Put the top-level directory of your plugin in `~/.vim/bundle/` to make it available next time you start Vim.
2. Run `:Helptags` to index your new plugin's documentation.

Benefits

- Each plugin resides in its own directory under `~/.vim/bundle/`.
- Your `.vimrc` stays clean from the configuration needed to load plugins.

The effort needed to "manage" a plugin is thus reduced to:

- **put** its top-level directory under `~/.vim/bundle/` to *install* it,
- **replace** its top-level directory to *update* it,
- **delete** its top-level directory to *uninstall* it.

How you perform those three actions (manually, via an automation tool, with Git/Svn/Hg/whatever…) is completely up to you.

Read Extending Vim online: https://riptutorial.com/vim/topic/3659/extending-vim

www.dbooks.org

Chapter 17: Filetype plugins

Examples

Where to put custom filetype plugins?

Filetype plugins for `foo` filetype are sourced in that order:

1. `$HOME/.vim/ftplugin/foo.vim`. Be careful with what you put in that file as it may be overridden by `$VIMRUNTIME/ftplugin/foo.vim` -- under windows, it'll be instead `$HOME/vimfiles/ftplugin/foo.vim`

2. `$VIMRUNTIME/ftplugin/foo.vim`. Like everything under `$VIMRUNTIME`, this file should not be changed.

3. `$HOME/.vim/after/ftplugin/foo.vim`. This file comes last so it's the ideal place for *your* filetype-specific settings.

Read Filetype plugins online: https://riptutorial.com/vim/topic/7734/filetype-plugins

Chapter 18: Find and Replace

Examples

Substitute Command

This command:

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

substitutes each occurrence of `foo` with `bar` on the current line.

```
fool around with a foodie
```

becomes

```
barl around with a bardie
```

If you leave off the last `/g`, it will only replace the first occurence on the line. For example,

```
:s/foo/bar
```

On the previous line would become

```
barl around with a foodie
```

This command:

```
:5,10s/foo/bar/g
```

performs the same substitution in lines 5 through 10.

This command

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

performs the same substitution from line 5 to the end of the file.

This command:

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

performs the same substitution on the whole buffer.

If you are in visual mode and hit the colon, the symbol `'<,'>` will appear. You can then do this

www.dbooks.org

```
:'<,'>s/foo/bar/g
```

and have the substitution occur within your visual mode selection.

This command:

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

is equivalent to the command above but asks for confirmation on each occurence thanks to the `/c` flag (for "confirmation").

See `:help :s` and `:help :s_flags`.

See also this section on command-line ranges.

Replace with or without Regular Expressions

This substitute command can use Regular Expressions and will match any instance of `foo` *followed by any(one) character* since the period `.` in Regular Expressions matches any character, hence the following command will match all instances of `foo` followed by any character in the current line.

```
:s/foo./bar/g
```

```
  1 fooing fooes fool foobar foosup
```

will become

```
  1 barng bars bar barar barup
```

If you want to match the literal `.` period you can escape it in the search field with a backslash `\`.

```
:s/foo\./bar/g
```

```
  1 fooing fooes foo.l foo.bar foosup
```

will become

```
  1 fooing fooes barl barbar foosup
```

Or disable all pattern matching by following the `s` command with `no`.

```
:sno/foo./bar/g
```

```
  1 fooing fooes foo.l foo.bar foosup
```

will raise an error

```
E486: Pattern not found
```

Read Find and Replace online: https://riptutorial.com/vim/topic/3533/find-and-replace

www.dbooks.org

Chapter 19: Folding

Remarks

Folding causes multiple lines of text to be collapsed and displayed as a single line. It is useful for hiding portions of a document considered unimportant for the current task. Folding is purely a visual change to the document: the folded lines are still present, unchanged.

A fold is persistent. Once created, a fold can be opened and closed without needing to re-create it. When closed, folds can be moved over or yanked and put as if they were a single line, even though the underlying operation will operate on all of the text underneath the fold

Examples

Configuring the Fold Method

`:set foldmethod={method}` sets the fold method for the current window. This determines how folds are manipulated within that window. Valid options for "method" are:

- `manual` (folds are manually created and destroyed by the user)
- `indent` (folds are created for lines of equal indentation)
- `marker` (substring markers are used to indicate the beginning and end of a fold)
- `syntax` (syntax highlighting items define folds)
- `expr` (a Vimscript expression is evaluated per line to define its fold level)
- `diff` (text change isn't changed in a diff view is folded)

The default is `manual`.

Creating a Fold Manually

- `zf{motion}` creates a fold that covers the text that "motion" would cover.
- `{count}zF` creates a fold that covers "count" lines.
- `{range}fo[ld]` creates a fold for the lines in the provided range.

All three commands are valid only when `foldmethod` is set to `manual` or `marker`. In the case of the former fold method, the newly-created folds are closed immediately.

Opening, Closing and Toggling Folds

- `zo` opens a fold underneath the cursor.
- `zO` opens all folds underneath the cursor, recursively.
- `zc` closes a fold underneath the cursor.
- `zC` closes all folds underneath the cursor, recursively.
- `za` toggles a fold under the cursor (a closed fold is opened, an opened fold is closed).
- `zM` closes all folds in the buffer.

- `zR` opens all folds in the buffer.
- `zm` closes a level of fold in the buffer.
- `zr` opens a level of fold in the buffer.

Showing the Line Containing the Cursor

`zv` will ensure the line containing the cursor is not folded. The minimum number of folds required to expose the cursor line will be opened.

Folding C blocks

This is our buffer:

```
void write_buffer(size_t size, char ** buffer)
{
    char * buf = *buffer;
    size_t iter;
    for(iter = 0; iter < size; iter++)
    {
        putc(*(buf + iter));
    }
}
```

The cursor is at [1][1] ([line][col]). Move the cursor to the curl bracket of the for loop: `/for<Enter>j` cursor is [6][2].

Now enter `zf%` (create folding, move to matching bracket). You have successfully create the first folding.

Now enter `:2<Enter>_`, the cursor is now at [2][1] and `zf%`: the complete function body is folded.

You are able to open all foldings you just created using `zO` and re-close them using `zC`.

Read Folding online: https://riptutorial.com/vim/topic/3791/folding

www.dbooks.org

Chapter 20: Get :help (using Vim's built-in manual)

Introduction

Vim's built-in manual is the authoritative source of information and documentation on every Vim feature, including configurations, built-in functions, and even Vimscript. While not the most beginner-friendly interface, if you know how to look through it, you can find what you need.

Start searching by executing `:help`, `:help [subject]`, or `:help :help`.

Syntax

- `:h[elp] [keyword]`

Parameters

Parameters	Details
`keyword`	Configuration, function name, or any other keyword with significance to Vim. Keywords with a leading colon `:` search for Vim commands; e.g `:help :split` yields the window-splitting command, and `:help split` yields the Vimscript function `split()` .

Examples

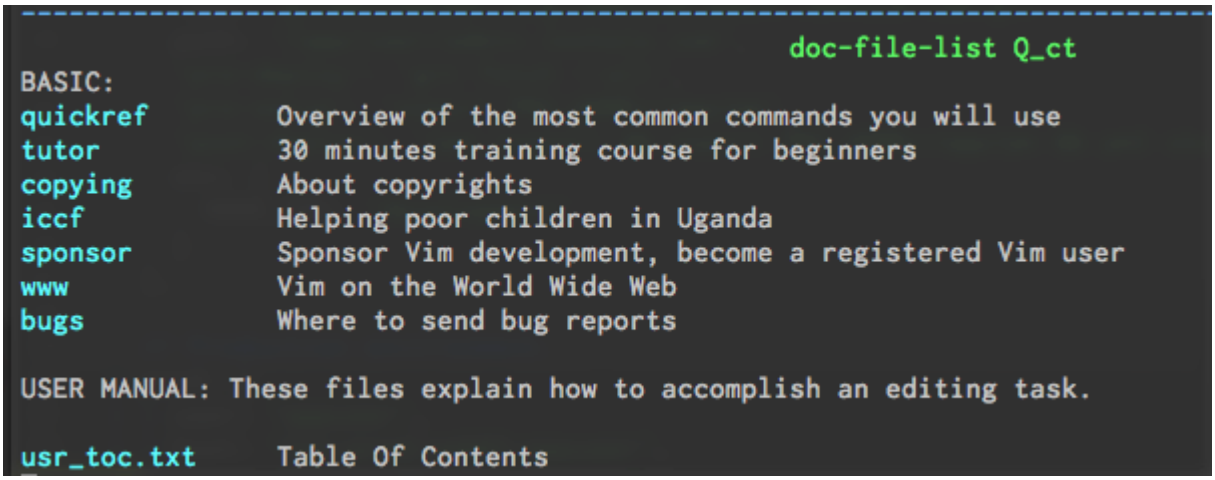
Getting started / Navigating help files

From anywhere in Vim, execute `:help :help`. This will open a horizontally split window with the manual page for the `:help` command. `:help` by itself will take you to the Table of Contents for the manual itself.

Vim's help files are navigable like regular files (you can search for keywords within a file like normal, with `/`), and additionally they are linked together by tags. Jump to the destination of a tag with `CTRL-]`.

Tags are words surrounded by pipe `|` characters. Versions 7.3 and up 'conceal' those pipe characters (`:help conceal`) and highlight them.

For example, the Table of Contents page shows the following. All of the words highlighted in blue are tags and are surrounded by pipe characters. Typing `CTRL-]` with the cursor on `quickref` will take you to a useful page with a list of tags to useful Vim features.



Searching the manual

`:help [subject]` attempts to find the "best" match for the argument you supply. The argument "can include wildcards like `*`, `?` and `[a-z]` (any letter).

You can additionally use Vim's command-line completion with `CTRL+D`:
`:help spli<Ctrl-D>` will display a list of help topics matching the pattern `spli`, including `split()`, and `:split`.

To search for `Ctrl`-based commands, like `Ctrl-V`, type:
`:help ^V` with a literal caret character, or even more specifically,
`:help i_^V` to get help on `Ctrl-V` in insert mode.

As you see, vim has a nomenclature for its help topics. For instance, options are quoted (see `:h 'sw'`), commands start with a colon (see `:h :split`), functions end with empty brackets (see `:h split()`), insert mode mappings start with `i_`, command mode mappings start with `c_`, and so on, except normal mode mappings that have no prefix.

Search term	Help page
`:help textwidth`	Configuration for line-length/text-width
`:help normal`	`:normal`command, to execute normal-mode commands from the command-line
`:help cursor`	Vimscript command to move the cursor around
`:help buffers`	Working with buffers; same as `:help windows`
`:help :buffer`	The `:buffer` command
`:help :vs`	Vertical splitting

Read Get :help (using Vim's built-in manual) online: https://riptutorial.com/vim/topic/8837/get--help--using-vim-s-built-in-manual-

www.dbooks.org

Chapter 21: How to Compile Vim

Examples

Compiling on Ubuntu

To build vim from source on Ubuntu:

1. Get a copy of the source code by downloading from the official Vim repository on GitHub.
2. Get the dependencies by running `$ sudo apt-get build-dep vim-gnome` or similar.
3. Go to the directory of the Vim source code: `cd vim/src`
4. Run `$ ./configure`. You can customize the build (and enable Perl, Python, etc. language integrations) by passing configuration options. See `src/INSTALL` for an overview.
5. Run `$ make`.
6. Finish the installation by running `$ sudo make install`. As your self-compiled Vim is not managed by the package manager, it will be placed in `/usr/local/bin/vim`, instead of `/usr/bin/vim`. So, to run it, you either need to specify the full path, or ensure that `/usr/local/bin` is before `/usr/bin` in your `PATH` (it usually is).
7. (Optional) Remove the distribution-provided version of Vim if you had it installed already: `$ sudo apt-get remove vim vim-runtime gvim`.

To verify the installation, you can run `$ which vim` which should return something like `/usr/local/bin/vim` if the installation was successful.

Read How to Compile Vim online: https://riptutorial.com/vim/topic/3737/how-to-compile-vim

Chapter 22: Indentation

Examples

Indent an entire file using built-in indentention engine

In command mode(Esc) enter `:gg=G` to use Vim's built-in indention engine.

Command Part	Description
gg	start of file
=	indent (when `equalprg` is empty)
G	end of file

You can set `equalprg` in your .vimrc to use a more sophisticated auto-formatting tool.

For example, to use `clang-format` for C/C++ put the following line in your `.vimrc` file:

```
autocmd FileType c,cpp setlocal equalprg=clang-format
```

For other file types, replace `c,cpp` with the filetype you want to format and `clang-format` with your preferred formatting tool for that filetype.

For example:

```
" Use xmllint for indenting XML files. Commented out.
"autocmd FileType xml setlocal equalprg=xmllint\ --format\ --recover\ -\ 2>/dev/null
" Tidy gives more formatting options than xmllint
autocmd FileType xml setlocal equalprg=tidy\ --indent-spaces\ 4\ --indent-attributes\ yes\ --
sort-attributes\ alpha\ --drop-empty-paras\ no\ --vertical-space\ yes\ --wrap\ 80\ -i\ -xml\
2>/dev/null
```

Indent or outdent lines

To indent our outdent the current line in normal mode press the greater than `>` key or the less than `<` twice accordingly. To do the same on multiple lines just add a number beforehand `6>>`

Command	Description
`>>`	indent current line
`<<`	outdent current line
`6>>`	indent next 6 lines

www.dbooks.org

You can also indent using motions. Here are a few useful examples.

Command	Description
`>gg`	indent from current line to first line in file
`>G`	indent from current line to last line in file
`>{`	indent previous paragraph
`>}`	indent next paragraph

In visual mode by pressing the greater than or less than key just once. Note that this causes an exit from visual mode. Then you can use `.` to repeat the edit if you need to and `u` to undo.

Read Indentation online: https://riptutorial.com/vim/topic/6324/indentation

Chapter 23: Inserting text

Examples

Leaving insert mode

Command	Description
`<Esc>`	Leaves insert mode, triggers autocommands and abbreviations
`<C-[>`	Exact synonymous of `<Esc>`
`<C-c>`	Leaves insert mode, doesn't trigger autocommands

Some people like to use a relatively uncommon pair of characters like `jk` as shortcut for `<Esc>` or `<C-[>` which can be hard to reach on some keyboards:

```
inoremap jk <Esc>l
```

Different ways to get into insert mode

Command	Description
`a`	Append text following current cursor position
`A`	Append text at the end of current line
`i`	Insert text before the current cursor position
`I`	Insert text before first non-blank character of current line
`gI`	Insert text in first column of cursor line
`gi`	Insert text at same position where it was left last time in Insert mode
`O`	Open up a new line above the current line and add text there (CAPITAL `O`)
`o`	Open up a new line below the current line and add text there (lowercase `o`)
`s` or `cl`	Delete character under the cursor and switch to insert mode
`S` or `cc`	Delete entire line and switch to Insert mode
`C`	Delete from the cursor position to the end of the line and start insert mode
`c{motion}`	Delete `{motion}` and start insert mode (see Basic Motion)

www.dbooks.org

Insert mode shortcuts

Command	Description
`<C-w>`	Delete word before cursor
`<C-t>`	Indent current line with by one `shiftwidth`
`<C-d>`	Unindent current line with by one `shiftwidth`
`<C-f>`	reindent the line, (move cursor to auto indent position)
`<C-a>`	Insert previously inserted text
`<C-e>`	Insert the character below
`<C-h>`	Delete one character backward
`<C-y>`	Insert the character above
`<C-r>{register}`	Insert the content of `{register}`
`<C-o>{normal mode command}`	execute `{normal mode command}` without leaving insert mode
`<C-n>`	Next autocomplete option for the current word
`<C-p>`	Previous autocomplete option for the current word
`<C-v>`	Insert a character by its ASCII value in decimal
`<C-v>x`	Insert a character by its ASCII value in hexadecimal
`<C-v>u`	Insert a character by its unicode value in hexadecimal
`<C-k>`	Enter a digraph

Running normal commands from insert mode

While in insert mode, press `<C-o>` to temporarily leave insert mode and execute a one-off normal command.

Example

`<C-o>2w` jumps to the second word to the left and returns to insert mode.

Note: Repeating with `.` will only repeat the actions from returning to insert mode

This allows for some useful mappings, e.g.

```
inoremap <C-f> <Right>
inoremap <C-b> <Left>
inoremap <C-a> <C-o>^
inoremap <C-e> <C-o>$
```

Now ctrl+a will put the cursor to the beginning of the line and ctrl+e - to the end of line. These mappings are used by default in `readline`, so might be useful for people who want consistency.

Insert text into multiple lines at once

Press `Ctrl` + `v` to enter into visual block mode.

Use `↑` / `↓` / `j` / `k` to select multiple lines.

Press `Shift` + `i` and start typing what you want.

After you press `Esc`, the text will be inserted into all the lines you selected.

Remember that `Ctrl`+`c` is not 100% equivalent to `Esc` and will not work in this situation!

There are slight variations of `Shift` + `i` that you can press while in visual block mode:

Key	Description
`c` or `s`	Delete selected block and enter insert mode
`C`	Delete selected lines (from cursor until end) and enter insert mode
`R`	Delete selected lines and enter insert mode
`A`	Append to selected lines, with the column at the end of the first line

Also note that pressing `.` after a visual block operation will repeat that operation where the cursor is!

Paste text using terminal "paste" command

If you use the paste command from your terminal emulator program, Vim will interpret the stream of characters as if they were typed. That will cause all kind of undesirable effects, particularly bad indendation.

To fix that, from command mode:

```
:set paste
```

Then move on to insert mode, with `i`, for example. Notice the mode is now `-- INSERT (paste) --`. Now paste with your terminal emulator command, or with the mouse. When finished go to command mode, with `Esc` and run:

www.dbooks.org

```
:set nopaste
```

There is a simpler way, when one wants to paste just once. Put this in your *.vimrc* (or use the plugin *unimpaired.vim*):

```
function! s:setup_paste() abort
  set paste
  augroup unimpaired_paste
    autocmd!
    autocmd InsertLeave *
      \ set nopaste |
      \ autocmd! unimpaired_paste
  augroup end
endfunction

nnoremap <silent> yo :call <SID>setup_paste()<CR>o
nnoremap <silent> yO :call <SID>setup_paste()<CR>O
```

Now one can simply press `yo` to paste code under the cursor, and then `<Esc>` to go back to normal/nopaste mode.

Pasting from a register while in insert mode

While in insert mode, you can use `<C-r>` to paste from a register, which is specified by the next keystroke. `<C-r>"` for example pastes from the unnamed (`"`) register.

See `:help registers`.

Advanced Insertion Commands and Shortcuts

Here is a quick reference for advanced insertion, formatting, and filtering commands/shortcuts.

Command/Shortcut	*Result*
`g + ?` + *m*	Perform rot13 encoding, on movement *m*
n + `ctrl + a`	+*n* to number under cursor
n + `ctrl + x`	-*n* to number under cursor
`g + q`+ *m*	Format lines of movement *m* to fixed width
:*r*ce w	Center lines in range *r* to width *w*
:*r*le i	Left align lines in range *r* with indent *i*
:*r*ri *w*	Right align lines in range *r* to width *w*
!*mc*	Filter lines of movement *m* through command *c*
n!!*c*	Filter *n* lines through command *c*

Command/Shortcut	*Result*
:*r*!*c*	Filter range *r* lines through command *c*

Disable auto-indent to paste code

When pasting text through a terminal emulator, the auto-indent feature *may* destroy the indentation of the pasted text.

For example:

```
function () {
    echo 'foo'
    echo 'bar'
    echo 'baz'
}
```

will be pasted as:

```
function () {
    echo 'foo'
        echo 'bar'
            echo 'baz'
            }
```

In these cases, use the `paste`/`nopaste` option to disable / enable the auto-indent feature:

```
:set paste
:set nopaste
```

Adding to this, there is a simpler approach to the problem: Add the following line in your .vimrc:

```
set pastetoggle=<F3>
```

And if you want to paste as is from the clipboard. Just press `F3` in `insert` mode, and paste. Press `F3` again to exit from the `paste` mode.

Read Inserting text online: https://riptutorial.com/vim/topic/953/inserting-text

www.dbooks.org

Chapter 24: Key Mappings in Vim

Introduction

Updating Vim key mappings allows you to solve two kinds of problems: Re-assigning key commands to letters that are more memorable or accessible, and creating key commands for functions which have none. Here you will learn about the various ways to [re]map key commands, and the context to which they apply *(i.e. vim modes)*

Examples

Basic mapping

map Overview

A key sequence can be re-mapped to another key sequence using one of the `map` variants.

As an example, the following typical `map` will exit *Insert mode* when you press `jk` in quick sequence:

```
:inoremap jk <Esc>
```

map Operator

There are multiple variants of `:map` for different modes.

Commands	Modes
`:map`, `:noremap`, `:unmap`	Normal, Visual and Operator-pending mode
`:map!`, `:noremap!`, `:unmap!`	Insert and Command-line mode
`:nmap`, `:nnoremap`, `:nunmap`	Normal mode
`:imap`, `:inoremap`, `:iunmap`	Insert and Replace mode
`:vmap`, `:vnoremap`, `:vunmap`	Visual and Select mode
`:xmap`, `:xnoremap`, `:xunmap`	Visual mode
`:smap`, `:snoremap`, `:sunmap`	Select mode
`:cmap`, `:cnoremap`, `:cunmap`	Command-line mode
`:omap`, `:onoremap`, `:ounmap`	Operator pending mode

Usually, you should use the `:noremap` variants; it makes the mapping immune to remapping and recursion.

map Command

- You can display all mappings using `:map` (or one of the variations above).
- To display the current mapping for a specific key sequence, use `:map <key>` where `<key>` is a sequence of keys
- Specials keys like `Esc` are mapped using special `<>` notation, like `<Esc>`. For the full list of key codes, see http://vimdoc.sourceforge.net/htmldoc/intro.html#keycodes
- `:nmapclear` - Clear all normal mode maps
- `:nunmap` - Unmap a normal mode map
- You can configure the maximum time between keys of a sequence by changing the `timeout` and `ttimeout` variables

Examples

- `imap jk <Esc>`: typing `jk` in insert mode will bring you back to normal mode
- `nnoremap tt :tabnew<CR>`: typing `tt` in normal mode will open a new tab page
- `nnoremap <C-j> <C-w>j`: typing `<C-j>` in normal mode will make you jump to the window below and to the left
- `vmap <C-c> \cc`: typing `<C-c>` in visual mode will execute `\cc` (NERDCommenter command to comment the line). As this relies on a plugin mapping, you cannot use `:vnoremap` here!

  futher reading here

Map leader key combination

The leader key could be used as a way to create a mapping with a key-binding that can be overridden by the end user.

The leader is the `\` key by default. In order to override it, the end-user would have to execute `:let g:mapleader='somekey(s)'` before defining the mapping.

In a typical scenario, the mapleader is set in the `.vimrc`, and plugins use `<Leader>` in the keybinding part of their mappings to have them customizable.

In the plugin, we would define mappings with:

```
:nnoremap <Leader>a somecomplexaction
```

This would map the `somecomplexaction` action to the `\+a` key combination.

The `a` action without the leader does not change.

It's also possible to use `<Plug>Mappings` to leave more room to customise plugins keybindings.

www.dbooks.org

Illustration of Basic mapping (Handy shortcuts).

In most text editors, the standard shortcut for saving the current document is `Ctrl`+`S` (or `Cmd`+`S` on macOS).

Vim doesn't have this feature by default but this can be mapped to make things easier. Adding the following lines in `.vimrc` file will do the job.

```
nnoremap <c-s> :w<CR>
inoremap <c-s> <c-o>:w<CR>
```

The `nnoremap` command maps `Ctrl`+`s` to `:w` (write current contents to file) command whereas the `inoremap` command maps the `Ctrl`+`S` to `:w` command and returns back to the insert mode (`<c-o>` goes into normal mode for one command and returns to insert mode afterwards, without altering cursor position which other solutions like `<esc>:w<cr>a` cannot ensure).

Similarly,

```
" This is commented, as Ctrl+Z is used in terminal emulators to suspend the ongoing
program/process.
" nnoremap <c-z> :u<CR>

" Thus, Ctrl+Z can be used in Insert mode
inoremap <c-z> <c-o>:u<CR>

" Enable Ctrl+C for copying selected text in Visual mode
vnoremap <c-c> <c-o>:y<CR>
```

PS: However it must be noted that `Ctrl`+`S` may not work as expected while using ssh (or PuTTY). The solution to this is not within the scope of this document, but can be found Here.

Read Key Mappings in Vim online: https://riptutorial.com/vim/topic/3535/key-mappings-in-vim

Chapter 25: Macros

Examples

Recording a macro

One way to create a macro is to *record* it.

Start recording a macro and save it to a register (in this example, we'll use `a`, but it can be any register you could normally yank text to):

```
qa
```

Then run the commands you want to record in the macro (here, we'll surround the contents of a line with `<li>` tags):

```
I<li><ESC>A</li>
```

When we're finished with the commands we want to record in the macro, stop the recording:

```
q
```

Now, any time we want to execute the recorded sequence of commands stored in `a`, use:

```
@a
```

and vim will repeat the recorded sequence.

Next time you would like to repeat the last macro that was used you can double type `@`:

```
@@
```

And as a extra bonus it is good to remember that if you put a number before a command it will repeat it that many times. So, you repeat the macro saved in register `a` 20 times with:

```
20@a
```

Editing a vim macro

Sometimes you will make a mistake with a lengthy macro, but would rather edit it than re-record it entirely. You can do this using the following process:

1. Put the macro on an empty line with `"<register>p`.

   If your macro is saved in register `a`, the command is `"ap`.

www.dbooks.org

2. Edit the macro as needed.

3. Yank the macro into the correct register by moving the cursor to the beginning of the line and using `"<register>y$`.

   You can re-use the original register or use another one. If you want to use register `b`, the command is `"by$`. or by using `"<register>d$` (deletes the unused line)

Recursive Macros

Vim macros can also be recursive. This is useful for when you need to act on every line (or other text object) till the end of the file.

To record a recursive macro, start with an empty register. (A register can be emptied using `q<register>q`.)

Choose a consistent starting point on each line to start and finish.

Before finishing recording, invoke the macro itself as the last command. (This is why the register must be empty: so it'll do nothing, as the macro doesn't exist yet).

Example, given the text:

```
line 1
line 2
line 3
foo bar
more random text
.
.
.
line ???
```

In normal mode, with the cursor on the first line and a empty register `a`, one could record this macro:

```
qaI"<Esc>A"<Esc>j@aq
```

Then with a single invocation of `@a`, all the lines of the file would be now inside double quotes.

What is a macro?

A macro is a series of keystrokes meant to be "played back" by Vim without any delay. Macros can be stored in registers or variables, bound to keys, or executed on the command line.

Here is a simple macro that uppercases the third `word` on a line:

```
0wwgUiw
```

That macro could be *recorded* into register `q`:

```
qq          start recording into register q
0wwgUiw
q           stop recording
```

or saved directly into register `q`:

```
:let @q = '0wwgUiw'
```

to be played back with:

```
@q
```

But it could also be typed directly in the command-line:

```
:normal 0wwgUiw
```

for instant playback via the `:normal` command.

Or put into a variable:

```
:let myvar = '0wwgUiw'
```

to be played back with:

```
@=myvar
```

Or saved as a mapping:

```
nnoremap <key> 0wwgUiw
```

to be played back by pressing `<key>`.

If you want to store a macro for later reuse you can type in insert mode:

```
<C-r>q
```

This inserts the macro in register `q` (in this example: `0wwgUiw`). You can use this output e.g. to define the macro in your `vimrc`:

```
let @q='0wwgUiw'
```

Doing so the register `q` is initialized with this macro every time you start vim.

Record and replay action (macros)

with `q` command we could simplify a lot of tedious work in vim.

www.dbooks.org

example 1. generate array sequence (1 to 20).

STEP 1. press `i` to enter insert mode, input `1`

```
1
```

STEP 2. Record following action: "append the last number to the next line, and increment the number"

1. type `esc` to exit input mode
2. type `qa` to enter record mode, using buffer `a`
3. type `yy` and `p` to copy current line and paste it as the next line
4. type `ctrl` + `a` to increment number
5. type `q` again to finish record

```
1
2
```

STEP 3. Replay action 18 times.

type `18@a` to replay action 3 and action 4 in step 2.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Read Macros online: https://riptutorial.com/vim/topic/1447/macros

Chapter 26: Manipulating text

Remarks

To increment and decrement things like `11:59AM`, `3rd`, and `XVIII`, use the plugin vim-speeddating

Examples

Converting text case

In normal mode:

- `~` inverts the case of the character under the cursor,
- `gu{motion}` lowercases the text covered by `{motion}`,
- `gU{motion}` uppercases the text covered by `{motion}`

Example (`^` marks the cursor position):

```
Lorem ipsum dolor sit amet.
       ^
Lorem ipSum dolor sit amet.    ~
Lorem IPSUM DOLOR sit amet.    gU2w
Lorem IPsum DOLOR sit amet.    gue
```

In visual mode:

- `~` inverts the case of the selected text,
- `u` lowercases the selected text,
- `U` uppercases the selected text

Example (`^^^` marks the visual selection):

```
Lorem ipsum dolor sit amet.
        ^^^^^^^^^^^^
Lorem ipSUM DOLOR SIT amet.    ~
Lorem ipSUM DOLOR SIT amet.    U
Lorem ipsum dolor sit amet.    u
```

Incrementing and decrementing numbers and alphabetical characters

In normal mode, we can increment the nearest number on the line at or after the cursor with `<C-a>` and decrement it with `<C-x>`. In the following examples, the cursor position is indicated by `^`.

Incrementing and decrementing numbers

www.dbooks.org

```
for i in range(11):
      ^
```

`<C-x>` decrements the number:

```
for i in range(10):
               ^
```

`10<C-a>` increments it by `10`:

```
for i in range(20):
               ^
```

Incrementing and decrementing alphabetical characters

To make increment and decrement also work with letters, either use the ex command `:set nrformats+=alpha` or add `set nrformats+=alpha` to your `.vimrc`.

Increment example:

```
AAD
 ^
```

`<C-a>` increments it to `B`:

```
ABD
 ^
```

Decrement example:

```
ABD
  ^
```

`<C-x>` decrements `D` to `C`:

```
ABC
  ^
```

Incrementing and decrementing numbers when alphabetical increment/decrement is enabled

Notice that enabling increment/decrement to work with alphabetical characters means that you have to be careful not to modify them when you really want to just modify numbers. You can either turn off alphabetical increment/decrement by using the ex command `:set nrformats-=alpha` or you can just be aware of it and be sure to move to the number before increment or decrement. Here is the "`for i in range(11):`" example from above redone to work while alphabetical increment/decrement is set:

Let's say you want to decrease `11` to `10` and alphabetical increment/decrement is active.

```
for i in range(11):
      ^
```

Since alphabetical increment/decrement is active, to avoid modifying the character under the cursor, first move forward to the first `1` using the *normal mode* movement command `f1` (that is lowercase `f` followed by the number `1`, not to be confused with a function key):

```
for i in range(11):
               ^
```

Now, since the cursor is on the number, you can decrement it with `<C-x>`. Upon decrement, the cursor is repositioned to the last digit of the numeral:

```
for i in range(10):
                ^
```

Formatting Code

In normal mode:

`gg` go to top

`=` then `G`

Using "verbs" and "nouns" for text editing

One of the ways to think about the commands that should be executed, to edit a text in a certain manner, is as entire sentences.

A command is an action performed on an object. Therefore it has a verb:

```
:normal i    " insert
:normal a    " append
:normal c    " overwrite
:normal y    " yank (copy)
:normal d    " delete
```

Some of these words work with an object like `d`, `c`, `y`. Such objects can be **word, line, sentence, paragraph, tag**. One can use these in combination:

```
:normal dw    " deletes the text from the position of the cursor to the end of the next word
:normal cw    " deletes the text from the cursor to the end of the next word and
              " enters insert mode
```

Also one could use a **modifier** to specify precisely where should the action be executed:

```
:normal diw    " delete inside word. I.e. delete the word in which is the cursor.
```


www.dbooks.org

```
:normal ciw    " removes the word, the cursor points at and enters insert mode
:normal ci"    " removes everything between the opening and closing quotes and
               " enters insert mode
:normal cap    " change the current paragraph
:normal ct8    " remove everything until the next number 8 and enter insert mode
:normal cf8    " like above but remove also the number
:normal c/goal " remove everything until the word 'goal' and enter insert mode
:normal ci{    " change everything inside the curly braces
```

More resources:

Learn to speak vim — verbs, nouns, and modifiers!

Learning Vim in 2014: Vim as Language

VimSpeak editing using Speech Grammar

Read Manipulating text online: https://riptutorial.com/vim/topic/1707/manipulating-text

Chapter 27: Modes - insert, normal, visual, ex

Examples

The basics about modes

`vim` is a modal editor. This means that at any time inside a `vim` session, the user is going to be in one of the modes of operation. Each one of offers a different set commands, operations, key bindings...

Normal mode (or Command mode)

- The mode `vim` starts in.
- From other modes, usually accessible by `Esc`.
- **Has most of the navigation and text manipulation commands.**

See `:help normal-mode`.

Insert mode

- Commonly accessed by: `a`, `i`, `A`, `I`, `c`, `s`.
- **For inserting text**.

See `:help insert-mode`.

Visual mode

- Commonly accessed by: `v` (characterwise), `V` (linewise), `<C-v>` (blockwise).
- Basically, for **text selection**; most normal commands are available, plus extra ones to act on the selected text.

See `:help visual-mode`.

Select mode

- Accessible from insert mode with `<C-g>`.
- Similar to visual mode but with a lot less available commands.
- Contrary to insert mode, it is possible to type right away.
- Rarely used.

See `:help select-mode`.

Replace mode

www.dbooks.org

- Accessible from normal mode with `R`.
- Allows to overwrite existing text.

See `:help replace-mode`.

Command-line mode

See `:help command-line-mode`.

Ex mode

See `:help Ex-mode`.

Read Modes - insert, normal, visual, ex online: https://riptutorial.com/vim/topic/2231/modes---insert--normal--visual--ex

Chapter 28: Motions and Text Objects

Remarks

A text object in Vim is another way to specify a chunk of text to operate on. They can be used with operators or in visual mode, instead of motions.

Examples

Changing the contents of a string or parameter list

Let's say you have this line of code:

```
printf("Hello, world!\n");
```

Now say you want to change the text to "Program exiting."

Command	Buffer	Mnemonic
`ci"`	`printf("¦");`	**c**hange **i**n the **"**.
`Program exiting.\n<esc>`	`printf("Program exiting.\n");`	

Read Motions and Text Objects online: https://riptutorial.com/vim/topic/4107/motions-and-text-objects

www.dbooks.org

Chapter 29: Movement

Examples

Searching

Jumping to characters

`f{char}` - move to the next occurrence of `{char}` to the right of the cursor on the same line

`F{char}` - move to the next occurrence of `{char}` to the left of the cursor on the same line

`t{char}` - move to the left of the next occurrence of `{char}` to the right of the cursor on the same line

`T{char}` - move to the right of next occurrence of `{char}` to the left of the cursor on the same line

Jump forward / backward between the 'results' via `;` and `,`.

Further you can search for whole words via `/<searchterm>Enter`.

Searching for strings

`*` - move to the next occurrence of the word under the cursor

`#` - move to the previous occurrence of the word under the cursor

`/searchtermEnter` brings you to next match (forward-search). If you use `?` instead of `/`, searching goes backwards.

Jump between the matches via `n` (next) and `N` (previous).

To view/edit your previous searches, type `/` and hit the `up` arrow key.

Helpful are also these settings: (note `:se` is equal to `:set`)

- `:se hls` HighLightSearch, highlights all search matches; use `:noh` for temporarily turning off the search/mark highlighting (`:set noh` or `:set nohls` turns off.)
- `:se is` or `:set incs` turns Incremental Search on, cursor jumps to the next match automatically. (`:se nois` turns off.)
- `:se ic` IgnoreCase, turns case sensitivity off. (`:se noic` turns on again.)
- `:se scs` SmartCaSe, can be used when IgnoreCase is set; makes case (in)sensitivity **smart**! e.g. `/the` will search for `the`, `The`, `ThE`, etc. while `/The` only will look for `The`.

Basic Motion

Remarks

- Every motion can be used after an operator command, so the command operates on the text comprised by the movement's reach.
- Just like operator commands, motions can include a count, so you can move by `2w`ords, for example.

Arrows

In Vim, normal arrow/cursor keys (←↓↑→) work as expected. However, for touch-typers, it's easier to use the `hjkl` alternative keys. On a typical keyboard, they're located next to each other on the same row, and easily accessible using right hand. The mnemonic technique to remember which is which among them goes like this:

- `h`/`l` — those are located "most to the left/right" among the four letters on the keyboard, so they are equivalent to "going left/right" respectively;
- `j` — lowercase "j" has its tail going "down" below typical letters, like a small arrow - so it's equivalent to "going down";
- `k` — conversely, lowercase "k" has its "ascender" going "up" above typical letters, like a small pointer - so it's equivalent to "going up".

Basic motions

All commands below should be done in **normal mode**.

Command	Description
`h` or `left`	go `[count]` characters to the left
`j` or `down`	go `[count]` characters below
`k` or `up`	go `[count]` characters above
`l` or `right`	go `[count]` characters to the right
`gg`	go the first line, or `[count]`'th line, if given
`H`	go to the first line in the visible screen
`M`	go to the middle line in the visible screen
`L`	go to the last line in the visible screen
`G`	go the last line, or `[count]`'th line, if given

www.dbooks.org

Command	Description
`Home` or `0`	go to first character of the line
`^`	go to first non-blank character of the line
`+`	go down one line to first non-blank character
`-`	go up one line to first non-blank character
`$` or `End`	go to the end of the line (if `[count]` is given, go `[count - 1]` lines down)
`|`	go to the `[count]`'th character or go to the beginning of the line if `count` not specified
`f{char}`	go to `[count]`'th occurrence of `{char}` to the right *inclusive*
`F{char}`	go to `[count]`'th occurrence of `{char}` to the left *inclusive*
`t{char}`	go to `[count]`'th occurrence of `{char}` to the right *exclusive*
`T{char}`	go to `[count]`'th occurrence of `{char}` to the left *exclusive*
`;`	repeat latest `f`, `t`, `F` or `T [count]` times
`,`	repeat latest `f`, `t`, `F` or `T`, in the opposite direction, `[count]` times
`w`	go to the beginning of the next word
`b`	go to the beginning of the previous word
`e`	go to the ending of the next word
`ge`	go to the ending of the previous word
`%`	go to matching pairs, e.g `()`, `[]`, `{}`, `/* */` or `#if`, `#ifdef`, `#else`, `#elif`, `#endif`
`{ }`	previous/next paragraph
`[{]}`	beginning/ending of block
`'{char}`	Go to mark (mark with `m{char}`)
`<C-B><C-F>`	previous/next page
`<C-O><C-I>`	Go back or foward in the "jump list" (requires `jumplist` feature, see `:help jumps`)

Note: `b`, `e`, and `w` consider a word to be letters, numbers, and underscores by default (this can be configured with the `iskeyword` setting). Each of these can also be capitalized, causing them to skip over anything that isn't whitespace as well.

Note: Vim recognizes two kinds of movement: operator movement (`:help movement`) and jumps (`:help jumplist`). Movements like those executed with `g` (`gg`, `G`, `g,`) count as jumps, as do changes. Changes get their own jumplist, which is navigable as mentioned above via `g,` and `g;` (see `:help changelist`). Jumps are not treated as motion commands by Vim

When moving up or down across lines, the cursor retains its column as would be expected. If the new line is too short the cursor moves to the end of the new line. If the column is beyond the end of the line, the cursor is displayed at the end of the line. The initial column number is still retained until an action is taken to alter it (such as editing text or explicitly moving column).

If a line's length exceeds the width of the screen, the text is wrapped (under default settings, this behaviour can be configured). To move through lines as displayed on screen, rather than lines within the file, add `g` in front of the usual command. For example, `gj` will move the cursor to the position displayed one line below its current position, even if this is in the same line of the file.

Searching For Pattern

Vim supports the use of regular expressions when searching through a file.

The character to indicate that you wish to perform a search is `/`.

The simplest search you can perform is the following

```
/if
```

This will search the entire file for all instances of `if`. However, our search `if` is actually a regular expression that will match any occurrence of the word `if` including those inside of other words.

For instance, our search would say all of the following words match our search: `if`, `spiffy`, `endif`, etc.

We can do more complicated searches by using more complicated regular expressions.

If our search was:

```
/\<if\>
```

then our search would only return exact matches to the full word `if`. The above `spiffy` and `endif` would not be returned by the search, only `if`.

We can also use ranges. Given a file:

```
hello1
hello2
hello3
hello4
```

If we want to search for those lines containing "hello" followed by a digit between 1 and 3 we would say:

www.dbooks.org

```
/hello[1-3]
```

Another example:

```
/(?:\d*\.)?\d+
```

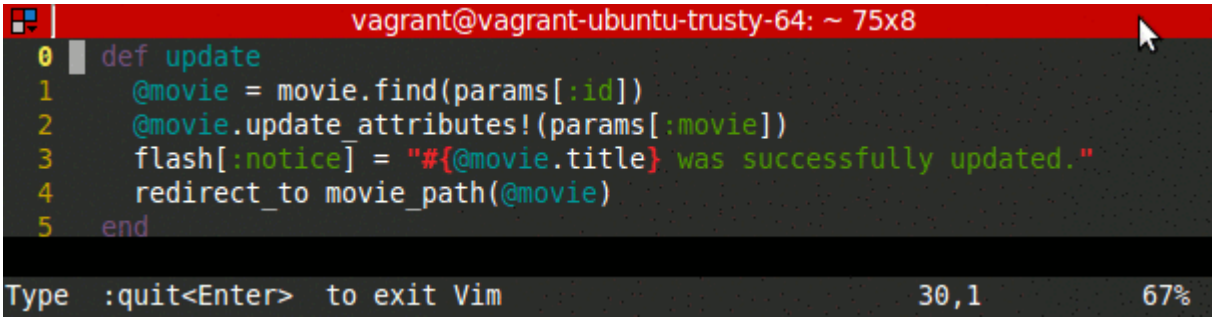
would find all of the integer and decimals numbers in the file.

Navigating to the beginning of a specific word

When editing text, a common task is to navigate to a particular word on the screen. In these examples we explore how we can navigate to the word `updated`. For the sake of consistency across the examples, we aim to land on the first letter of the word.

Mid-screen jump

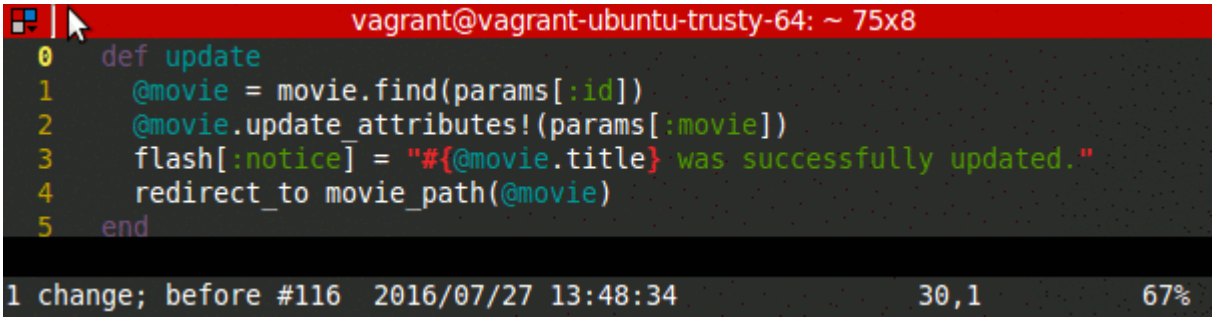
`M$B`



This approach is quick, using only 3 keystrokes. The disadvantage however, is that it is not very general, as it's not common for our target line to happen to lie right on the middle of the screen. Still, it is a useful motion when making less granular movements.

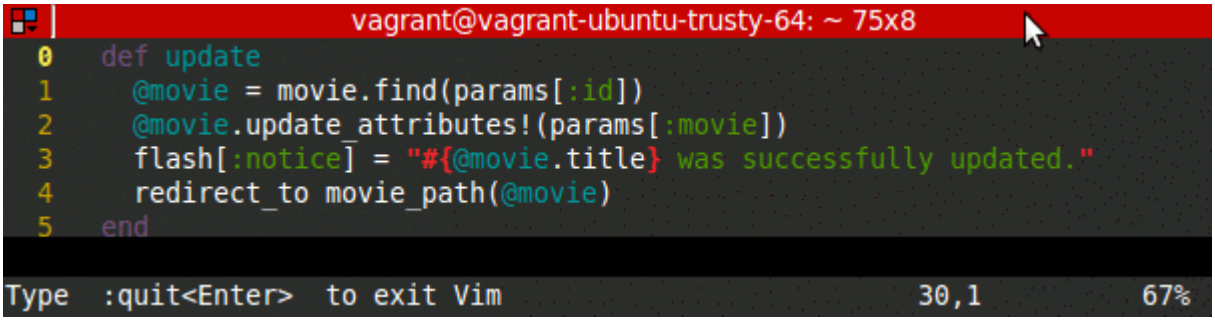
Using a count

`3jfu;;`



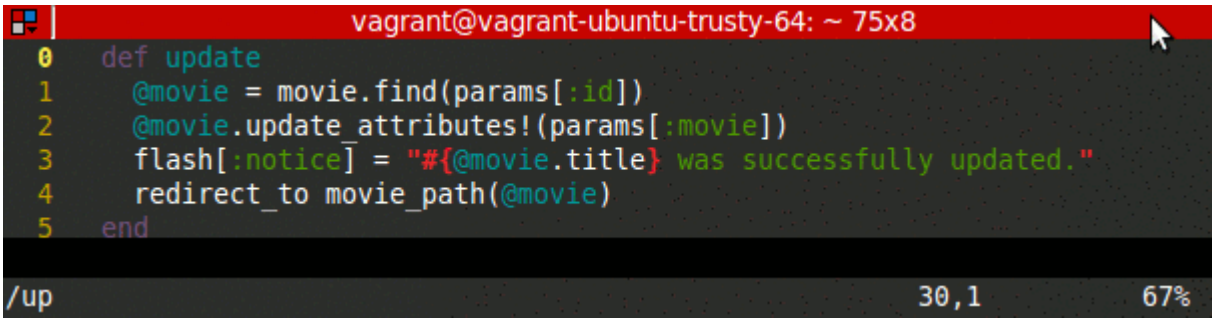
At first glance, this may appear to be a step back from the first approach because of the number of keystrokes. But since we use a count here instead of `M`, it is more flexible. We can quickly identify the correct count to use if relativenumber is enabled. To move to the target word, using `f` in combination with `;` can be surprisingly effective - and certainly better than repeatedly pressing `w`. If you overshoot your target with `;`, you can go backwards with `,`.

Explicit search
`/upEnternn`



Navigating via `/` can be very powerful. We can often jump directly to our target word by typing it out. Here we type out only the first two characters in the hope that it uniquely matches our word. Unfortunately, there are multiple matches, but we can quickly jump to the next match with `n`.

Implicit search
`/ySpaceEnterw`



In some cases, it may be more efficient to jump *near* our target rather than aiming to go directly to it. Here we observe that there is an infrequently occurring letter, `y`, right next to the target. We can add a `Space` to our search term to decrease the chances that we hit some other `y` character along the way. This can also be used to great effect with `f{char}`, as in the example *Using a count*.

Using Marks to Move Around

Marks are like bookmarks; they help you find places you've already been.

TLDR

Set them in normal mode with `m{a-zA-Z}`, and jump to them in normal or visual mode with `'{a-zA-Z}` (single quote) or `` `{a-zA-Z} `` (backtick). Lowercase letters are for marks within a buffer, and capital letters and digits are global. See your currently set marks with `:marks`, and for more info see `:help mark`.

Set a mark

Vim's built-in help says:

www.dbooks.org

```
m{a-zA-Z}               Set mark {a-zA-Z} at cursor position (does not move
                        the cursor, this is not a motion command).
```

The mark will keep track of which line and column it was placed at. There is no visual confirmation that a mark was set, or if a mark had a previous value and has been overwritten.

Jump to a mark

Vim's built-in help says:

```
Jumping to a mark can be done in two ways:
1. With ` (backtick):     The cursor is positioned at the specified location
                          and the motion is exclusive.
2. With ' (single quote): The cursor is positioned on the first non-blank
                          character in the line of the specified location and
                          the motion is linewise.
```

Backtick uses the column position, while Single-quote does not. The difference between simply allows you to ignore the column position of your mark if you want.

You can jump between non-global marks in visual mode in addition to normal mode, to allow for selecting text based on marks.

Global Marks

Global marks (capital letters) allow for jumping between files. What that means is if, for example, mark `A` is set in `foo.txt`, then from `bar.txt` (anywhere in my filesystem), if I jump to mark `A`, my current buffer will be replaced with `foo.txt`. Vim will prompt to save changes.

Jumping to a mark in another file is **not** considered to be a movement, and visual selections (among other things) will not work like jumping to marks within a buffer.

To go back to the previous file (`bar.txt` in this case), use `:b[uffer] #` (that is, `:b#` or `:buffer#`).

Note:

Special marks

There are certain marks that Vim sets automatically (which you are able to overwrite yourself, but probably won't need to).

For example (paraphrased from Vim's help):

```
`[` and `]`: jump to the first or last character of the previously changed or
             yanked text.  {not in Vi}
```

```
`<` and `>`: jump to the first or last line (with `'`) or character (with
             <code>`</code>) of the last selected Visual area in the current
             buffer.  For block mode it may also be the last character in the
             first line (to be able to define the block).  {not in Vi}.
```

More, from Vim's built-in help:

```
''  ``              To the position before the latest jump, or where the
                    last "m'" or "m`" command was given.  Not set when the
                    :keepjumps command modifier was used.
                    Also see restore-position.


'"  `"              To the cursor position when last exiting the current
                    buffer.  Defaults to the first character of the first
                    line.  See last-position-jump for how to use this
                    for each opened file.
                    Only one position is remembered per buffer, not one
                    for each window.  As long as the buffer is visible in
                    a window the position won't be changed.
                    {not in Vi}.

'.  `.              To the position where the last change was made.  The
                    position is at or near where the change started.
                    Sometimes a command is executed as several changes,
                    then the position can be near the end of what the
                    command changed.  For example when inserting a word,
                    the position will be on the last character.
                    {not in Vi}

'"  `"              To the cursor position when last exiting the current
                    buffer.  Defaults to the first character of the first
                    line.  See last-position-jump for how to use this
                    for each opened file.
                    Only one position is remembered per buffer, not one
                    for each window.  As long as the buffer is visible in
                    a window the position won't be changed.
                    {not in Vi}.

'^  `^              To the position where the cursor was the last time
                    when Insert mode was stopped.  This is used by the
                    gi command.  Not set when the :keepjumps command
                    modifier was used.  {not in Vi}
```

Additionally, the characters `(`,`)`,`{`, and `}` are marks which jump to the same position as would their normal-mode commands – that is, `'}` does the same thing in normal mode as `}`.

Jump to specific line

To jump to a specific line with colon number. To jump to the first line of a file use

```
:1
```

To jump to line 23

```
:23
```


www.dbooks.org

Read Movement online: https://riptutorial.com/vim/topic/1117/movement

Chapter 30: Normal mode commands

Syntax

- :[range]sor[t][!] [b][f][i][n][o][r][u][x] [/{pattern}/]
- Note: Options [n][f][x][o][b] are mutually exclusive.

Remarks

See sorting in the vim manual for the canonical explanation

Examples

Sorting text

Normal sorting

Highlight the text to sort, and the type:

```
:sort
```

If you don't highlight text or specify a range, the whole buffer is sorted.

Reverse sorting

```
:sort!
```

Case insensitive sorting

```
:sort i
```

Numerical sorting

Sort by the first number to appear on each line:

```
:sort n
```

Remove duplicates after sorting

```
:sort u
```


www.dbooks.org

(u stands for unique)

Combining options

To get a reverse case-insensitive sort with duplicates removed:

```
:sort! iu
```

Read Normal mode commands online: https://riptutorial.com/vim/topic/6005/normal-mode-commands

Chapter 31: Normal mode commands (Editing)

Examples

Introduction - Quick Note on Normal Mode

In Normal Mode, commands can be entered by direct key combinations (typing `u` to undo the last change, for example). These commands often have equivalents in 'ex' mode, accessed by typing a colon `:`, which drops you into a single-line buffer at the bottom of the Vim window.

In 'ex' mode, after typing the colon you type a command name or its abbreviation followed by `Enter` to execute the command. So, `:undoEnter` accomplishes the same thing as directly typing `u` in Normal Mode.

You can see that the direct commands will often be faster (once learned) than the 'ex' commands for simple editing, but for completeness, wherever possible in the documentation that follows, if both are available for use then both will be shown.

Most of these commands can also be preceded with a *count* by prefixing or interspersing a number - typing `3dd` in Normal Mode, for example, deletes three lines (beginning from the current cursor position).

Basic Undo and Redo

Undo

Command	:	Description
`u`	`u,undo`	Undo the most recent change
`5u`		Undo the *five* most recent changes (use any number)

Please be aware that in Vim, the 'most recent change' varies according to the mode you are in. If you enter Insert Mode (`i`) and type out an entire paragraph before dropping back to Normal Mode (`Esc`), *that entire paragraph* is considered the most recent change.

Redo

Command	:	Description
`Ctrl-R`	`red,redo`	Redo the most recent undone change
`2Ctrl-R`		Redo the *two* most recent undone changes (use any number)

www.dbooks.org

There is one other way to undo and redo changes in Vim that is handled a bit differently. When you undo a change with `u`, you traverse back up the nodes on a 'tree' of your changes, and pressing `Ctrl-R` walks back down those nodes in order. (The undo tree is a separate topic and is too complex to cover here.)

You can *also* use `U` (that is, uppercase) to remove all the latest changes on a single line (the line where your last changes were made). This *does not* traverse the nodes of the tree in the same way as `u`. Using `U` actually counts as a change itself - another node *forward* on the tree - so that if you press `U` a second time immediately after the first it will act as a Redo command.

Each has its uses, but `u` / `:undo` should cover most simple cases.

Repeat the Last Change

The Repeat command, executed with the dot or period key (`.`), is more useful than it first appears. Once learned, you will find yourself using it often.

Command	:	Description
`.`		Repeat the last change
`10.`		Repeat the last change 10 times

So then, for a very simple example, if you make a change to line 1 by typing `iI`Esc, with the following result:

```
1 I made a mistake
2  made a mistake
3  made a mistake
```

Your cursor will be at position 1 of line 1, and all you need to do to fix the next two lines is press `j.` twice - that is, `j` to move down a line and `.` to repeat the last change, which was the addition of the `I`. No need to jump back into Insert Mode twice to fix those lines.

It becomes much more powerful when used to repeat macros.

Copy, Cut and Paste

In Vim, these operations are handled differently from what you might be used to in almost any other modern editor or word processor (`Ctrl-C`, `Ctrl-X`, `Ctrl-V`). To understand, you need to know a little about registers and motions.

Note: this section will not cover Visual Mode copying and cutting or range yanking as these are beyond the scope of both Normal Mode and basic editing.

Registers

Vim uses the concept of *registers* to handle moving text around within the program itself. Windows

has a single clipboard for this purpose, which is analogous to a single register in Vim. When copying, cutting, and pasting in Vim, there are ways to use a similarly simple editing workflow (where you don't have to think about registers), but there are also much more complex possibilities.

A register is targeted for the input/output of a command by prefixing the command with `"` and a lowercase letter name.

Motions

A motion in Vim is any command that moves the cursor position elsewhere. When copying, cutting, and pasting in Normal Mode, the possibilities of text selection for movement are only limited by your knowledge of motions. A few will be illustrated below.

Copying and Cutting

The basic commands copy and cut operations are built on are `y` ('yank', for copy) and `d` ('delete', for cut). You'll see the similarities in the following table.

Command	:	Description
`y{motion}`		Copy ('yank') text indicated by the motion into the default register
`yy`		Copy the current line into the default register, *linewise*
`Y`		Copy the current line into the default register (synonym for `yy`)
`"ayiw`		Copy the word the cursor is on into register 'a'
`20"byy`		Copy twenty lines, beginning from the cursor, into register 'b'
`d{motion}`		Cut ('delete') text indicated by the motion into the default register
`dd`		Cut the current line into the default register, *linewise*
`D`		Cut from the cursor to end of line into the default register (NOT a synonym for `dd`)
`"adiw`		Cut the word the cursor is on into register 'a'
`20"bdd`		Cut twenty lines, beginning from the cursor, into register 'b'

Note: when something is copied or cut *linewise*, the paste behavior shown below will place text either before or after the current *line* (rather than the cursor). Examples follow to clarify.

Pasting

www.dbooks.org

There are several ways to paste in Vim, depending on what you are trying to accomplish.

Command	:	Description
`p`		Paste whatever is in the default register *after* the cursor
`P`		Paste whatever is in the default register *before* the cursor
`"ap`		Paste the contents of register 'a' after the cursor
`"cP`		Paste the contents of register 'c' before the cursor

So, How Do I Perform A Really Simple Cut and Paste?

If I have the following text:

```
1 This line should be second
2 This line should be first
```

I can do the simplest cut-and-paste by placing my cursor somewhere on line 1 and typing `ddp`. Here are the results:

```
1 This line should be first
2 This line should be second
```

What happened? `dd` 'Cuts' the first line (linewise) into the default register - which will only contain one thing at a time, like the Windows clipboard - and `p` pastes the line after the current one, which has just changed due to the `dd` command.

Here's a not-quite-as-simple example. I need to move a couple of words around. (This is contrived and unnecessary, but you can apply this principle to larger chunks of code.)

```
1 These words order out are of
```

I can repeat `w` to get to the 'o' at the front of 'order' (or `b` if I just typed it and realized my mistake).

Then `"adaw` to put 'order ' in register 'a'.

Then `w` to get to the 'a' in 'are'.

Following this, I would type `"bdaw` to put 'are ' into register 'b'. Now I have this displayed:

```
1 These words out of
```

To be clear, now 'order ' is in register 'a' and 'are ' is in register 'b', like two separate clipboards.

To arrange the words correctly, I type `b` to get to the 'o' in 'out', and then `"bP` to put 'are ' from register 'b' in front of 'out':

```
1 These words are out of
```

Now I type `A` to get to the end of the line, followed by `SpaceEsc` (assuming there was no space after 'of') and `"ap` to put 'order' where it belongs.

```
1 These words are out of order
```

Completion

Completion can be used to match words used in a document. When typing a word, `Ctrlp` or `Ctrln` will match previous or next similar words in the document.

This can even be combined with `Ctrl-X` mode to complete entire lines. For instance type something like:

```
This is an example sentence.
```

then go to the next line and begin typing the same sentence:

```
Thi
```

and then hit `Ctrlp` which will result in:

```
This
```

Now still in insert mode, hit `Ctrlx Ctrlp` and then next word will be completed resulting in:

```
This is
```

Continue hitting `Ctrlx Ctrlp` until the entire line is completed.

If you know you want to complete an entire line type this:

```
This is an example sentence.
```

then on the next line type:

```
Thi
```

and hit `x Ctrll` to complete the line.

If the completion being done is a filename `Ctrlx Ctrlf` can be used to complete that directory. Type:

```
~/Deskt
```

then hit `Ctrlx Ctrlf` and:

www.dbooks.org

```
~/Desktop
```

will be completed (if at that location). `Ctrlx Ctrlf` can then be repeatedly used to list the files in the Desktop.

Read Normal mode commands (Editing) online: https://riptutorial.com/vim/topic/5250/normal-mode-commands--editing-

Chapter 32: Plugins

Examples

Fugitive Vim

Fugitive Vim is a plugin by Tim Pope that provides access to git commands that you can execute without leaving vim.

Some common commands include:

`:Gedit` - edit a file in the index and write it to stage the the changes
`:Gstatus` - equivalent of `git status`
`:Gblame` - brings up vertical split of output from `git blame`
`:Gmove` - for git mv
`:Gremove` - for git rm
`:Git` - run any command

It also adds items to the `statusline` like indicating the current branch.

Please see their GitHub for more details and installation instructions.

NERD Tree

NERD TREE is a plugin by scrooloose that allows you to explore the file system while using vim. You can open files and directories via a tree system that you can manipulate with the keyboard or the mouse.

Add this to your .vimrc to start NERDTree automatically when vim starts up:

```
autocmd vimenter * NERDTree
```

To automatically close NERDTree if it's the only window left add this to your .vimrc:

```
autocmd bufenter * if (winnr("$") == 1 && exists("b:NERDTree") && b:NERDTree.isTabTree()) | q
| endif
```

It's recommended to map a key combination to the NERDTreeToggle command. Add this to your .vimrc (this example uses Ctrl + N)

```
map <C-n> :NERDTreeToggle<CR>
```

Full details and installation instructions can be view on their Github.

Read Plugins online: https://riptutorial.com/vim/topic/9976/plugins

www.dbooks.org

Chapter 33: Regular expressions

Remarks

execute `:h pattern` to see a lot of regex related information

Examples

Word

Vim has special operators to match word beginning, word, end, and so forth.`\<` represents the beginning of a word and `\>` represents the end of a word.

Searching for `/\<foo\>` in the following text will only return the last foo.

> football is not foolish `foo`

Read Regular expressions online: https://riptutorial.com/vim/topic/6533/regular-expressions

Chapter 34: Regular expressions in Ex Mode

Examples

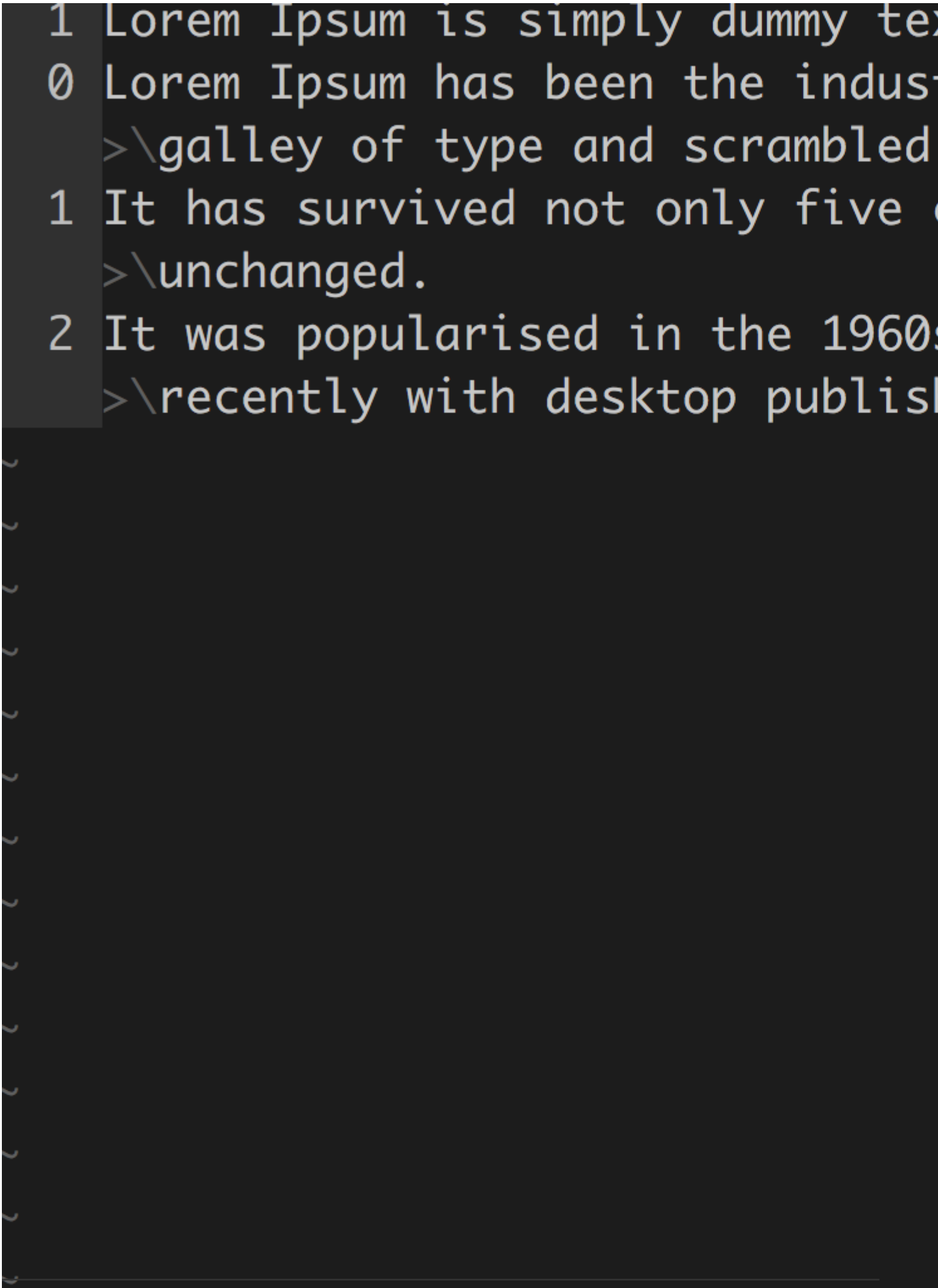
Edit a regular expression in Ex mode

Suppose you are searching for a `Title Case` pattern in a large text file and you want to edit a incorrect regular expression:

1. First, go into `Ex` mode by typing `q:`
2. You will now see all the commands that you typed in `commandline` mode, press `j` to go the regular expression you want to edit (`/\v[A-Z]\w+\s[A-Z]\w+`)
3. Once done, press `ESC` to go to normal mode
4. Then press `Enter` to run the search patten

Here is a screen shot demonstrating a `Title Case` search

www.dbooks.org


1 Lorem Ipsum is simply dummy tex
0 Lorem Ipsum has been the indus
>\galley of type and scrambled
1 It has survived not only five
>\unchanged.
2 It was popularised in the 1960
>\recently with desktop publis

https://riptutorial.com/vim/topic/6472/regular-expressions-in-ex-mode

www.dbooks.org

Chapter 35: Saving

Examples

Saving a buffer in a non-existent dir

```
:!mkdir -p %:h
```

to create the missing directories, then

```
:w
```

Read Saving online: https://riptutorial.com/vim/topic/6440/saving

Chapter 36: Scrolling

Examples

Scrolling downwards

Command	Description
`Ctrl+E`	Scroll one line down.
`Ctrl+D`	Scroll half a screen down (configurable using the `scroll` option).
`Ctrl+F`	Scroll a full screen down.
`z+`	Draw the first line below the window at the top of the window.

Scrolling upwards

Command	Description
`Ctrl+Y`	Scroll one line up.
`Ctrl+U`	Scroll half a screen up (configurable using the `scroll` option).
`Ctrl+B`	Scroll a full screen up.
`z^`	Draw the first line above the window at the bottom of the window.

Scrolling relative to cursor position

Command	Description
`z`	Redraw current line at the top of the window and put the cursor on the first non-blank character on the line.
`zt`	Like `z` but leave the cursor in the same column.
`z.`	Redraw current line at the center of the window and put the cursor on the first non-blank character on the line.
`zz`	Like `z.` but leave the cursor in the same column.
`z-`	Redraw current line at the bottom of the window and put the cursor on the first non-blank character on the line.
`zb`	Like `z-` but leave the cursor in the same column.

www.dbooks.org

Read Scrolling online: https://riptutorial.com/vim/topic/3000/scrolling

Chapter 37: Searching in the current buffer

Examples

Searching for an arbitrary pattern

Vim's standard search commands are `/` for forward search and `?` for backward search.

To start a search from normal mode:

1. press `/`,
2. type your pattern,
3. press `<CR>` to perform the search.

Examples:

```
/foobar<CR>      search forward for foobar
?foo\/bar<CR>    search backward for foo/bar
```

`n` and `N` can be used to jump to the next and previous occurence:

- Pressing `n` after a forward search positions the cursor on the next occurence, *forwards*.
- Pressing `N` after a forward search positions the cursor on the next occurence, *backwards*.
- Pressing `n` after a backward search positions the cursor on the next occurence, *backwards*.
- Pressing `N` after a backward search positions the cursor on the next occurence, *forwards*.

Searching for the word under the cursor

In normal mode, move the cursor to any word then press `*` to search forwards for the next occurrence of the word under the cursor, or press `#` to search backwards.

`*` or `#` search for the exact word under the cursor: searching for `big` would only find `big` and not `bigger`.

Under the hood, Vim uses a simple search with *word boundaries* atoms:

- `/\<big\>` for `*`,
- `?\<big\>` for `#`.

`g*` or `g#` don't search for the exact word under the cursor: searching for `big` would find `bigger`.

Under the hood, Vim uses a simple search without *word boundaries* atoms:

- `/\<big\>` for `*`,
- `?\<big\>` for `#`.

www.dbooks.org

execute command on lines that contain text

The `:global` command already has its own topic: The global command

Read Searching in the current buffer online: https://riptutorial.com/vim/topic/3269/searching-in-the-current-buffer

Chapter 38: Solarized Vim

Introduction

Spending most of the time on terminal can be a big deal for eyes. Wisely choosing color scheme can benefit your eyes in many ways. Recently I ran into `Solarized ColorScheme for Vim`. Adding this small plugin can make a big difference on text appearance on terminal. Many thanks to Ethan Schoonover for developing this package. The `howtos` are explained pretty well here. Enjoy!

Examples

.vimrc

Solarized has two options - light and dark mode.

Light Mode:

```
syntax enable
set background=light
colorscheme solarized
```

Dark Mode:

```
syntax enable
set background=dark
colorscheme solarized
```

Read Solarized Vim online: https://riptutorial.com/vim/topic/9500/solarized-vim

www.dbooks.org

Chapter 39: Spell checker

Examples

Spell Checking

To turn on the vim spell checker run `:set spell`. To turn it off run `:set nospell`. If you always want the spell checker to be on, add `set spell` to your vimrc. You can turn spelling on only for certain filetypes using an auto command.

Once the spell checker is on, misspelled words will be highlighted. Type `]s` to move to the next misspelled word and `[s` to move to the previous one. To see a list of corrected spellings, place the cursor on a misspelled word and type `z=`. You can type the number of the word you wish to replace the misspelled word with and hit `<enter>` to replace it, or you can just hit enter to leave the word unchanged.

With the cursor on a misspelled word, you can also type `<number>z=` to change to the `<number>`th correction without viewing the list. Typically you will use `1z=` if you think `vim`'s first guess is likely to be the correct word.

Set Word List

To set the word list that `vim` will use for spell checking set the `spelllang` option. For example

```
:set spelllang=en        # set to English, usually this is the default
:set spelllang=en_us     # set to U.S. English
:set spelllang=en_uk     # set to U.K. English spellings
:set spelllang=es        # set to Spanish
```

If you want to set the `spelllang` and turn on spell checking in one command, you can do:

```
:setlocal spell spelllang=en
```

Read Spell checker online: https://riptutorial.com/vim/topic/3653/spell-checker

Chapter 40: Split windows

Syntax

- `:split <file>`
- `:vsplit <file>`
- `:sp` <- shorthand for split
- `:vsp` <- shorthand for vsplit

Remarks

When called from the command line, multiple files can be provided in the argument and vim will create one split for each file. When called from ex mode, only one file can be opened per invocation of the command.

Examples

Opening multiple files in splits from the command line

Horizontally

```
vim -o file1.txt file2.txt
```

Vertically

```
vim -O file1.txt file2.txt
```

You may optionally specify the number of splits to open. The following example opens two horizontal splits and loads `file3.txt` in a buffer:

```
vim -o2 file1.txt file2.txt file3.txt
```

Opening a new split window

You can open a new split within Vim with the following commands, in *normal* mode:

Horizontally:

```
:split <file name>
:new
```

Vertically:

```
:vsplit <file name>
```


www.dbooks.org

```
:vnew
```

split will open the file in a new split at the top or left of your screen (or current split.) `:sp` and `:vs` are convenient shortcuts.

new will open an empty split

Changing the size of a split or vsplit

You may sometimes want to change the size of a split or vsplit.

To change the size of the currently active split, use `:resize <new size>`. `:resize 30` for example would make the split 30 lines tall.

To change the size of the currently active vsplit, use `:vertical resize <new size>`. `:vertical resize 80` for example would make the vsplit 80 characters wide.

Shortcuts

- `Ctrl + w` and `+` increase the size of the splited window
- `Ctrl + w` and `-` decrease the size of the splited window
- `Ctrl + w` and `=` set an equal size to the splited windows

Close all splits but the current one

Normal mode
`Ctrl-wo`

Ex mode

```
:only
```

or short

```
:on
```

Managing Open Split Windows (Keyboard Shortcuts)

After you have opened a split window in vim (as demonstrated by many examples under this tag) then you will likely want to control windows quickly. Here is how to control split windows using keyboard shortcuts.

Move to split Above/Below:

- `Ctrl + w` and `k`
- `Ctrl + w` and `j`

Move to split Left/Right:

- `Ctrl + w` and `h`
- `Ctrl + w` and `l`

Move to split Above/Below (wrap):

- `Ctrl + w` and `w`

Create new empty window:

- `Ctrl + w` and `n` -or- :new

Create new split horizontal/vertical:

- `Ctrl`+`W`, `S` (upper case)
- `Ctrl`+`W`, `v` (lower case)

Make the currently active split the one on screen:

- `Ctrl + w` and `o` -or- :on

Move between splits

To move to split on left, use `<C-w><C-h>`
To move to split below, use `<C-w><C-j>`
To move to split on right, use `<C-w><C-k>`
To move to split above, use `<C-w><C-l>`

Sane split opening

It's a better experience to open split below and on right

set it using

```
set splitbelow
set splitright
```

Read Split windows online: https://riptutorial.com/vim/topic/1705/split-windows

www.dbooks.org

Chapter 41: Substitution

Syntax

- `s/<pattern>/<pattern>/optional-flags`
- `<pattern>` is a Regex

Parameters

Flag	Meaning
&	Keep the flags from the previous substitute.
c	Prompt to confirm each substitution.
e	Do not report errors.
g	Replace all occurrences in the line.
i	Case-insensitive matching.
I	Case-sensitive matching.
n	Report the number of matches, do not actually substitute.

Remarks

Use `set gdefault` to avoid having to specify the 'g' flag on every substitute.

Example

When `gdefault` is set, running `:s/foo/bar` on the line `foo baz foo` will yield `bar baz bar` instead of `bar baz foo`.

Examples

Simple replacement

`:s/foo/bar` Replace the **first** instance of *foo* with *bar* on the current line.

`:s/foo/bar/g` Replace every instance of *foo* with *bar* on the current line.

`:%s/foo/bar/g` Replace *foo* with *bar* throughout the entire file.

Quickly refactor the word under the cursor

1. `*` on the word you want to substitute.
2. `:%s//replacement/g`, leaving the *find* pattern empty.

Replacement with interactive approval

`:s/foo/bar/c` Marks the first instance of *foo* on the line and asks for confirmation for substitution with *bar*

`:%s/foo/bar/gc` Marks consecutively every match of *foo* in the file and asks for confirmation for substitution with *bar*

Keyboard short-cut to replace currenlty highlighted word

For example, with following `nmap`:

```
nmap <expr> <S-F6> ':%s/' . @/ . '//gc<LEFT><LEFT><LEFT>'
```

select a word with `*`, type `Shift-F6`, type in a replacement and hit `Enter` to rename all occurrences interactively.

Read Substitution online: https://riptutorial.com/vim/topic/3384/substitution

www.dbooks.org

Chapter 42: The dot operator

Examples

Basic Usage

The dot operator repeats the last action performed, for instance:

file `test.tx`

```
helo, World!
helo, David!
```

(cursor at [1][1])
Now perform a `cwHello<Esc>` (Change Word `helo` to `Hello`)
Now the buffer looks like that:

```
Hello, World!
helo, David!
```

(cursor at [1][5])
After typing `j_` the cursor is at [2][1].

Now enter the `.` and the last action is performed again:

```
Hello, World!
Hello, David!
```

(cursor at [2][5])

Set indent

This is very useful when setting indent of your code

```
if condition1
if condition2
# some commands here
endif
endif
```

move your cursor to the 2nd line, then `>>`, the code will indent to right.

Now you can repeat your action by continue to 3rd line, then hit `.` twice, the result will be

```
if condition1
    if condition2
        # some commands here
endif
```

```
endif
```

Read The dot operator online: https://riptutorial.com/vim/topic/3665/the-dot-operator

www.dbooks.org

Chapter 43: Tips and tricks to boost productivity

Syntax

- :set relativenumber
- :set number
- :set nonumber / :set nonu
- :pwd

Remarks

This automatic reload will only happen if you edit your `vimrc` in a version full version of vim which supports `autocmd`.

Examples

Quick "throwaway" macros

Add this to your vimrc:

```
nnoremap Q @q
```

To start recording the "throwaway" macro, use `qq`. To finish recording hit `q` (in normal mode for both).

To execute the recorded macro, use `Q`.

This is useful for macros that you need to repeat many times in a row but won't be likely to use again afterward.

Using the path completion feature inside Vim

This is very common, you memorize a path to a file or folder, you open up Vim and try to write what you've just memorized, but you are not 100% sure it's correct, so you close the editor and start over.

When you want the path completion feature, and you have a file `/home/ubuntu/my_folder/my_file` and you are editing another file referencing to the path of the previous one:

Enter insert mode: `insert` or do it the way you want. Next, write `/h`. When the cursor is under `h`, press `Ctrlx` and then `Ctrlf` so the editor will complete it to `/home/` because the pattern `/h` is unique

Now, suppose you have two folders inside `/home/ubuntu/` called `my_folder_1` `my_folder_2`

and you want the path `/home/ubuntu/my_folder_2`

as usual:

Enter insert mode

write `/h` and press `Ctrlx` and then `Ctrlf` . Now you have `/home/` Next add u after /home/ and press `Ctrlx` and then `Ctrlf` . Now, you have `/home/ubuntu/` because that path is unique. Now, write `my` after `/home/ubuntu/` and press `Ctrlx` and then `Ctrlf` . The editor will complete your word until `my_folder_` and you will see the directory tree so use the arrow keys to choose the one you want.

Turn On Relative Line Numbers

To delete some lines of text when you don't know exact number of lines to delete, you try `10dd` , `5dd` , `3dd` until you remove all the lines.

Relative line numbers solves this problem, suppose we have a file containing :

```
sometimes, you see a block of
text. You want to remove
it but you
cannot directly get the
exact number of
lines to delete
so you try
10d , 5d
3d until
you
remove all the block.
```

Enter NORMAL mode: `Esc`

Now, execute `:set relativenumber`. Once done the file will be displayed as:

```
3 sometimes, you see a block of
2 text. You want to remove
1 it but you
0 cannot directly get the
1 exact number of
2 lines to delete
3 so you try
4 10d , 5d
5 3d until
6 you
7 remove all the block.
```

where `0` is the line number for the current line and it also shows the real line number in front of relative number, so now you don't have to estimate the numbers of lines from the current line to cut or delete or worse count them one by one.

You can now execute your usual command like `6dd` and you are sure about the number of lines.

You can also use the short form of the same command `:set rnu` to turn on relative numbers and

www.dbooks.org

`:set nornu` to turn off the same.

If you also `:set number` or have `:set number` already on, you'll get the line number of the line in which the cursor is at.

```
3 sometimes, you see a block of
2 text. You want to remove
1 it but you
4 cannot directly get the
1 exact number of
2 lines to delete
3 so you try
4 10d , 5d
5 3d until
6 you
7 remove all the block.
```

Viewing line numbers

To view line numbers from Default view enter

```
:set number
```

To hide line numbers

```
:set nonumber
```

There is also a shortcut for above. `nu` is same as `number`.

```
:set nonu
```

To hide line numbers, we can also use

```
:set nu!
```

Mappings for exiting Insert mode

A lot of Vim users find the `Esc` too hard to reach, and end up finding another mapping that's easy to reach from the home row. Note that `Ctrl-[` may be equivalent to `Esc` on an English keyboard, and is much easier to reach.

jk

```
inoremap jk <ESC>
```

This one is really easy to trigger; just smash your first two fingers on the home row at the same time. It's also hard to trigger accidentally since "jk" never appears in any English word, and if you're in normal mode it doesn't do anything. If you don't type "blackjack" too much, then consider

also adding `inoremap kj <ESC>` so you don't need to worry about timing of the two keys.

Caps Lock

Linux

On Linux, you can use `xmodmap` to make `Caps Lock` do the same thing as `Esc`. Put this in a file:

```
!! No clear Lock
clear lock
!! make caps lock an escape key
keycode 0x42 = Escape
```

Then run `xmodmap file`. This remaps `Caps Lock` to `Esc`.

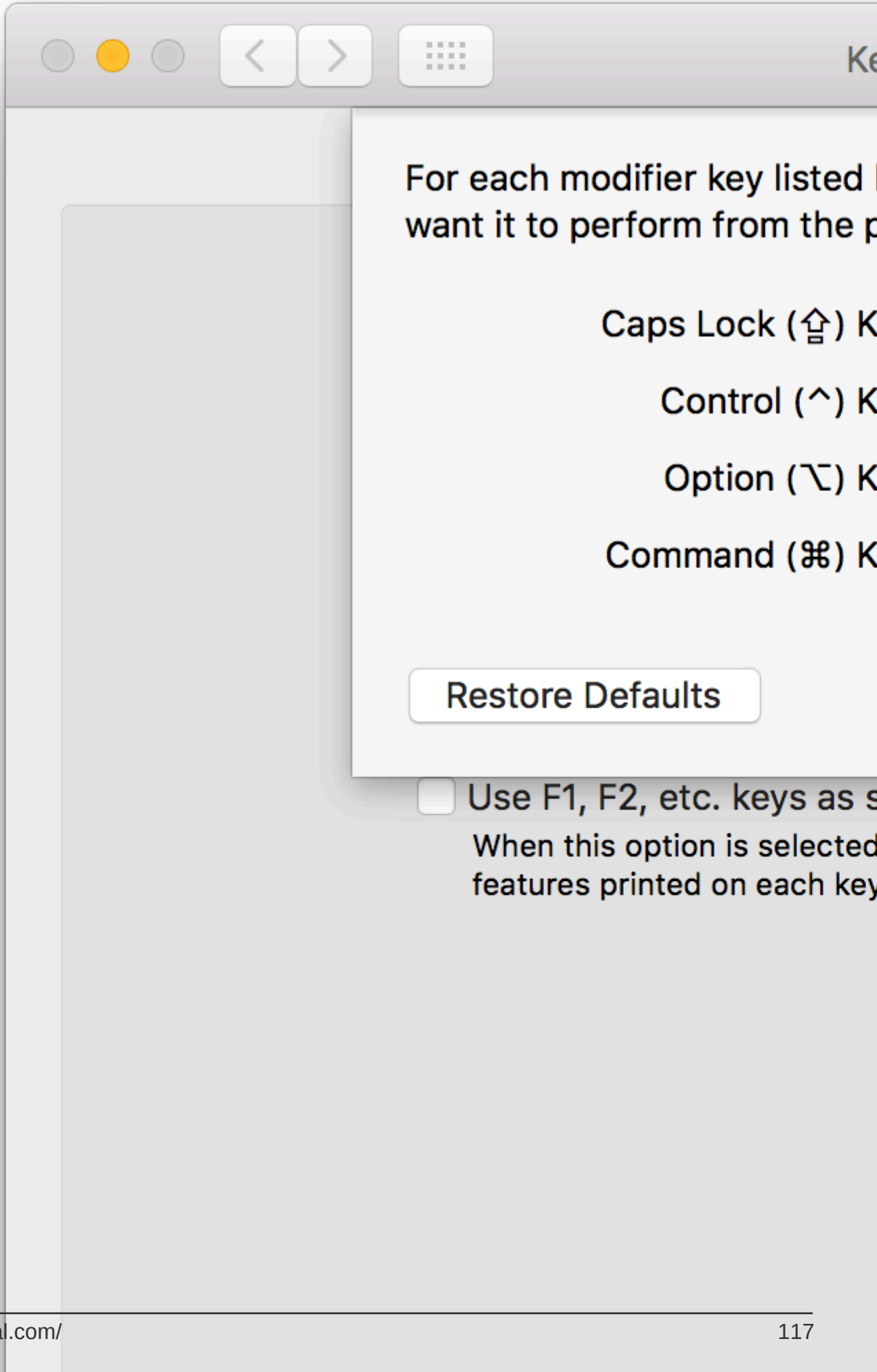
Windows

On Windows you can use SharpKey or AutoHotkey.

macOS

If you have macOS 10.12.1 or later, you can remap Caps Lock to Escape using System Preferences. Select Keyboard, go to the Keyboard tab, and click Modifier Keys.

www.dbooks.org


For each modifier key listed
want it to perform from the
Caps Lock (⇪) K
Control (^) K
Option (⌥) K
Command (⌘) K
Restore Defaults
Use F1, F2, etc. keys as
When this option is selected
features printed on each key

- escape key, `<CR>` - enter key) :

```
vi{<ESC>/\%Vfoo<CR>
```

now you can jump between the matches within the block by pressing `n` and `p`. If you have `hlsearch` option enabled this will highlight all the matches. `\%V` is a special search pattern part, that tells vim to search only in the visually selected area. You can also make a mapping like this:

```
:vnoremap g/ <ESC>/\%V
```

After this the above command is shortened to the following:

```
vi{g/foo<CR>
```

Another useful trick is to print all the lines containing the pattern:

```
vi{
:'<,'>g/foo/#
```

The `'<,'>` range is inserted automatically.

See `:help range` and `:help :g`.

Copy, move or delete found line

A lot of users find themselves in a situation where they just want to copy, move or delete a line quickly and return to where they were.

Usually, if you'd want to move a line which contains the word `lot` below the current line you'd type something like:

```
/lot<Esc>dd<C-o>p
```

But to boost productivity you can use this shortcut in these cases:

```
" It's recommended to turn on incremental search when doing so
set incsearch

" copy the found line
cnoremap $t <CR>:t''<CR>
" move the found line
cnoremap $m <CR>:m''<CR>
" delete the found line
cnoremap $d <CR>:d<CR>``
```

So a search like this:

```
/lot$m
```


www.dbooks.org

would move the line which contains `lot` below the line your cursor was on when you started the search.

Write a file if you forget to `sudo` before starting vim

This command will save the open file with sudo rights

```
:w !sudo tee % >/dev/null
```

You can also map `w!!` to write out a file as root

```
:cnoremap w!! w !sudo tee % >/dev/null
```

Automatically reload vimrc upon save

To automatically reload `vimrc` upon save, add the following to your `vimrc`:

```
if has('autocmd') " ignore this section if your vim does not support autocommands
    augroup reload_vimrc
        autocmd!
        autocmd! BufWritePost $MYVIMRC,$MYGVIMRC nested source %
    augroup END
endif
```

and then for the last time, type:

```
:so $MYVIMRC
```

The next time you save your `vimrc`, it will be automatically reloaded.

`nested` is useful if you're using vim-airline. The process of loading airline triggers some autocommands, but since you're in the process of executing an autocommand they get skipped. `nested` allows triggering nested autocommands and allows airline to load properly.

Command line completion

set `wildmenu` to turn on completion suggestions for command line.
Execute the following

```
set wildmenu
set wildmode=list:longest,full
```

Now if you do say, `:colortab`,

You'll get

```
256-jungle  Benokai  BlackSea  C64  CandyPaper  Chasing_Logic  ChocolateLiquor
:color 0x7A69_dark
```

Read Tips and tricks to boost productivity online: https://riptutorial.com/vim/topic/3382/tips-and-tricks-to-boost-productivity

www.dbooks.org

Chapter 44: Useful configurations that can be put in .vimrc

Syntax

- set mouse=a
- set wrap
- nmap j gj
- nmap k gk

Examples

Move up/down displayed lines when wrapping

Usually, `J` and `K` move up and down file lines. But when you have wrapping on, you may want them to move up and down the **displayed** lines instead.

```
set wrap " if you haven't already set it
nmap j gj
nmap k gk
```

Enable Mouse Interaction

```
set mouse=a
```

This will enable mouse interaction in the `vim` editor. The mouse can

- change the current cursor's position
- select text

Configure the default register to be used as system clipboard

```
set clipboard=unnamed
```

This makes it possible to copy/paste between Vim and the system clipboard without specifying any special register.

`yy` yanks the current line into the system clipboard

`p` pastes the content of the system clipboard into Vim

This only works if your Vim installation has clipboard support. Run the following command in the terminal to check if the clipboard option is available: `vim --version | grep clipboard`

Read Useful configurations that can be put in .vimrc online:
https://riptutorial.com/vim/topic/6560/useful-configurations-that-can-be-put-in--vimrc

www.dbooks.org

Chapter 45: Using ex from the command line

Examples

Substitution from the command line

If you would like to use vim in a manner similar to `sed`, you may use the `-c` flag to run an ex command from the command line. This command will run automatically before presenting the file to you. For example, to replace `foo` with `bar`:

```
vim file.txt -c "s/foo/bar"
```

This will open up the file with all instances of `foo` replaced with `bar`. If you would to like to make changes to the file *without* having to manually save, you can run multiple ex commands, and have the last command write and quit. For example:

```
vim file.txt -c "s/foo/bar" -c "wq"
```

Important note:

You can *not* run multiple ex commands separated by a bar `|`. For example

```
vim file.txt -c "s/foobar | wq"
```

Is *not* correct; however, it CAN be done if you use `ex`.

```
ex -c ":%s/this/that/g | wq" file.txt
```

Read Using ex from the command line online: https://riptutorial.com/vim/topic/6819/using-ex-from-the-command-line

Chapter 46: Using Python for Vim scripting

Syntax

- :[range]py[thon] {statement}

Examples

Check Python version in Vim

Vim has its own built-in Python interpreter. Thus it could use a different version of the default interpreter for the operating system.

To check with which version of Python Vim was compiled, type the following command:

```
:python import sys; print(sys.version)
```

This imports the `sys` module and prints its `version` property, containing the version of the currently used Python interpreter.

Execute Vim normal mode commands through Python statement

To be able to use vim commands in Python, the `vim` module should be imported.

```
:python import vim
```

After having this module imported, the user has access to the `command` function:

```
:python vim.command("normal iText to insert")
```

This command would execute `i` in normal mode then type `Text to insert` and fall back to normal mode.

Executing multi-line Python code

Every Python statement in Vim should be prefixed with the `:python` command, to instruct Vim that the next command is not Vimscript but Python.

To avoid typing this command on each line, when executing multi-line Python code, it is possible to instruct Vim to interpret the code between two marker expressions as Python.

To achieve this, use:

```
:python << {marker_name}
a = "Hello World"
```


www.dbooks.org

```
print(a)
{marker_name}
```

where `{marker_name}` is the word you want to use to designate the end of the python block.

E.g.:

```
:python << endpython
surname = "Doe"
forename = "Jane"
print("Hello, %s %s" % (forename, surname))
endpython
```

would print:

```
Hello, Jane Doe
```

Read Using Python for Vim scripting online: https://riptutorial.com/vim/topic/5604/using-python-for-vim-scripting

Chapter 47: vglobal: Execute commands on lines that do not match globally

Introduction

:vglobal or :v is the opposite of :global or :g that operates on lines not matching the specified pattern (inverse).

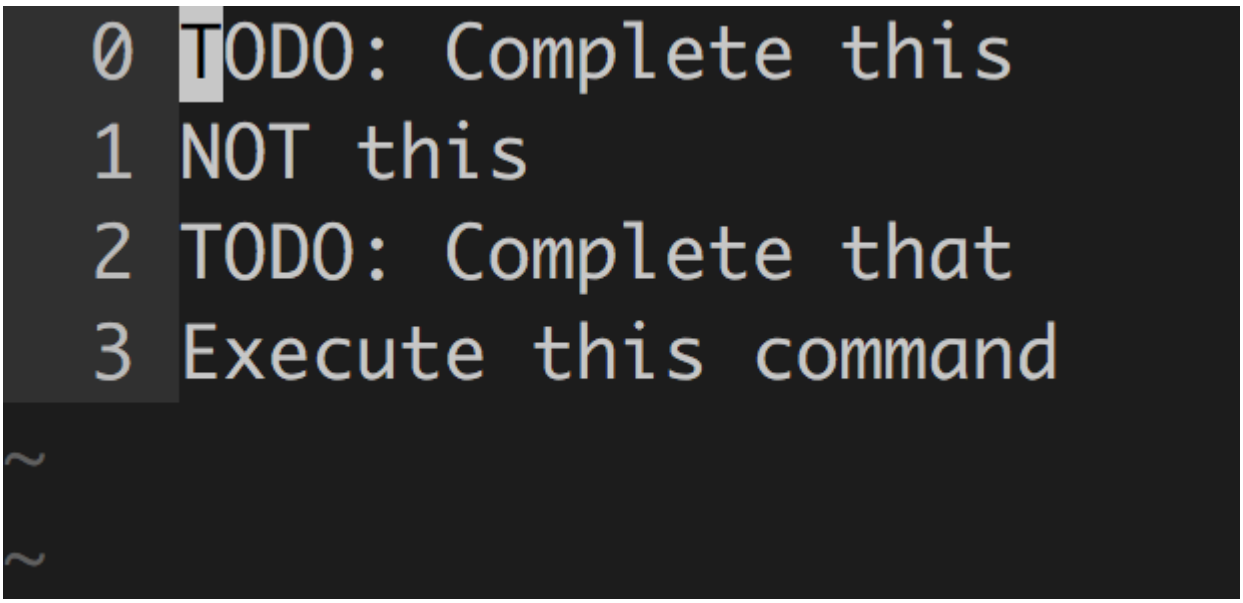
Examples

:v/pattern/d

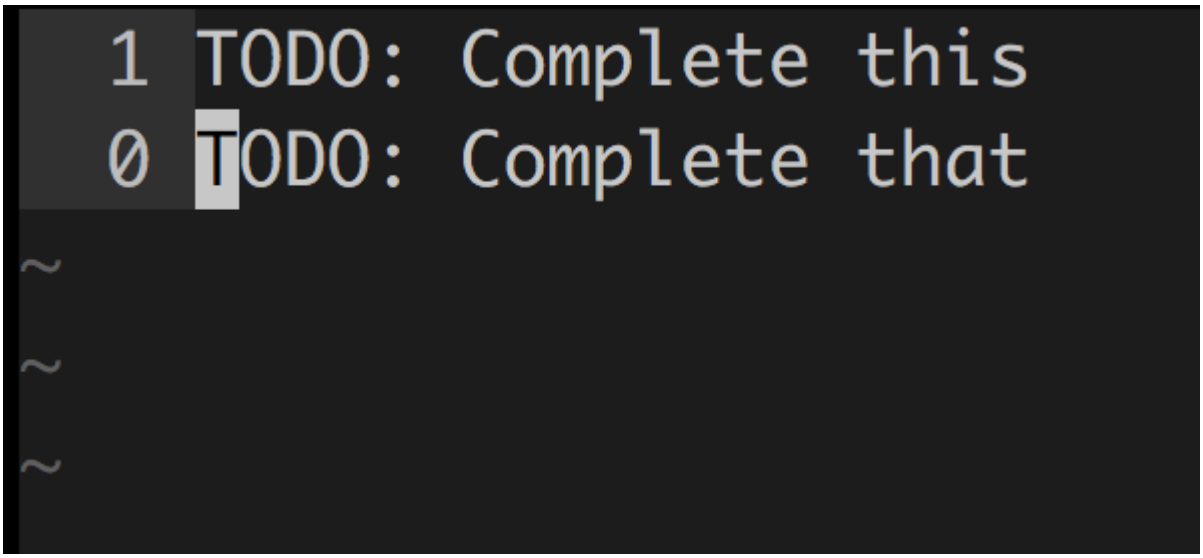
Example:

```
> cat example.txt
  TODO: complete this
  NOT this
  NOT that
  TODO: Complete that
```

Open the `example.txt` using `vim` and type `:v/TODO/d` in the `Ex` mode. This will delete all lines that do not contain the `TODO` pattern.



www.dbooks.org



Read vglobal: Execute commands on lines that do not match globally online: https://riptutorial.com/vim/topic/9867/vglobal--execute-commands-on-lines-that-do-not-match-globally

Chapter 48: Vim Options

Syntax

- `:set [no](option|shortcut)`
- `:set (option|shortcut)=value`
- `:set (option|shortcut)(?|&)`
- do not use `:` in the vimrc file

Remarks

See vimcast 1 video

See vimcast 1 transcript

Examples

Set

To set the options - use `:set` instruction. Example:

```
:set ts=4
:set shiftwidth=4
:set expandtab
:set autoindent
```

To view the current value of the option - type `:set {option}?`. Example:

```
:set ts?
```

To reset the value of the option to its default - type `:set {option}&`. Example:

```
:set ts&
```

Indentation

Width

To make indentations 4 spaces wide:

```
:set shiftwidth=4
```


www.dbooks.org

Spaces

To use spaces as indents, 4 spaces wide:

```
:set expandtab
:set softtabstop=4
```

`softtabstop` and `sts` are equivalent:

```
:set sts=4
```

Tabs

To use tabs as indents, 4 spaces wide:

```
:set noexpandtab
:set tabstop=4
```

`tabstop` and `ts` are equivalent:

```
:set ts=4
```

Automatic Indentation

```
:set autoindent
```

Instruction descriptions

Instruction	Description	Default
tabstop	width of tab character	8
expandtab	causes spaces to be use instead of tab character	off
softabstop	tune the whitespace	0
shiftwidth	determines whitespace amount when in normal mode	8

Invisible characters

Show or hide invisible characters

To show invisible characters:

```
:set list
```

To hide invisible characters:

```
:set nolist
```

To toggle between showing and hiding invisible characters:

```
:set list!
```

Default symbol characters

Symbol	Character
^I	Tab
$	New Line

Customize symbols

To set the tab character to **> ** and the new line character to ¬

```
set listchars=tab:>\ ,eol:¬
```

To set the spaces to _

```
set listchars=spaces
```

To see a list of character options

```
:help listchars
```

Read Vim Options online: https://riptutorial.com/vim/topic/2407/vim-options

www.dbooks.org

Chapter 49: Vim Registers

Parameters

Functionality	Registers
default register	`""`
history registers	`"[1-9]`
yank register	`"0`
named registers	`"[a-z]`, `"[A-Z]` same as `"[a-z]` but appends
recall current search pattern	`"/`
small deletes (diw, cit, ...)	`"-`
expression registers for simple math	`"=`
black hole register to eliminate large chunks of deleted text from mem	`"_`
last command	`":`
last inserted text	`".`
filename	`"%`
clipboard	`"*`
selected text	`"+`
dropped text	`"~`

Examples

Delete a range of lines into a named register

In Normal, type the following to delete a range of lines into a named register

```
:10,20d a
```

This will delete lines 10,20 in register `"a`. We can verify this by typing

```
:reg
```

This will show the text that was delete in register `"a`.

To paste the contents in `"a`, just type

```
"ap
```

Paste the filename while in insert mode using the filename register

In Insert mode, press `<C-r>` and then `%` to insert the filename.

This technique is applicable to all registers.

For e.g. if in insert mode, you want to paste the current search pattern, you can type `<C-r>` and then `/`.

Copy/paste between Vim and system clipboard

Use the quotestar register to copy/paste between Vim and system clipboard

`"*yy` copies the current line into the system clipboard

`"*p` pastes the content of the system clipboard into Vim

Append to a register

Yank all lines containing TODO into a register by using append operation

```
:global/TODO/yank A
```

Here, we are searching for a `TODO` keyword globally, yanking all lines into register `a` (`A` register appends all lines to `a` register).

NOTE: It is in general a good practice to clear a register before performing the append operation.

To clear a register, in the normal mode, type `qaq`. Confirm that the `a` register is empty by typing `:reg` and observing that `a` register is empty.

Read Vim Registers online: https://riptutorial.com/vim/topic/4278/vim-registers

www.dbooks.org

Chapter 50: Vim Resources

Remarks

This Topic is about **Source Code mirrors**, **Books**, **Vim-Wikis**. It is **NOT** about Blog entries, Wikipedia, Tutorials. The resources should not be opinion based.

Examples

Learning Vimscript the Hard Way

A book explaining how Vimscript works, full of examples. It can be found on http://learnvimscriptthehardway.stevelosh.com/

Read Vim Resources online: https://riptutorial.com/vim/topic/6383/vim-resources

Chapter 51: Vim Text Objects

Examples

Select a word without surrounding white space

Suppose we want to select a word without surrounding white spaces, use the text object `iw` for inner word using visual mode:

1. Got to normal mode by pressing `ESC`
2. Type `viw` at the beginning of a word
3. This will select the inner word



Select a word with surrounding white space

Suppose we want to select a word with a surrounding white space, use the text object `aw` for around a word using visual mode:

1. Got to normal mode by pressing ESC
2. Type `vaw` at the beginning of a word
3. This will select the word with white space

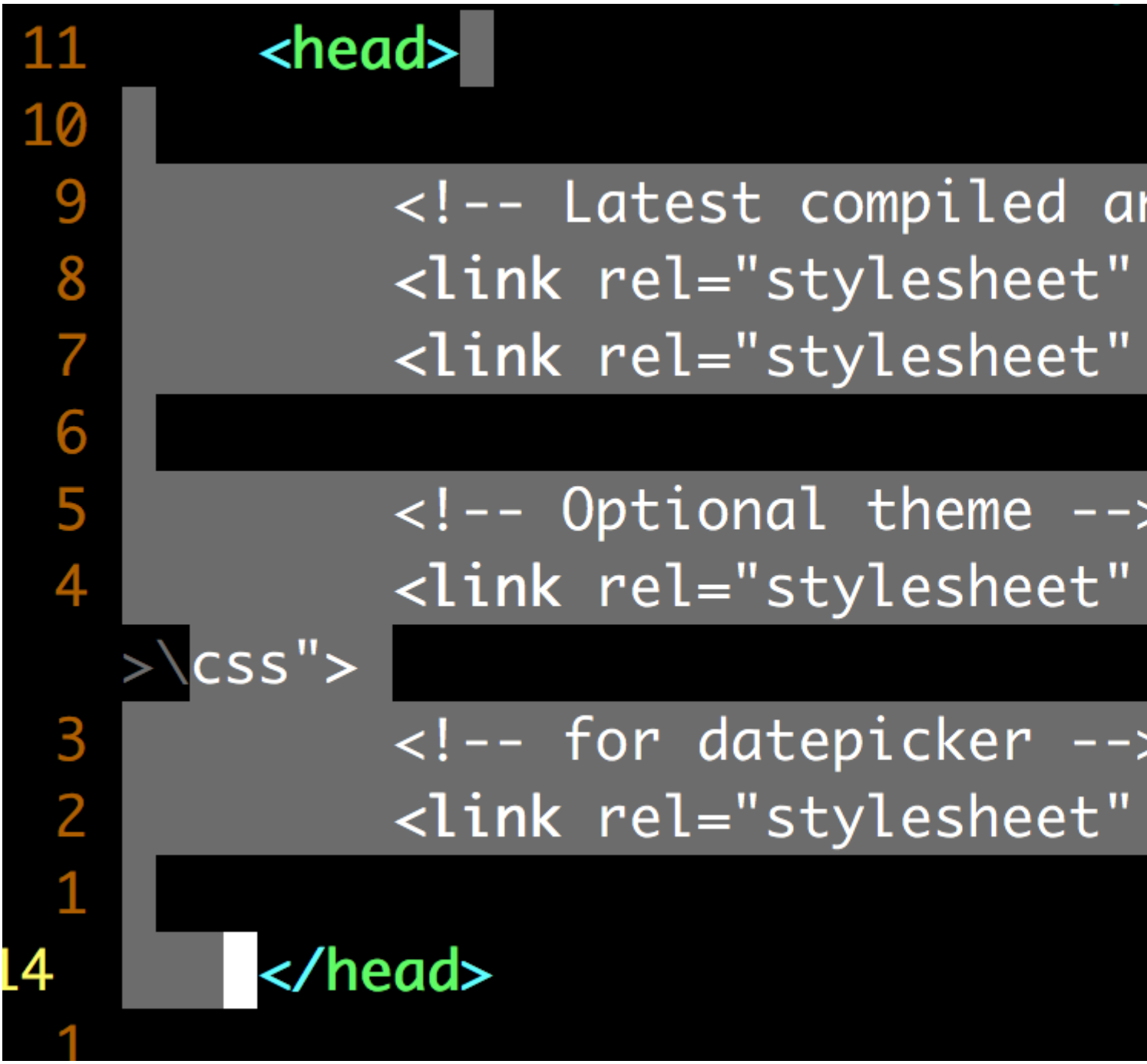


Select text inside a tag

We can select a text within an `html` or `xml` tag by using visual selection `v` and text object `it` .

1. Go to normal mode py pressing `ESC`
2. Type `vit` from anywhere within the `html` or `xml` section
3. This will visually select all text inside the `tag`

www.dbooks.org



All other text objects can also be used to operate on the text inside the tag

1. `cit` - delete text inside the tag and place in `insert` mode
2. `dit` - delete text inside the tag and remain in `normal` mode
3. `cat` - delete around tag and place in `insert` mode
4. `dat` - delete text around the tag and remain in `normal` mode

Read Vim Text Objects online: https://riptutorial.com/vim/topic/4050/vim-text-objects

Chapter 52: Vimscript

Remarks

The commands in a Vimscript file are executed in `command mode` by default. Therefore all non-`command mode` directives should be prefixed.

Examples

Hello World

When attempting to print something for debugging in vimscript, it is tempting to simply do the following.

```
echo "Hello World!"
```

However, in the context of a complex plugin, there are often many other things happening right after you attempt to print your message, so it is important to add `sleep` after your message so you can actually see it before it disappears.

```
echo "Hello World!"
sleep 5
```

Using Normal Mode Commands in Vimscript

Since a Vimscript file is a collection of Command mode actions, the user needs to specify that the desired actions should be executed in normal mode.

Therefore executing a normal mode command like `i`, `a`, `d` etc. in Vimscript is done by prepending the command with `normal`:

Going to the bottom of the file and selecting the last 5 rows:

```
normal GV5k
```

Here the `G` instructs vim to change the cursor position to the last row, the `V` to go to linewise visual mode , and the `5k` to go 5 rows up.

Inserting your name at the end of the row:

```
normal ABoris
```

where the `A` puts the editor in insert mode at the end of the row and the rest is the text to insert.

Read Vimscript online: https://riptutorial.com/vim/topic/5136/vimscript

www.dbooks.org

Chapter 53: Whitespace

Introduction

Here is how you can clean up whitespace.

Remarks

See vimcast 4 transcript

Examples

Delete trailing spaces in a file

You can delete trailing spaces with the following command.

```
:%s/\s\+$//e
```

This command is explained as follows:

- enter Command mode with `:`
- do this to the entire file with `%` (default would be for the current line)
- substitute action `s`
- `/` start of the search pattern
- `\s` whitespace character
- `\+` escaped + sign, one or more spaces should be matched
- before the line end `$`
- `/` end of the search pattern, beginning of replacement pattern
- `/` end of the replacement pattern. Basically, replace with nothing.
- `e` suppress error messages if no match found

Delete blank lines in a file

You can delete all blank lines in a file with the following command: :g/^$/d

This command is explained as follows:

- enter Command mode with `:`
- `g` is a global command that should occur on the entire file
- `/` start of the search pattern
- the search pattern of blank line is `^g`
- `/`end of the search pattern
- Ex command `d` deletes a line

Convert tabs to spaces and spaces to tabs

You can convert tabs to spaces by doing the following:

First check that expandtab is switched off

```
:set noexpandtab
```

Then

```
:retab!
```

which replaces spaces of a certain length with tabs

If you enable expandtab again `:set expandtab` then and run the `:retab!` command then all the tabs becomes spaces.

If you want to do this for selected text then first select the text in visual mode.

Read Whitespace online: https://riptutorial.com/vim/topic/8288/whitespace

www.dbooks.org

Credits

S. No	Chapters	Contributors
1	Getting started with vim	A. Raza, akavel, Ashok, carrdelling, Christian Rondeau, Community, Cows quack, Daniel, Daniel Käfer, Daniel Margosian, Deborah V, depperm, ericdwang, ExistMe, GiftZwergrapper, gmoshkin, HerrSerker, James, Js Lim, KerDam, LittleByBlue, liuyang1, LotoLo, Marek Skiba, Mattias, Miljen Mikic, mnoronha, Nasreddine, Nhan, Nick Weseman, pktangyue, redBit Device, Romain Vincent, romainl, ropata, Rory O'Kane, Sardathrion, sascha, SeekAndDestroy, sjas, sudo bangbang, Sumner Evans, tbodt, Tejus Prasad, TheMole, timss, Tom Gijselinck, Tom Lord, user2314737, user45891, Vin, Vishnu Kumar, vvnraman, Wieland, Wojciech Kazior, zarak, Zaz
2	:global	cmlaverdiere, DJMcMayhem, LittleByBlue, tbodt, Vin
3	Advantages of vim	gmoshkin, LittleByBlue
4	Ask to create non-existant directories upon saving a new file	Tom Hale
5	Autocommands	joeytwiddle, tbodt, Tom Hale
6	Auto-Format Code	Philip Kirkbride
7	Buffers	Chris Jones, eli, joeytwiddle, sudo bangbang
8	Building from vim	grochmal, Josh Petrie, LittleByBlue, Luc Hermitte
9	Command-line ranges	RamenChef, romainl
10	Configuring Vim	Aaron Thoma, bn., Christian Rondeau, Cometsong, Cows quack, Daniel, Johnathan Andersen, KerDam, Luc Hermitte, lwassink, mezzode, nobe4, romainl, SnoringFrog, sudo bangbang, Sumner Evans, timss, Wojciech Kazior, Yosh
11	Converting text files from DOS to UNIX with vi	grochmal, LazyBrush
12	Differences between	still_dreaming_1, tbodt

	Neovim and Vim	
13	Easter Eggs	Aaron Thoma, Andrea Romagnoli, Christian Rondeau, cmlaverdiere, Daniel Käfer, Gerard Roche, LittleByBlue, Mateusz Piotrowski, Mattias, mcarton, nobe4, NonlinearFruit, sudo bangbang, tbodt, Tejus Prasad
14	Enhanced undo and redo with a undodir	GiftZwergrapper
15	Exiting Vim	Arulpandiyan Vadivel, asclepix, AWippler, Nick Weseman, Yosef Nasr
16	Extending Vim	baptistemm, LittleByBlue, Nikola Geneshki, romainl, satyanarayan rao, Sumner Evans, void
17	Filetype plugins	Luc Hermitte, romainl
18	Find and Replace	DJMcMayhem, Kara, naveen.panwar, ncmathsadist, romainl, sudo bangbang, zzz
19	Folding	Josh Petrie, LittleByBlue, sudo bangbang
20	Get :help (using Vim's built-in manual)	Aidan Miles, Luc Hermitte
21	How to Compile Vim	Ingo Karkat, Josh Petrie, romainl
22	Indentation	dallyingllama, Daniel, RamenChef, toto21
23	Inserting text	Batsu, Boysenb3rry, Christopher Bottoms, cmlaverdiere, codefly , DJMcMayhem, Eric Bouchut, GiftZwergrapper, gmoshkin, Johnathan Andersen, Kent, lazysoundsystem, Mahmood, omul, Promarbler, RamenChef, rodrigo, romainl, satyanarayan rao, Scroff, SnoringFrog, sudo bangbang, Sundeep, timss, Tom Lord , UNagaswamy, Xavier Nicollet
24	Key Mappings in Vim	Christian Rondeau, Ingo Karkat, KerDam, Luc Hermitte, madD7, New Alexandria, Nikola Geneshki, RamenChef, romainl
25	Macros	Johan, Johnathan Andersen, lazysoundsystem, LittleByBlue, Oliver Wespi, rjmill, romainl, TheMole, Victor Schröder, vielmetti , Wenzhong
26	Manipulating text	Chris Nager, Christopher Bottoms, LittleByBlue, Nikola Geneshki, Philip Kirkbride, romainl, till, Tom Hale, zarak
27	Modes - insert, normal, visual, ex	rgoliveira, romainl

www.dbooks.org

28	Motions and Text Objects	tbodt
29	Movement	Aidan Miles, akavel, Boysenb3rry, Caek, Chris H, Cows quack, depperm, fedorqui, Georgi Dimitrov, gmoshkin, JacobLeach, jamessan, KerDam, Madis Pukkonen, Noam Hacker, rgoliveira, sjas, SnoringFrog, Sundeep, timss, Tyler, Vin, Wazam, zarak
30	Normal mode commands	Tom Hale
31	Normal mode commands (Editing)	A. Raza, rodericktech, romainl, The Nightman
32	Plugins	Nick Weseman
33	Regular expressions	4444, sudo bangbang
34	Regular expressions in Ex Mode	UNagaswamy
35	Saving	abidibo
36	Scrolling	Delapouite, evuez
37	Searching in the current buffer	Abdelaziz Dabebi, DJMcMayhem, LittleByBlue, romainl
38	Solarized Vim	Luc Hermitte, Nick Weseman, satyanarayan rao
39	Spell checker	andipla, Johnathan Andersen, lwassink
40	Split windows	beardc, Boysenb3rry, Downgoat, Goluxas, grenangen, HerrSerker, Johnathan Andersen, KerDam, madD7, Sachin Divekar, sudo bangbang, timss, Victor Schröder, zarak
41	Substitution	cmlaverdiere, LittleByBlue, Nikola Geneshki, Timur
42	The dot operator	Js Lim, LittleByBlue
43	Tips and tricks to boost productivity	Abdelaziz Dabebi, adelarsq, Chris Midgley, depperm, GanitK, gath, gmoshkin, Hotschke, KerDam, LittleByBlue, LotoLo, mash, naveen.panwar, RamenChef, rjmill, romainl, Simone Bronzini, souparno majumder, Stryker, sudo bangbang, tbodt, Tom Hale
44	Useful configurations that can be put in .vimrc	Cows quack, maniacmic, Tomh
45	Using ex from the	DJMcMayhem, Matt Clark

	command line	
46	Using Python for Vim scripting	Nikola Geneshki
47	vglobal: Execute commands on lines that do not match globally	UNagaswamy
48	Vim Options	dallyingllama, LittleByBlue, mezzode, Wojciech Kazior, Yosh
49	Vim Registers	maniacmic, romainl, UNagaswamy
50	Vim Resources	Nikola Geneshki
51	Vim Text Objects	UNagaswamy
52	Vimscript	merlin2011, Nikola Geneshki
53	Whitespace	dallyingllama

www.dbooks.org