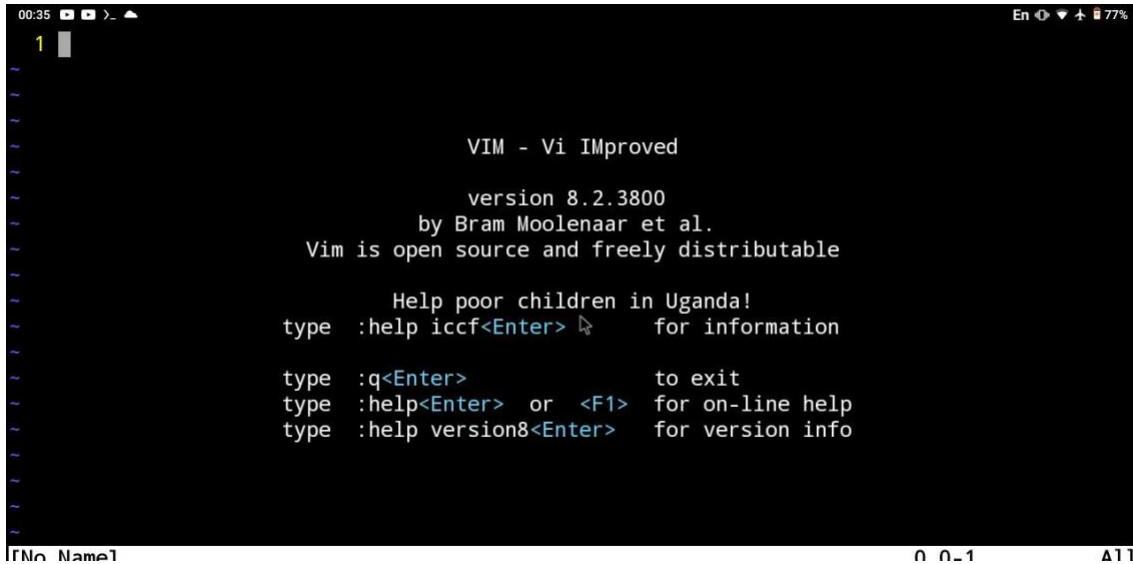


# Learning Vim

Author: Lucas Gouveia Belon

## Save and Quit

When we enter VIM, the first thing we need to know is how to save and quit.

A screenshot of the Vim text editor's startup screen. The background is black with white text. At the top left, it shows the time '00:35' and some system icons. At the top right, it shows 'En', a battery icon, and '77%'. The main text in the center reads: 'VIM - Vi IMproved', 'version 8.2.3800', 'by Bram Moolenaar et al.', and 'Vim is open source and freely distributable'. Below this, there is a section titled 'Help poor children in Uganda!' followed by several lines of help text: 'type :help iccf<Enter> for information', 'type :q<Enter> to exit', 'type :help<Enter> or <F1> for on-line help', and 'type :help version8<Enter> for version info'. At the bottom left, it says '[No Name]', and at the bottom right, it says '0 0-1 Δ111'.

When we write `$ vim` we enter in this screen. In order to save this empty document we must write with the command `:write` and insert a name to the file. If we want to quit VIM we must write the command `:quit`. If we want to make a copy of the file and edit we can use the command `:saveas` and then name the new file.

Almost all commands have a reduced form, and many can be combined. We also have lots and lots of shortcuts. To write and then quit the document we can write `:wq`. The shortcut for `:wq` is `ZZ`, pay attention, it's upper cased.

If we change the file, and want to quit without saving it, we must force quit with `:quit!`. Now we're able to enter and exit from VIM. Most of the times you'll be able to see your command on the right bottom corner.

The text editor comes with a tutorial to help with the basic commands, to enter on it we need to write `$ vimtutor` on the terminal. When we use the `:help` we'll find out that some words are links to other texts. To follow the link we can use the `<CTRL-J>` command, and to go back to where you were you can press `<CTRL-T>`.

## Basic Modes

When we open VIM we found ourselves in what's called the **NORMAL MODE**. When we want to move fast inside a file, generally, we use this mode. It's probably the mode with more shortcuts. In this mode we can move, write, quit, scroll the screen, set some marks, change to other modes, and mainly, we can manage our tabs, buffers, and windows. If we enter in some other mode we always can write `<ESC>` or `<CTRL-c>` to get back to this mode. Since sometimes the `<ESC>` key is so much far, so, personally, I rather use the `<CTRL-c>` command.

When in normal mode we can change to the **INSERT MODE** with many different commands. We can write `i` to start writing wherever our cursor is, we can use the `a`, that moves

the cursor one position to the right and then starts the insert mode. We can use the **s** that erases the character under the cursor and then starts the insert mode. More than this, we can use upper cased **i** to start the insert mode in the beginning of the line, upper cased **a** to start in the end of the line, and upper cased **S** to erase the whole line and then start insert mode. There are also the **c** and upper cased **C** command, but we'll talk in depth about this one later.

The last basic mode is the **VISUAL MODE**. Many times we select some text by holding shift and moving with arrow keys, or even by using the mouse. VIM has a mode for this. Actually there are three kinds of visual mode. When we're at normal mode we can enter visual mode by writing **v**. We can also write the upper case **V** to enter on the **VISUAL-LINE MODE**, that select only whole lines. And we can enter on **VISUAL-BLOCK MODE**, by pressing **<CTRL-v>**. The block mode is useful to manage tables. When we're at visual mode we can change the side of our selection with the **o** command. It is easier to understand when we try it.

There is a mode which is not much told, the **REPLACE MODE**. We can replace a single character with the **r** command, or we can replace until we quit the mode with the **R** command. It may not be that useful, but it's good to know that it exists.

## Moving

### Moving through text

VIM has this pretty nice idea about combining commands. The idea is the Number-Action-Movement chain of commands. First we'll see how to move, later we'll see some actions. When we're at insert mode we can move with the arrow keys, but the most powerful movements can be done on normal mode.

If you saw the vimtutor you found that the most basic way to move is with the **h-j-k-l** keys. With **j** we go down one line. With **k** we go up one line. With **h** we go to the left one character, and with **l** we go right.

Moving one character or one line can be boring. To make things go quickly we can use numbers, like, **3j** to go 3 lines down, or **10l** to go 10 characters to the right. We're using the number-movement pattern. With those command we can have some level of freedom of motion, but things can get better.

We can use a command to jump words, it will be faster most of the time. This time we have two commands, **W** and **e**. One considers dots and underscores, dashes, other special characters, the other doesn't. To go back we use **b**. We can remember it with the mnemonic **w – word** and **b – backwards**. I'm not sure about how those commands behave when upper cased, try it and find out.

Still on the line moves subject, we can go to the start of the line with the **O (zero)** command, and to the end with **\$**. But, lets say that we have a line with a empty space and we want to go to the first non-empty position of the line, we can do it with the command **^**. Since sometimes we need the press it twice to work I suggest that this should be done by pressing **^** and then space.

Here are a couple of move commands that are very powerful. When we want to go to a specific character of the line we can jump to it with the **t** command. Remember with "go 'til [character]". The usage is **t[character]**. The **t** command goes forward, the **f** command does the same, but backwards. Nice right?

When we're programming we found ourselves in need to jump to the ( or to the ) that is close to us. Or even to the [ or ], or to the { or }. Summarizing, we can jump to the other side of a logical parenthesis with a command, the % command.

Many times we will want to move through whole paragraphs. To do it we can use the those {} keys. We didn't saw this yet, but, when we move with { or with } we leave a mark, it will work like a jump history. We can move through this history with <CTRL-o> to go to the previous position and <CTRL-i> to go to the next position. Other jumps also leave a jump history, but you'll find out whose are they the more you use VIM.

We can jump to the beginning of the file with the gg command. Also, we can go to the end with the G command. Many motion commands exists in pairs.

At last, we can jump to a specific line of our text with :[number of the line]. This way, if we're editing a code and on execution it gives us an error that is on line 124 we can go there with :124. If you remember all the move commands you'll be able to move with freedom.

Until now we saw many ways to move, keep in mind that it's very important to be familiar with all kinds of moves, cause later we will use the full Number-Action-Movement to see some examples.

## Moving the screen

We have the option to do a page-up, page-down, but also a half-page up/down and even a one-line up/down. Those three pairs of commands always work this way: The left hand goes down, the right hand goes up.

To use the whole page-up we can do <CTRL-b>, and to the whole page-down we can do <CTRL-f>. a mnemonic to it is "f is for falling", and "b is for bring up".

To use the half page up we can do <CTRL-u> and to the half page-down we can do <CTRL-b>. To remember this one is easy, u stands for up, d stands for down.

To use the one-line up we can do <CTRL-y> and to the one-line down we can do <CTRL-e>. There is no easy way to remember this one, but with practice you'll be able to use it.

We can use more than only those page-up like commands, we can move the screen relatively to the position of the cursor. We can scroll the screen to make the cursor to stand at the top, at the bottom, and at the middle of the screen.

To make the cursor stand in the middle of the screen we can do zz.

To make the cursor stand at the top of the screen we can do zt.

To make the cursor stand at the bottom of the screen we can do zb.

## Deleting

Now we'll start to see more in depth some Number-Action-Movement stuff. Remember that most of the time we'll want to delete and then start the insert mode to make a correction.

We can delete a character with the x command, or we can use d[motion] to do it. If we use, for example, dl, we'll get the same effect of x. But we can go beyond. We can use 3dl to delete 3 characters, or d\$ to delete from the cursor to the end of the line. Also, we can use dd and delete a whole line.

Some commands are a shortcut to some combination of **d[motion]**, like **D** that is equal to the **d\$** command, it erases from where the cursor is to the end of the line.

## Visualizing motion

First we'll see the **aw** command, by letting the cursor at the middle of a word and starting visual mode.

Here we can see that our cursor is at the end of the word. Since we're at the visual mode we can change the position of our cursor to be at the other corner of the selection.



4

## Searching in our file

We have at least three ways to search. We can use the `/[item]` command to search for an item that exists (or not) at our file, but it will search forwards. We can search backwards with the `?[item]` command. And at last, we can search for the occurrences of the word that's under our cursor with the `#` command, but it works backwards, to jump forwards we can use `*` command. If we want to repeat the last search we can do `//` or `??`.

Our search accepts REGEX syntax, so you can become a all-powerful searcher, and also, to know REGEX will give superpowers at other commands of our favorite text editor.

## Cut, Copy, Paste and the clipboard

Many devices as smartphones, computers and text editors have a Cut, Copy and Paste function. VIM also does, but it's more powerful than that.

Every time we delete something it goes to our clipboard, so we already know how to cut. To paste is easy, just press `p` to paste it on the below line or `P` to paste at this line, right under the cursor. The behavior may vary depending on what you copied. Generally, if you copied a whole line, `p` will do as I said.

And now, how to Copy? That's also easy. You can use the Action-Movement to copy with `y[motion]` or copy a whole line with `yy`, we say that we yanked the text. We have a shortcut to `yy` which is `Y`. It's good to know that VIM has a very complex clipboard system, so you can copy to what is called **Registers**, but we'll see it later, then we'll be able to save different fragments of text in different "slots".

## Substitution

What if we want to change every occurrence of "now" to "right now" in our file? And if we want to do it in only one line, or in a specific range of lines? Well, we can do it.

To replace "now" by "right now" in the current line we can do `:s/now/right now`. Yeah, but what if we have more then one "now" to be replaced? The above command will only replace the first occurrence of the word. To replace all the occurrences in the line we can do `:s/now/right now/g`.

To replace in a number of lines we can do `:#, #s/now/right now`. This one isn't that clear in the generic way. So lets put it in an example. To replace "now" by "right now" in the lines 3,4,5 and 6, we can do `:3,6s/now/right now`. Remember that without the `/g` it will change only the first occurrence of each line.

To replace all occurrences in the whole file we can do `:%s/now/right now/g`. When we do such a radical substitution we may want to do it with more caution. To ask before any change we can do `:$s/now/right now/gc`.

I hope that the elements that compose the substitution command are now easy to use, since we can mix then to achieve the desired effect.

## Undo and Redo

Everybody makes mistakes. Most of programs have a <CTRL-Z> to undo those mistakes, and also VIM has, but with another shortcut. To undo our changes we can use `u`. We have in VIM a multi-level undo, and there is also something called undo tree, but I'll not talk about it because I don't know for sure how it works and most of the time it has no use.

And, as the counterpart of our undo, we have the Redo command **<CTRL-r>**. Easy and intuitive right? Remember that we already have replace under the **r** command.

## Other Commands

We can do some changes without entering the **Insert Mode**, that maybe useful, like to change the case, or shift the text to right or left. We can also see some information about the current file. Here's how:

- With the **~** command we can change letters from 'a' to 'A'.
- With the **<** and **>** command we can shift our text (many times it's easier to press **<<** and **>>** than **<space** or **>space**).
- We can Join lines, merge into a single one with **J**.
- With **<CTRL-g>** we can see the name, state, percentage of file read, and the total amount of lines the file has.

We can duplicate a part of a line with a weird command. When on **insert mode** we can press **<CTRL-y>** to duplicate the below character on the current cursor position. With **<CTRL-e>** we can duplicate the above character to the current cursor position. It may seem like useless, but sometimes, when writing code, it can be very useful.

## Fold Action

When we're programming sometimes we want to make that portion of code a little smaller. We could fold these line up to a single line, and since it's a very cool function to have on a text editor we'll have it here on VIM.

We make it enable with **:set foldenable**. Then we choose the fold method with **:set foldmethod=indent**, and obviously we can use other methods, these are: marker, manual, expr, syntax, diff, instead of indent.

And if we don't want this functionality anymore we can run **:set nofoldenable**. The common keys to manage folds are:

- **zc** - close fold.
- **zo** - open fold.
- **zM** - close all folds.
- **zR** - open all folds.
- **za** - open/close folds.

## Macros

Macros are a way to save and reproduce a repetitive task . We can, for example, change the case of the first letter of each line in 24 lines.

So... To start recording a macro just push the **q** key, some letter, and start doing stuff. It will record every move and you'll be able to redo every step.

We can in our example do **qb O~jq**. Let's break it down. **qb** starts the macro at letter b. **O** (zero) goes to the beginning of the line, **~** changes the case, **j** goes one line down, and **q** ends the recording. We can re-run that chain of commands with the **@(letter)**, and with the number-action we can do it as many times as we want. In our example we can do **24@b**, and it will do the job. You have 26 slots to your macros, since you have 26 letters.

If we want to add some step to our already recorded macro we can do it by record with an upper cased letter. It will append more commands to our macro.

## MARKS

You can set a mark at your text in a way that it's easier to go back to that point with a single command. You can set a mark with **:ma [character]**.

Marks works by saving the line and the column number, so you can jump to the exact place of the mark or to the beginning of the line that the mark is set. You can set a mark that will able you to jump between files, with an upper cased letter.

To jump to the exact place of the mark you can do **^(character)**. And to jump to the beginning of the line of that mark you can do **^[character]**. If you have two marks, you can switch between them with **``**.

You can list all your marks with **:marks**. Sometimes it won't jump right to the mark, VIM has many different versions, so you need to try it to find if it works perfectly fine.

## REGISTERS

The clipboard is bigger then just a slot to our cut and copy commands. We actually have all the letters, all the number digits, and some at special characters that have a special behavior. Marks and macros are storage at registers, so you may accidentally overwrite then, you need to be careful.

To save a piece of text we can do **"fyas**. Let's break it down. **"f** is to activate the **f** register. **yas** is to yank a sentence, in other words, we put a word in the **f** register. Other example is **"g3Y**, we yank 3 lines in the **g** register. Now we've saved we'll want to paste it. The command is **"gp** or **"gP** depending on what kind of paste you want to use.

To save into a register doesn't need to be a destructive action, we can append to a register new information. The trick is to use the upper cased letter with the **!** syntax. It's that simple. Now we can do up to 26 copy and paste parallel operations.

You can see all the slots in our registers, even those used in the storage of our macros and marks with the command **:registers**. You'll see that we have a kind of history of deleted things there, in the numbered registers. If you accidentally overwrite a clipboard you can use those to get them back.

## Command Mode

Many useful commands are under the command mode. The command mode can be entered by pressing **:** over the normal mode. The save, quit, help, and even set commands were already seen. Now we'll go a little further on it.

We can search outside of our file, for example, for a piece of text in all our files that are on the same directory with **:vimgrep (word) (pattern)**. For example, we can do **:vimgrep tomorrow \*.txt** to search for the word "tomorrow" in all .txt files.

We can save parts of our current file by selecting it with visual mode and then running **:w (filename)** and we can also append with **:w >> (filename)**. Or, if you want to add to your current file some external file, you can do **:read (file/filepath)**. With this we can go beyond editing only the current file.



When we use a command that may have options, like `:w (filename)` we can press `<CTRL-d>` and it will show us our options. Like this: `:w <CTRL-d>`.

You can also use `<TAB>` to navigate through options. VIM has some completion tools, but it will be seen later.

Many times, mainly when programming, we want to run the current file, and we could close VIM and use the terminal to run the program, but we can do something clever. We can do the `:! (command)` to run a command on the command line interface. For example, to run the current python script we can do `:! python3 %`. The `%` sign is an alias to the name of the current file.

If we need to go to the terminal while we're editing we can always use `<CTRL-z>` to stop vim. When finished at doing terminal stuff we can do `$ fg` and we're back again at VIM.

Its exhaustive to keep writing paths from our local directory. We can make it easier with `:cd` and `:pwd`.

## ABBREVIATIONS

When we're writing on insert mode we can let it set up some abbreviations that will substitute the current writing word to another word. If you abbreviate in a cool way, and with a lot of hard work, you'll be able to write as fast as it's possible, or even to write some sequence of letters that will write a whole skeleton of text to some html file for example.

You can create abrr's to you text with `:iabbrev ad advertisement`. When on insert mode, if you write "ad" it will automatically replace it. You can make it unabbreviated with `:unabbreviate ad`. And if you lose track of your abbreviations, found yourself lost and want to erase it all, you can remove all abbr's with `:abclear`. There are other ways to use abbreviations, but you'll always be able to find about it on the manual.

## MULTIFILE EDITING

There are 3 kinds of multi-file editing tools that can be used isolated or in compose with the others to create a layout of editing that are pretty pleasant.

### Buffers explanation

Everything is a buffer when we're editing. A buffer is the loaded file into the memory that allows you to edit. You can have more than one and navigate through it, it will work like another opened VIM. Every time we create a new file a new buffer is created, so as soon as you enter VIM with the `$ vim` command you're already at a buffer.

There is a buffer stack, and closing a buffer with `:q` will not remove it from there, but will turn them hidden. You can see what buffers are active on your session with the `:buffers` command or with the shortcut `:ls`.

### BUFFERS

We can open many files to edit. The first way to do it is with `:edit <file> <file> <file>`. It will open a buffer with all those files within. An alternative is `:e <file1> <file2> <file3>`, It's the same to run `:vim <file1> <file2> <file3>`.

Yes, many ways to load up to our buffer stack different files. If you want to edit another file, but not to write the changes in the current file yet, you can make it hidden with `:hide edit`



**foo.txt**. It will throw it to the buffer stack but with changes up to do. They will only take effect if you **:write all** the buffers.

- To open a new Buffer you can **:badd**. It stands for buffer add.
- To close the current Buffer you can do **:bdelete** or **:bd** as a shortcut.
- To change to next Buffer **:bnext** or **:bn** as a shortcut.
- To change to previous Buffer **:bprevious** or **:bp** as a shortcut.
- To change to the first/last Buffer **:bfirst** or **:bf** as a shortcut and **:blast** or **:bl** as a shortcut.
- To open all buffers (it will split the windows) **:ball**.

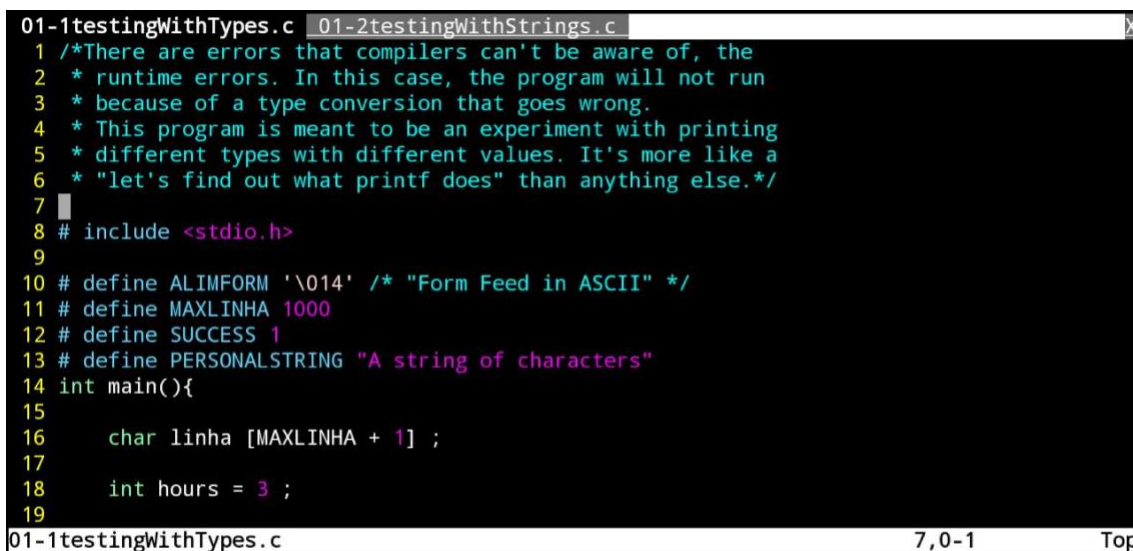
When into splitted screen after **:ball** you can quit the windows with **:q1**, or **:q2**, etc. The number will be the number of the buffer, you can see them with **:buffers** or **:ls**. Yet, the same can be done with **:bdelete**, but it will remove from our buffers stack.

You can see what files you're editing by **:args** and can choose another collection of files to edit without exiting VIM with **:args <file1> <file2> <file3>**.

## TABS

To understand what they are, you can think on it like a way to overlap windows, or, as we saw, overlap buffers.

- You can open a new tab with **:tabnew**.
- And close the current tab with **:tabclose**.
- Change between tabs with **:tabnext** and **:tabprevious**
- Also go to the first and last tabs with **:tabfirst** and **:tablast**.



```
01-1testingWithTypes.c 01-2testingWithStrings.c
1 /*There are errors that compilers can't be aware of, the
2 * runtime errors. In this case, the program will not run
3 * because of a type conversion that goes wrong.
4 * This program is meant to be an experiment with printing
5 * different types with different values. It's more like a
6 * "let's find out what printf does" than anything else.*/
7
8 # include <stdio.h>
9
10 # define ALIMFORM '\014' /* "Form Feed in ASCII" */
11 # define MAXLINHA 1000
12 # define SUCCESS 1
13 # define PERSONALSTRING "A string of characters"
14 int main(){
15
16     char linha [MAXLINHA + 1] ;
17
18     int hours = 3 ;
19
20 }
```

01-1testingWithTypes.c 7,0-1 Top

## SPLITTED WINDOWS

If you want to split our screen with a new file just go with **:new** or **:vnew**. It will open the new file on top position or on left position, later we'll see how to control where to open the splitted window.

To split the screen with a file from our path or our stack we **:split <filename>**, or as I rather do, let Ctrl + D show options. If only **:split** is given, it will copy the actual buffer.

A Vertical split is possible with **:vertical split** or **:vs** as a shortcut

To open as split an existing buffer **:sbuffer 1** or **:sb1** as a shortcut

```

2 ed.c 2 ld.c 3 us.c 3 us.c 2 ar.c 2 nt.c 3 nt.c 2 nt.c 2 nt.c tion.c 2 ne.c
1 /* print a Celsius-Fahrenheit conversion
table
2 * with a for loop.*/
3 main(){
4     int fahr ;
5     printf("This is my third program\n F
ahr \t Celsius\n") ;
6     for (fahr = 0; fahr <= 300; fahr = fa
hr + 20){
7         printf("%4d°F %6.1f°C\n", fahr, (
5.0 / 9.0) * (fahr - 32)) ;
8     }
9 }
10 /* On the first argument we see the lower
limit. On the second the upper limit.
11 * On the third the loop step.
12 * Its 3 main parts are: Start, if true c
ontinue, reboot with step
13 * This code is on page 24-25.
03-1HeaderFahrCelsius.c 1,1 Top
1 /* print a Celsius-Fahrenheit conversion
table
2 * with a for loop, inverted.*/
3 main(){
4     int fahr ;
5     printf("This is my fourth program\n F
ahr \t Celsius\n") ;
6     for (fahr = 300; fahr >= 0; fahr = fa
hr - 20){
03-3Constants.c 1,1 Top
1 /* This program is a example of symbolic
constants
2 * They're a way to remove meaningless nu
mbers from
3 * the code. This way we can use names in
stead of
4 * numbers. We're actually adding a level
of abstraction
5 * to make our code understandable*/
:tabprevious

```

I said that we would learn how to control where would our splitted screen open. We do it with these eight combinations. When we're splitting vertically we must focus on the "left/right" part of command, and when opening horizontally must focus on "above/below" and "top/bottom" part.

The top/bottom will throw our window to the edge of the screen, while the above/below will open it at the side of the currently active buffer.

Here are all 8 combinations:

window horizontal up --> :topleft split  
window horizontal down --> :botright split  
window vertical left --> :topleft vsplit  
window vertical right --> :botright vsplit  
buffer horizontal up --> :leftabove split  
buffer horizontal down --> :rightbelow split  
buffer vertical left --> :leftabove vsplit  
buffer vertical right --> :rightbelow vsplit

## MULTISPLIT MANIPULATION

- Ctrl+W J moves the current window to fill the bottom of the screen.
- Ctrl+W H moves the current window to fill left corner of the screen.
- Ctrl+W K move current window to top.
- Ctrl+W L move current window to far right.
- Ctrl+W x exchange current window with its neighbor.
- Ctrl+W r rotates the windows (imagine with 3 opened windows).
- Ctrl+W \_ maximize height of current window.
- Ctrl+W | maximize width of current window.
- Ctrl+W w cycle between the open windows.

- Ctrl+W h focus the window to the left.
- Ctrl+W j focus the window to the down.
- Ctrl+W k focus the window to the up.
- Ctrl+W l focus the window to the right.
- Ctrl+W t focus the window on top.
- Ctrl+W b focus the window on b.

## Change Size of Splits

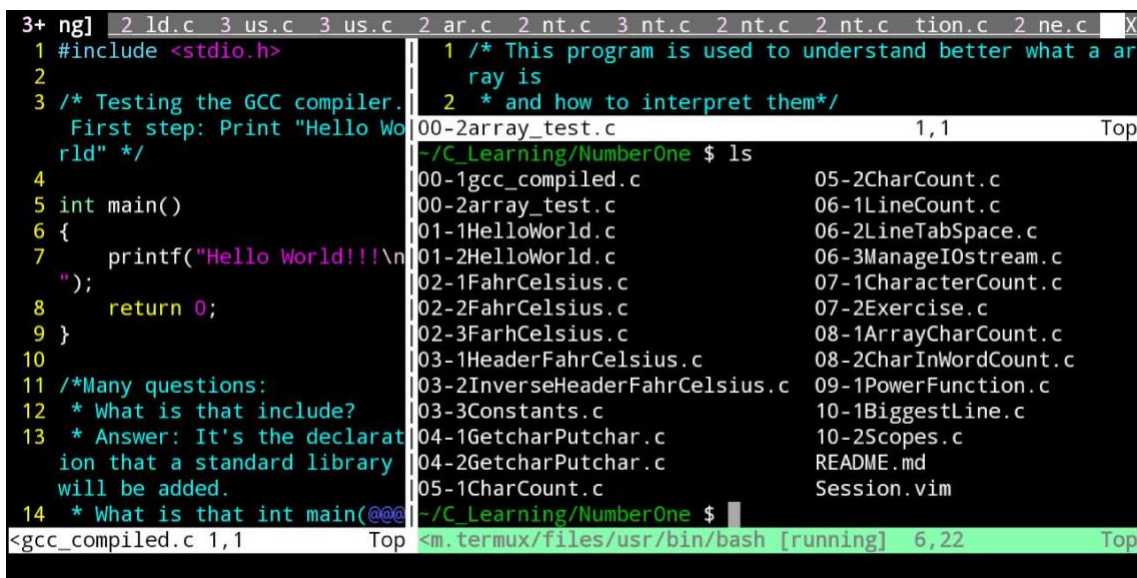
What if we want to change the size of our splitted screen? We can do it with the binding:

- Ctrl+W + to make the screen grow.
- Ctrl+W - to make the screen shrink.
- Ctrl+W = to make screens equally sized.

We can do it also with the command `:resize <new size>` or `:vertical resize <new size>`. You may want to use the `:resize +1` and `:resize -1`

## Splitting a terminal window

You can open a split with a terminal inside it with `:term`. The above/below left/right syntax works with it. There's only one side effect, that is once you're at the terminal, to do the commands you'll need to press "Ctrl+W :".



## The Mark of a Session

We saw that they are used to store text, our macros, but here's a thing: It can store our sessions at a file, or better saying, a kind of history of changes.

When we exit a VIM it creates a mark, it gets stored on the 0-8 mark names. We can see some of the old files with `:oldfiles` and edit them with `:e #<1`. It works with even older files. You can use it another way, with `:browse oldfiles` and it will lead you to choose the file you want.

## WORKING WITH SESSIONS

Sessions keeps file list, windows layout, global variables, options and others.

- How to create a session? `:mksession Session.vim`.
- How to restore a session? `:source vimbook.vim`.
- If you want to start VIM and restore a specific session? `vim -S Session.vim`.

It can be used to save the layout of splitted screen, so you'll not lose the work of configuring it. Remember to save the session again if you changed the layout, it don't have an auto save feature. You may need to force it with `mks!`

## WORKING WITH VIEW

Imagine it as a smaller version of sessions. It stores single windows. We operate with `:mkview` and to load it again we can `:loadview`. There's more about views, they can store up to 10 versions of a file, and can be stored with specific names.

## Version diff

When we're doing some version of our code often we'll need to see the difference \$ `vimdiff <file> <file>` or \$ `vim -d <file> <file>` then we can perform `:diffsplit` or even `:vert diffsplit`.

They will be with scroll binded, to remove it you can `:set noscrollbind` or you can put it back with `:set scrollbind`. We can update the diff with `:diffupdate` and jump between changes with

- `[c` - Jumps to previous change
- `]c` - Jumps to next change

We can apply changes in current diff window with `:diffget` and apply changes from current pane to another with `:diffput`

## Settings

```
1  "
2  "
3  "
4  "
5  "
6  "
7  "
8  "
9  "
10 "
11 " Author: Lucas Gouveia Belon, a brazilian guy.
12 " This file is a bunch of pieces tied together
13
14 let $RTP=split(&runtimepath, ',')[0]
15 let $RC="$HOME/.vim/vimrc"
16
17 syntax on          " Enable syntax processing
18 " syntax enable    " Turn the syntax to the standard?
19
20 filetype plugin indent on " Filetype detection + Filetype plugin + Filetype indent
vimrc                                     1,1                               Top
"vimrc" 150L, 5861B
```

VIM has a default configuration file that comes with it, it's on `/usr/share/vim/` on Linux systems. If you're using other systems you should run `:echo $VIMRUNTIME` and take a look at there. Many configuration files exists as dot files, so they keep hidden from our normal use, but since we're going to do a lot of changes in our VIM we'll need to see it in further details.

You can change the default configuration file and leave it there or you can use one of those two options:

- Keep a copy at your `~/` directory with the name `.vimrc`
- Create a folder called `~/vim/` and put `vimrc` there

Both ways work, and if you don't want to use and install and manage plugins, you can use the first alternative. But, to create a folder keeps it more organized and, if you decide to do some specific settings, this folder will be the home of other folders.

You should see the `:options` to get some configuration examples. You can use the `:help [option]` to get more information about it. After `:set` command, sometimes it will be needed to do `:source %` to make these settings to take effect if you're editing the `vimrc` file, else you'll need to `:source (pathToVimrc)`.

If you're doing `:set` on a session, it will only work on this run of VIM's editor, and if we want to make it permanent we can configure on `vimrc`.

Some configurations are, for example

- `:set nocompatible` – for no compatibility with Vi editor
- `:set spell!` – make the word correction (un)active
- `:set spelllang=en` – sets the language to English (pt-br for Brazilian Portuguese)

If we take a look at the already setup `.vimrc`, we'll find that we can remap our key bindings with `nnoremap`, and also make some commonly used commands to work with these key bindings, like `:nohlsearch`

- `nnoremap <leader><space> :nohlsearch` – will stop the highlight of the searched text.

Other common options to put on `vimrc` are

- `syntax on`
- `set number`
- `hlsearch`
- `set autowrite`
- `set wildmenu`

It's highly recommended to seek for `vimrc` files from other people. Many of those files has comments that will say what that configuration does.

## Carry your macros with you

Sometimes you'll want to reuse a macro and let it as standard in your VIM. And you can do it with a combination of the knowledge of macros, registers and the `vimrc`.

First of all you'll need to save a macro. Do you remember that every macro goes to a register slot? And we know how to paste that content. Whatever the content is, we can go to our `vimrc` file and paste as it follows:

```
let @M='(YourMacroPasted)'
```

Pay attention. The letter of the macro must be upper cased, and you'll find some weird characters when you paste the text of your macro, later you'll find that even the return/enter key has a key binding in VIM. The text of the macro must be inside the single quotes. And that's it, you'll retry a few times and find out how to do it correctly. It can save some time and you'll be interested on saving more complex macros.



## MODELINES

To make a option to run only on a specific file you can setup modelines.

```
/* vim:set shiftwidth=4: */
```

Put this on the first or last five lines in the file. You'll also want to put on your .vimrc **:set modelines=5** to make VIM inspect the first and last 5 lines to find modelines.

This is the format for modelines:

any-text vim:set {option}={value} ... : any-text

## Built-in Plugin

### NETRW

VIM has a browsing tool to navigate through our files (and more). Forget the **nerdtree** plugin, we will use **:edit** or we can use **:Explore**. It opens up a file browser that shows you many things. Press <F1> to get help on the things you can do in the netrw file browser.

When on the cursor atop a filename you can:

<enter>	Open the file in the current window.	netrw-cr
o	Horizontally split window and display file	netrw-o
v	Vertically split window and display file	netrw-v
p	Use the  preview-window	netrw-p
P	Edit in the previous window	netrw-P
t	Open file in a new tab	netrw-t
X	Executes the file under the cursor	

### OMNICO completion

VIM has a completion function that works on insert mode. You can press **<CTRL-x>** on insert mode to choose the category of completion that you want. You can also use **<CTRL-n>** and **<CTRL-p>** to navigate through suggestions. To use Omni completion you'll need to add to your vimrc file the **set omnifunc=syntaxcomplete#Complete** to enable it to work all the time.