C editing with VIM HOWTO

Siddharth Heroor

Revision History

Revision v1.0	Jan 14, 2001	Revised by: sh
Second Revision. Corrected some typos.		
Revision v0.1	Dec 04, 2000	Revised by: sh
First Revision. I would love to have your feedback		

This document gives an introduction to editing C and other language files, whose syntax is similar, like C++ and Java in vi/VIM.

Table of Contents

1. Introduction

The purpose of this document is to introduce the novice VIM user to the editing options available in VIM for C files. The document introduces some commands and keystrokes which will help in improving the productivity of programmers using VIM to edit C files.

The scope of the document is to describe how one can edit C files with VIM. However most of what is described is also applicable for vi. Plus what is mentioned here about editing C files is more or less applicable to C++, Java and other similar languages.

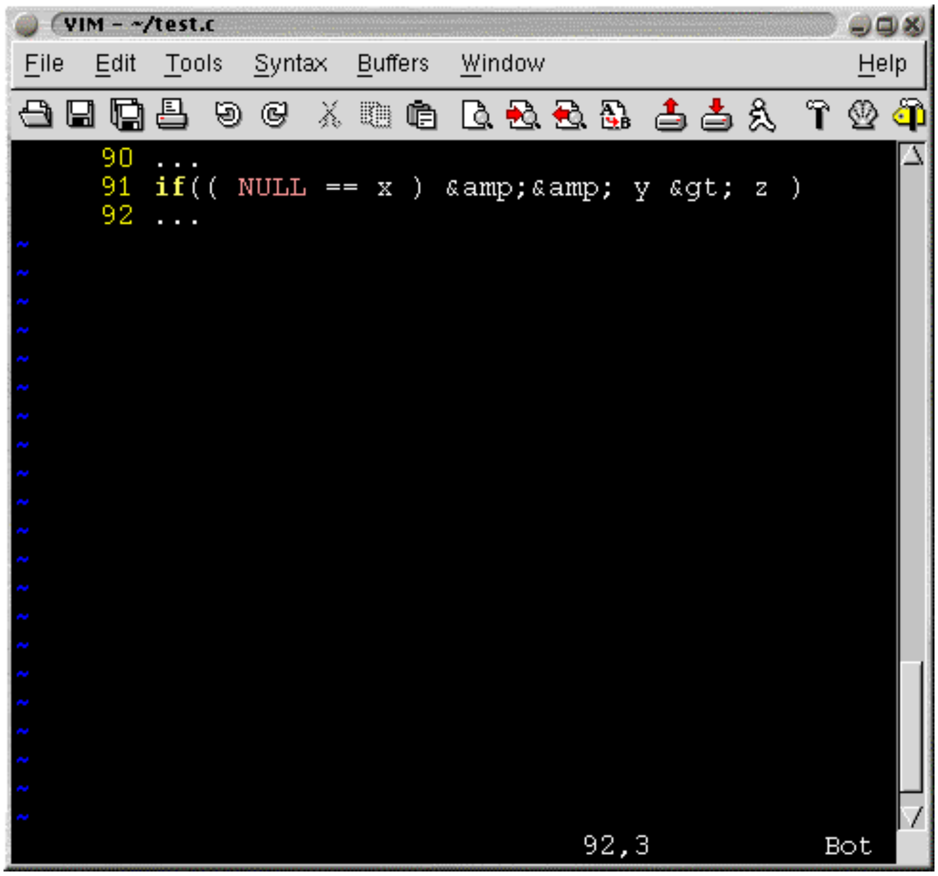
2. Moving around.

2.1. w, e and b keystrokes

One can use the **w**, **e** and **b** keys to move around a file. VIM is capable of recognizing the different tokens within a C expression.

Consider the following C code

Figure 1. A C snippet




```
...
if(( NULL == x ) 38;38; y > z )
...
```

Assume that the cursor is positioned at the beginning of the **if** statement. By pressing w once the cursor jumps to the first (. By typing **w** again the cursor moves to **NULL**. Next time the cursor will move to the **==** token. Further keystrokes will take you as follows. **x**...)... **&&**... **y**... **>**... **z**... and finally **)**...

e is similar to **w** only that it takes you to the end of the current word and not to the beginning of the next word.

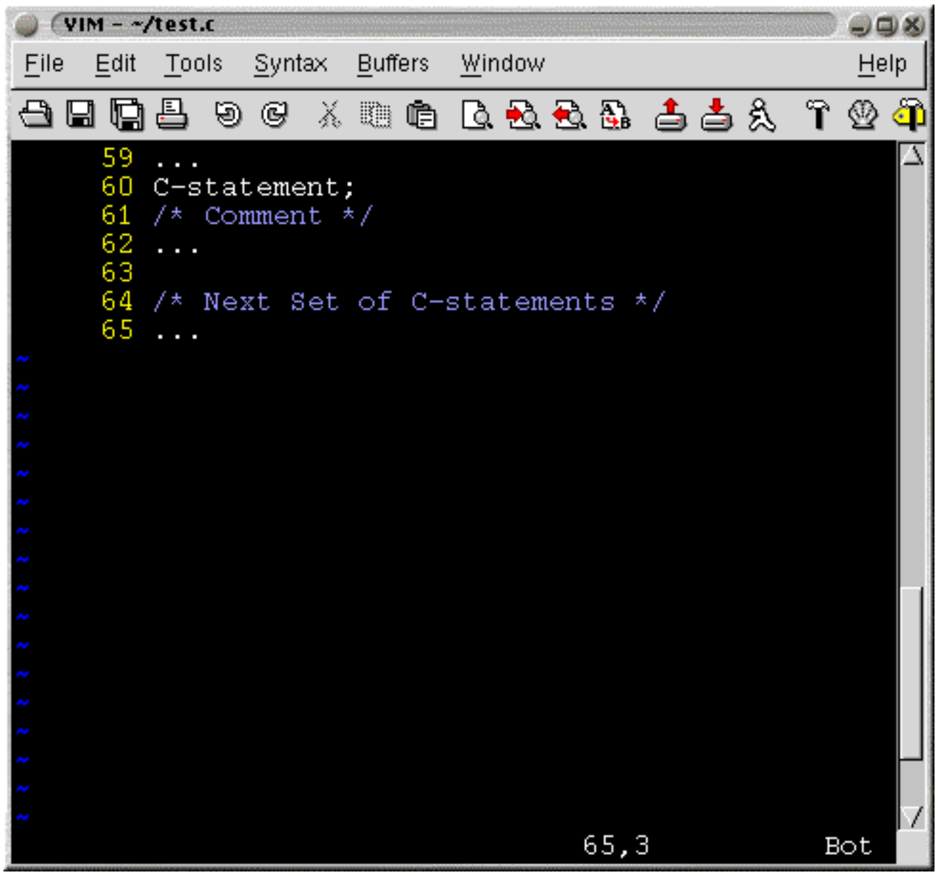
b does the exact opposite of **w**. It moves the cursor in the opposite direction. So you can moving backwards using the **b** keystroke.

2.2. {, }, [[and]] keystrokes

The **{** and **}** keys are used to move from paragraph to paragraph. When editing C files these keys have a slightly different meaning. Here a paragraph is taken as a bunch of lines separated by an empty line.

For Example

Figure 2. Another C snippet

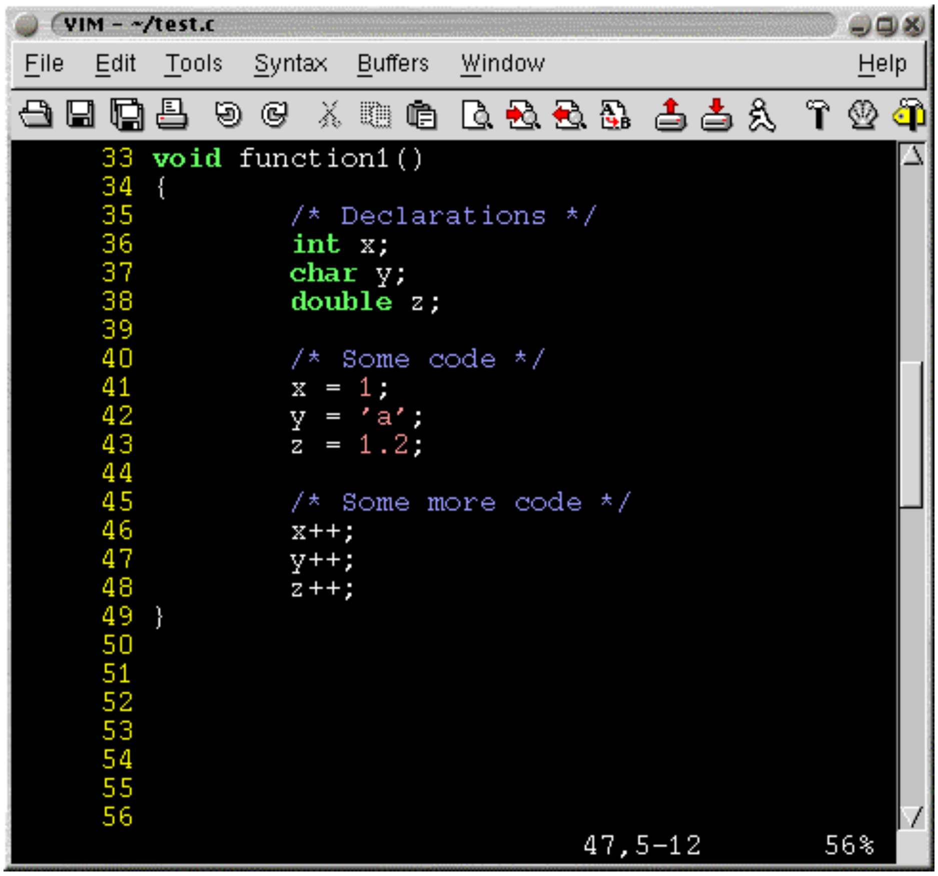


The above snippet shows two paragraphs. One can easily move from the beginning of one to the other, by using the **{** and **}** keys. **{** will take the cursor to the paragraph above and **}** will take the cursor to the paragraph below.

Many people have the coding style where a logical set of statements are grouped together and separated by one or more blank lines.

For Example

Figure 3. Another C snippet

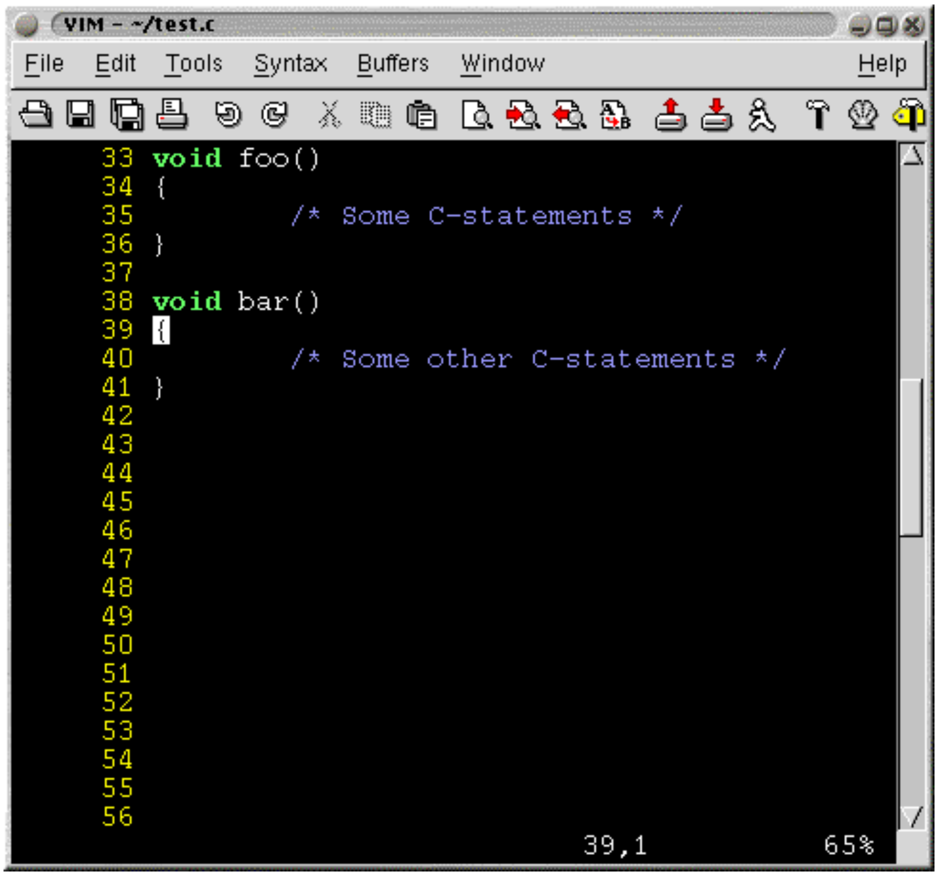


The **{** and **}** keys are very useful in such situations. One can very easily move from one "paragraph" to another.

Another set of keys which are useful are the **[[** and **]]** keys. These keys allow you to jump to the previous { or next { in the first column.

For Example

Figure 4. The next snippet of C code



Lets say you were editing foo() and now you want to edit bar(). Just type **]]** and the cursor will take you to the opening { of the bar() function. The reverse is slightly different. If you were in the middle of bar() and you type **[[** the cursor will move to the first { above i.e. the beginning of bar() itself. One has to type **[[** again to move to the beginning of foo(). The number of keystrokes can be minimized by typing **2[[** to take the cursor to the beginning of the previous function.

Another set of similar keystrokes are the **][** and **[]** keystrokes. **][** takes the cursor to next } in the first column. If you were editing foo() and wanted to go to the end of foo() then **][** will take you there. Similarly if you were editing bar() and wanted to go to the end of foo() then **[]** would take the cursor there.

The way to remember the keystrokes is by breaking them up. The first keystroke will indicated whether to move up or down. **[** will move up and **]** will move down. The next keystroke indicates the type of brace to match. If it same same as the previous keystroke then the cursor will move to {. If the keystroke is different then the cursor will move to }.

One caveat of the **]]**, **][**, **[[** and **[]** keystrokes is that they match the braces which are in the *first column*. If one wants to match all braces upwards and downwards regardless of whether its in the first column or not is not possible. The VIM documentation has a workaround. One has to map the keystrokes to find the braces. Without spending too much time on mapping, the suggested mappings are

:map [[?{<CTRL−VCTRL−M>w99[{

:map][/}<CTRL−VCTRL−M>b99]}

:map]] j0[[%/{<CTRL−VCTRL−M>

:map [] k$][%?}<CTRL−VCTRL−M>

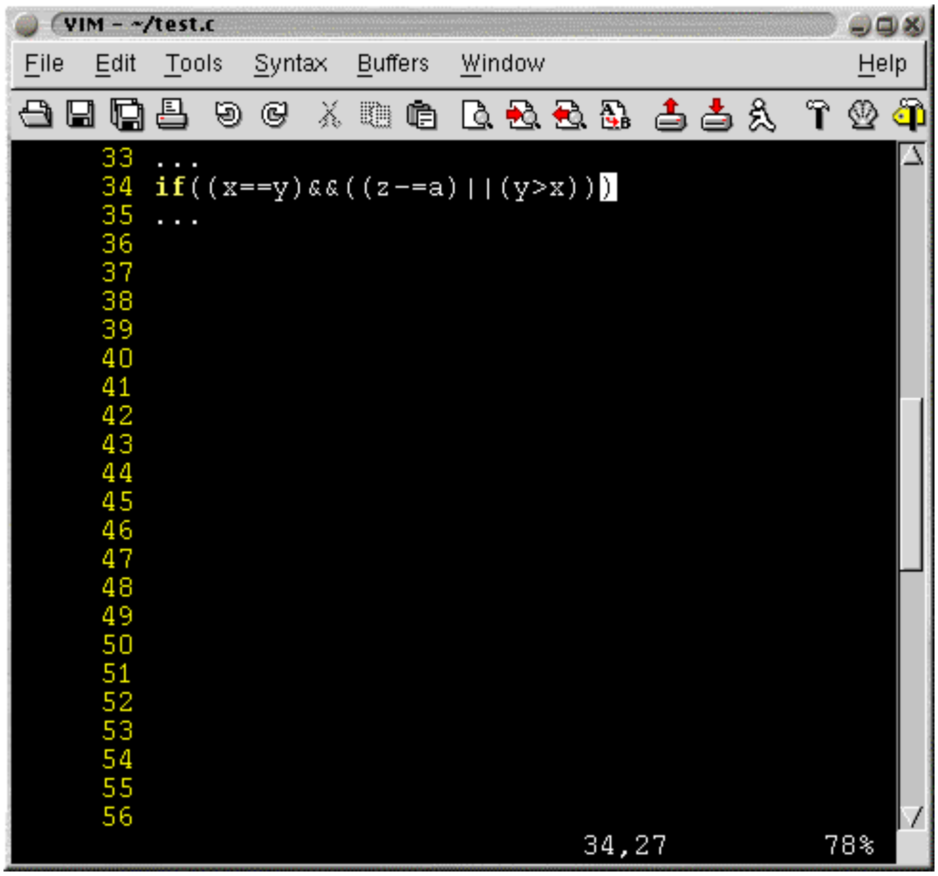
2.3. % keystroke

The **%** is used to match the item under the cursor. The item under the cursor can be a parenthesis, a curly bracket or a square bracket. By pressing the % key the cursor will jump to the corresponding match.

Amongst other things, the **%** keystroke can be used to match #if, #ifdef, #else #elif and #endif also.

This keystroke is very useful in validating code that one has written. For Example

Figure 5. The next snippet of C code



Checking the above code will involve checking the correctness of the parenthesis. The **%** can be used to jump from one (to its corresponding) and vice versa. Thus, one can find which opening parenthesis corresponds to which closing parenthesis and use the information to validate the code.

Similarly the **%** can also be used to jump from a { to its corresponding }.

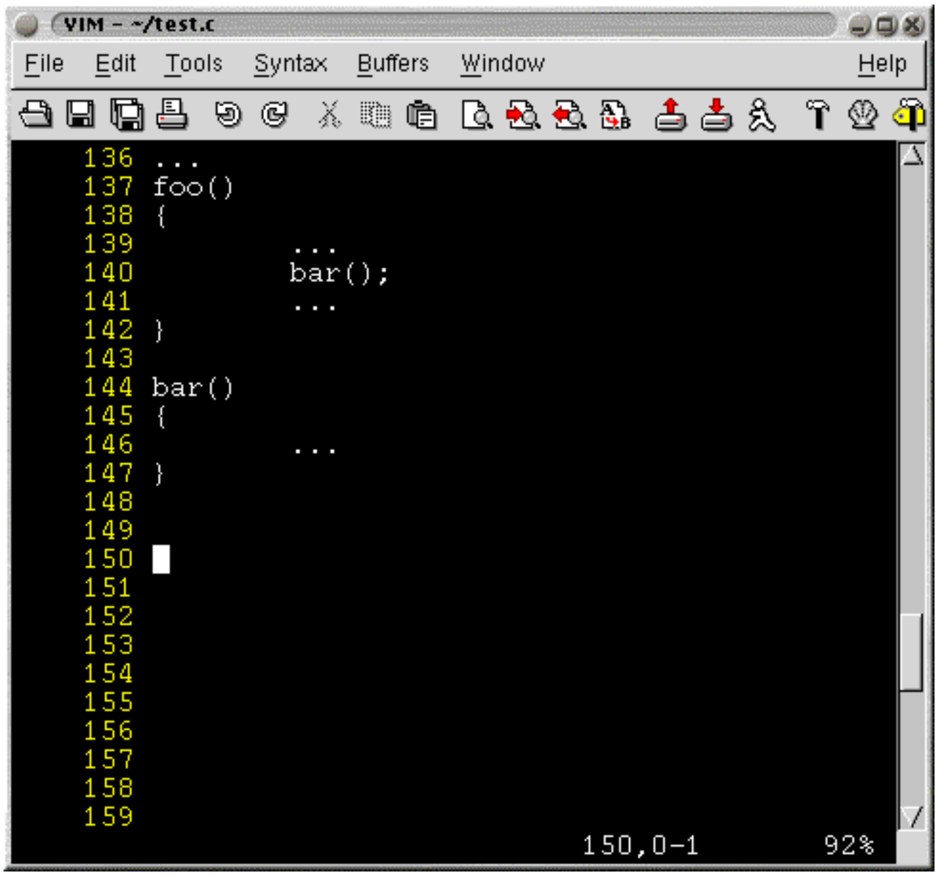
3. Jumping to random positions in C files

3.1. ctags

A Tag is a sort of placeholder. Tags are very useful in understanding and editing C. Tags are a set of book–marks to each function in a C file. Tags are very useful in jumping to the definition of a function from where it is called and then jumping back.

Take the following example.

Figure 6. Tags Example



Lets say that you are editing the function foo() and you come across the function bar(). Now, to see what bar() does, one makes uses of Tags. One can jump to the definition of bar() and then jump back later. If need be, one can jump to another function called within bar() and back.

To use Tags one must first run the program ctags on all the source files. This creates a file called tags. This file contains pointers to all the function definitions and is used by VIM to take you to the function definition.

The actual keystrokes for jumping to and fro are **CTRL–]** and **CTRL–T**. By hitting **CTRL–]** in foo() at the place where bar() is called, takes the cursor to the beginning of bar(). One can jump back from bar() to foo() by just hitting **CTRL–T**.

ctags are called by

```
$ ctags options file(s)
```

To make a tags file from all the *.c files in the current directory all one needs to say is

```
$ ctags *.c
```

In case of a source tree which contains C files in different sub directories, one can call ctags in the root directory of the source tree with the −R option and a tags file containing Tags to all functions in the source tree will be created. For Example.

```
$ ctags −R *.c
```

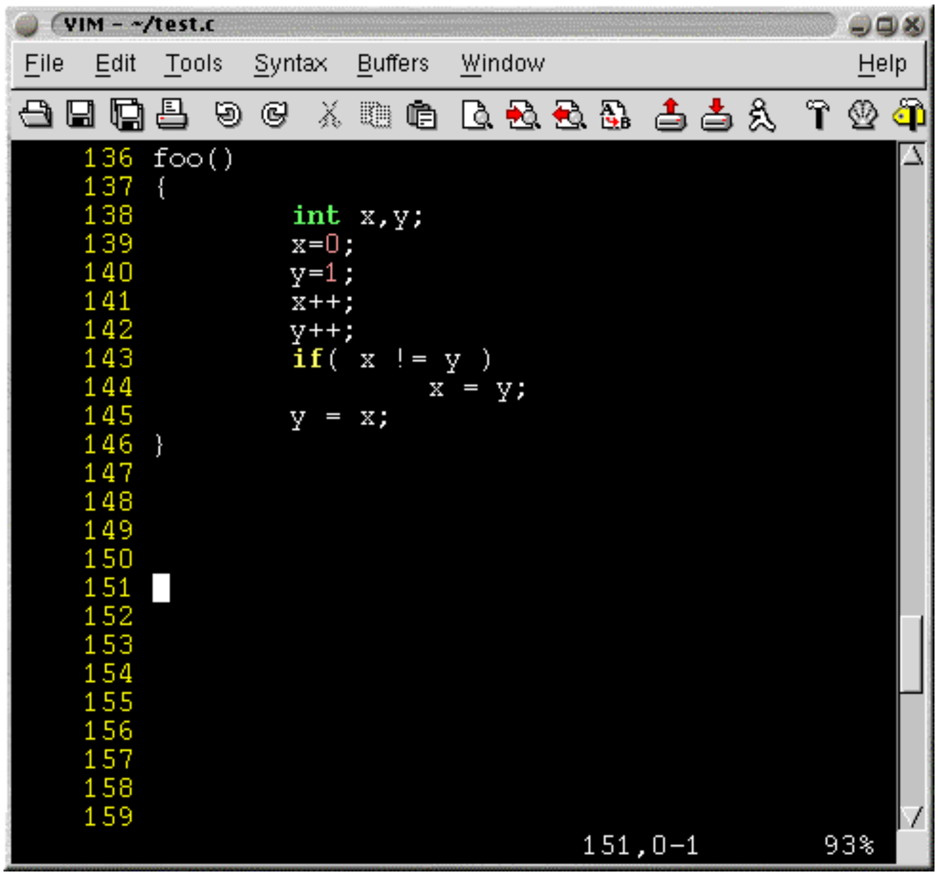
There are many other options to use with ctags. These options are explained in the man file for ctags.

3.2. marks

Marks are place−holders like Tags. However, marks can be set at any point in a file and is not limited to only functions, enums etc.. Plus marks have be set manually by the user.

By setting a mark there is no visible indication of the same. A mark is just a position in a file which is remembered by VIM. Consider the following code

Figure 7. The marks example



Suppose you are editing the line x++; and you want to come back to that line after editing some other line. You can set a mark on that line with the keystroke **m'** and come back to the same line later by hitting **''**.

VIM allows you to set more than one mark. These marks are stored in registers a–z, A–Z and 1–0. To set a mark and store the same in a register say j, all one has to hit is **mj**. To go back to the mark one has to hit **'j**.

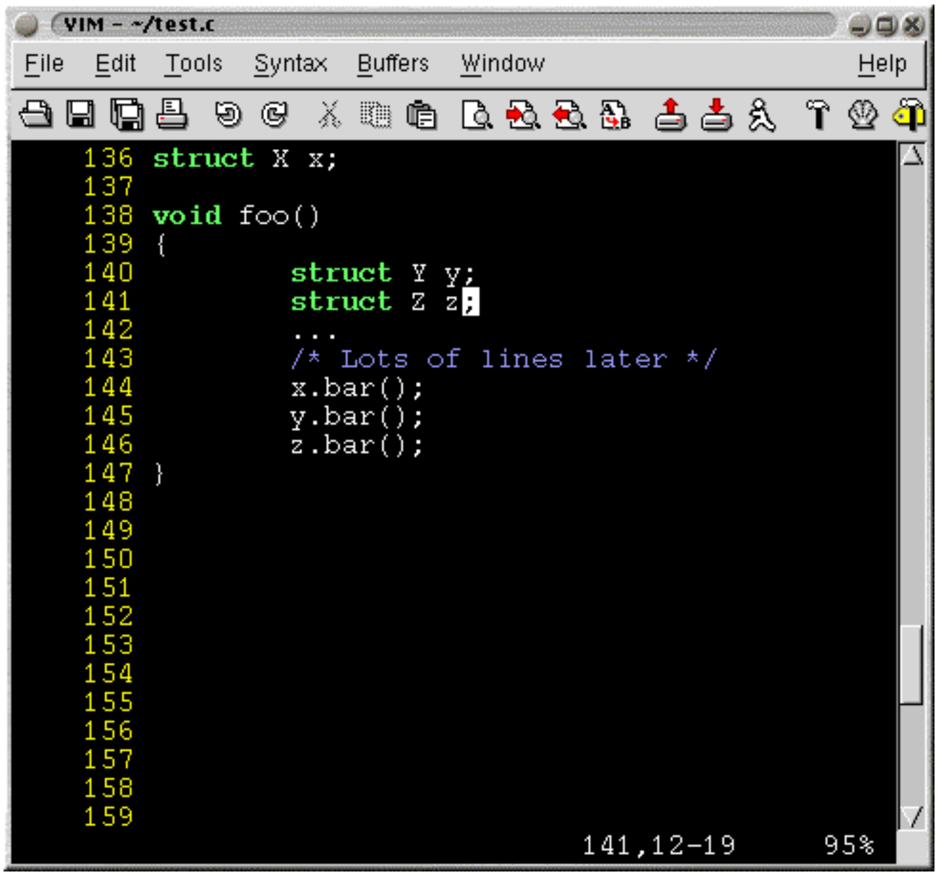
Multiple marks are really useful in going back and fro within a piece of code. Taking the same example, one might want one mark at x++; and another at y=x; and jump between them or to any other place and then jump back.

Marks can span across files. To use such marks one has to use upper–case registers i.e. A–Z. Lower–case registers are used only within files and do not span files. That's to say, if you were to set a mark in a file foo.c in register "a" and then move to another file and hit **'a**, the cursor will not jump back to the previous location. If you want a mark which will take you to a different file then you will need to use an upper–case register. For example, use **mA** instead of **ma**. I'll talk about editing multiple files in a later section.

3.3. gd keystroke

Consider the following piece of code.

Figure 8. The third example



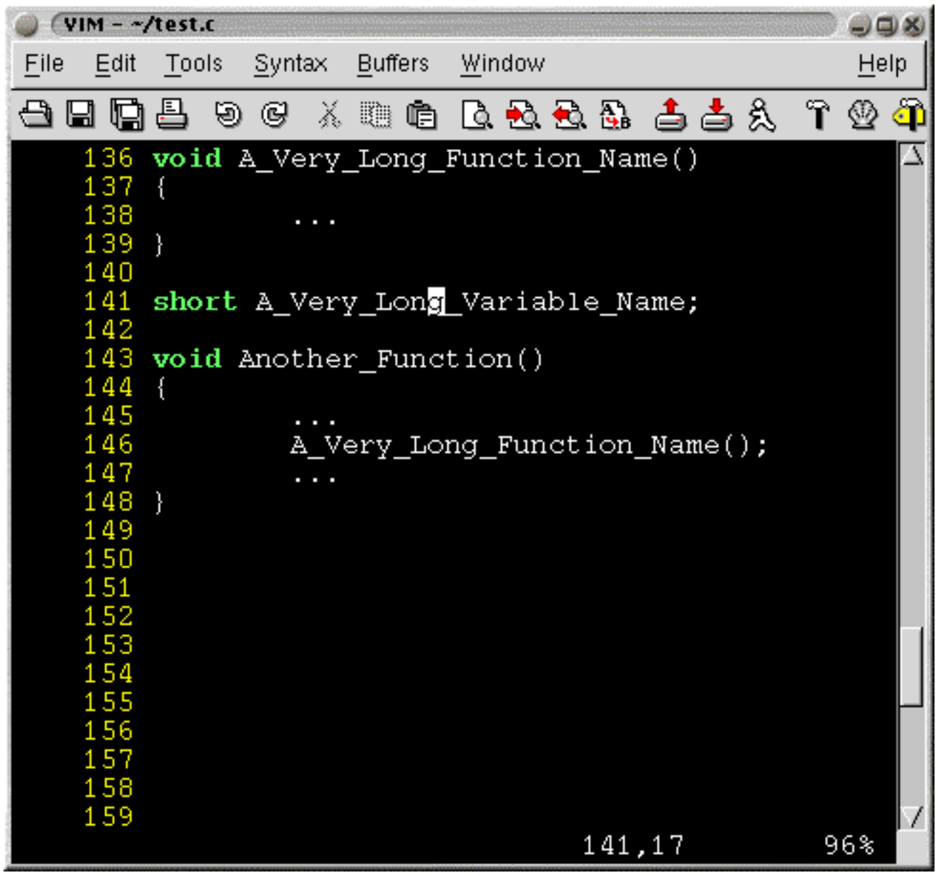
For some reason you've forgotten what y and z are and want to go to their declaration double quick. One way of doing this is by searching backwards for y or z. VIM offers a simpler and quicker solution. The **gd** keystroke stands for Goto Declaration. With the cursor on "y" if you hit **gd** the cursor will take you to the declaration :– struct Y y;.

A similar keystroke is **gD**. This takes you to the global declaration of the variable under the cursor. So if one want to go to the declaration of x, then all one needs to do is hit **gD** and the cursor will move to the declaration of x.

4. Auto–Completing Words

Consider the following code

Figure 9. Auto–completion example



The function A_Very_Long_Function_Name() can be quite exasperating to type over and over again. While still in insert–mode, one can auto–complete a word by either searching forwards or backwards. In function, Another_Function() one can type A_Very... and hit **CTRL–P**. The first matching word will be displayed first. In this case it would be A_Very_Long_Variable_Name. To complete it correctly, one can hit **CTRL–P** again and the search continues upwards to the next matching word, which is A_Very_Long_Function_Name. As soon as the correct word is matched you can continue typing. VIM remains in insert–mode during the entire process.

Similar to **CTRL–P** is the keystroke **CTRL–N**. This searches forwards instead of backwards. Both the keystrokes continue to search until they hit the top or bottom.

Both **CTRL–P** and **CTRL–N** are part of a mode known as CTRL–X mode. CTRL–X mode is a sub–mode of the insert mode. So you can enter this mode when you are in the insert–mode. To leave CTRL–X mode you can hit any keystroke other than CTRL–X, CTRL–P and CTRL–N. Once you leave CTRL–X mode you return to insert–mode.

CTRL–X mode allows you do auto–completion in a variety of ways. One can even autocomplete filenames.

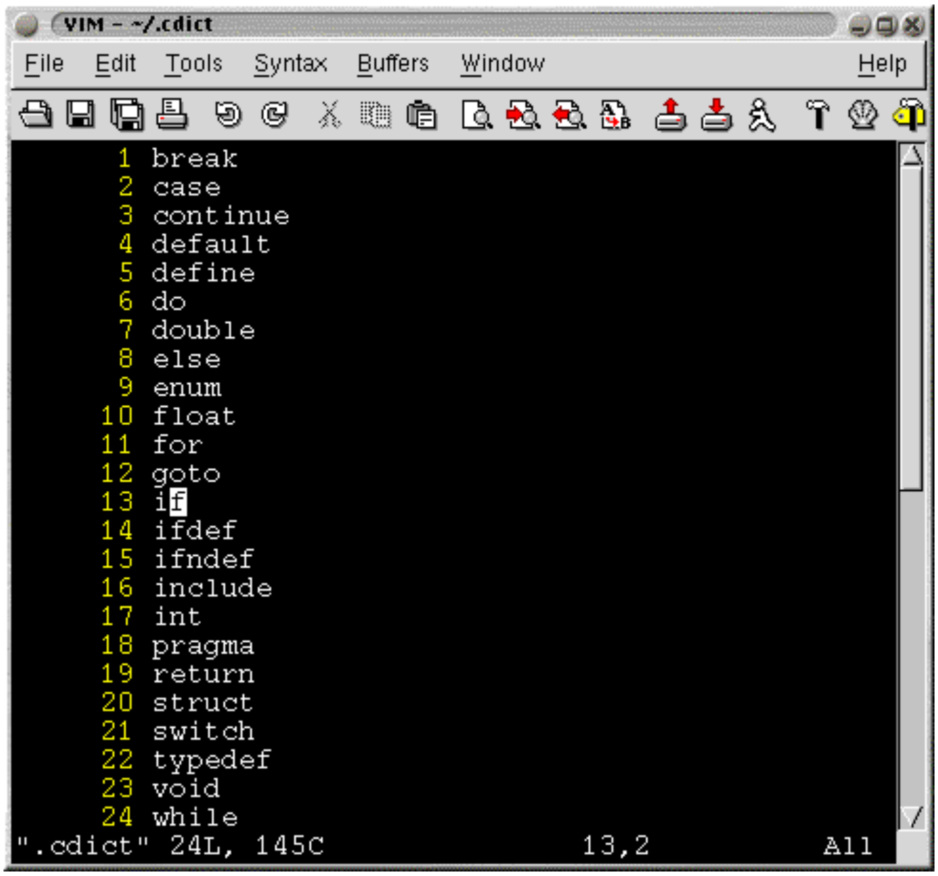
This is particularly useful when you have to include header files. Using CTRL–X mode you can include a file foo.h using the following mechanism.

```
        #include "f CTRL-X CTRL-F"
```

That's CTRL–X CTRL–F. I know... I know... Its sounds like emacs ;–). There are other things you can do in the CTRL–X mode. One of them is dictionary completion. Dictionary completion allows one to specify a file containing a list of words which are used for completion. By default the dictionary option is not set. This option is set by the command **:set dictionary=file**. Typically one can put in C keywords, typedefs, #defines in the dictionary file. C++ and Java programmers may be interested in adding class names as well.

The format of a dictionary file is simple. Just put a word you want in line by itself. So a C dictionary file would look something like this.

Figure 10. A sample dictionary file



To use the dictionary completion, one needs to hit **CTRL–X CTRL–K**. The completion is similar to the **CTRL–P** and **CTRL–N** keystrokes. So... to type "typedef" all one needs to do is t CTRL–X CTRL–K and poof... the name completed.

5. Formating automatically

5.1. Restricting column width

One often has to restrict the column width to 80 or 75 or whatever. One can set this quite easily by using the command

```
:set textwidth=80
```

To do this automatically just put the command in your .vimrc.

In addition to textwidth you may want the text to wrap at a certain column. Often such choices are dictated by the terminal one is using or it could just be by choice. The command for such a case is

```
:set wrapwidth=60
```

The above command makes the text wrap at 60 columns.

5.2. Automatically indent code

While coding in C, one often indents inner−blocks of code. To do this automatically while coding, VIM has an option called cindent. To set this, just use the command

```
:set cindent
```

By setting cindent, code is automatically beautified. To set this command automatically, just add it to your .vimrc

---

5.3. Comments

VIM also allows you to auto−format comments. You can split comments into 3 stages: The first part, the middle part and the end part. For example your coding style requirements may require comments to be in the following style

```
            /*
            * This is the comment
            */
```

In such a case the following command can be used

```
:set comments=sl:/*,mb:*,elx:*/
```

Let me decipher the command for you. The commands has three parts. The first part is sl:/*. This tells VIM that three piece comments begin with /*. The next part tells VIM that the middle part of the comment is *. The last part of the command tells vim a couple of things. One that the command should end with */ and that it should automatically complete the comment when you hit just /.

Let me give another example. Lets say your coding guidelines are as follows

```
            /*
            ** This is the comment
            */
```

In such a situation you can use following command for comments

```
:set comments=sl:/*,mb:**,elx:*
```

to insert a comment just type /* and hit enter. The next line will automatically contain the **. After you've finished the comment just hit enter again and another ** will be inserted. However to end the comment you want a */ and not **/. VIM is quite clever here. You don't need to delete the last * and replace it with /. Instead, just hit / and VIM will recognise it as the end of the comment and will automatically change the line from ** to */.

For more info hit **:h comments**

6. Multi−file editing

One often needs to edit more than one file at a time. For example one maybe editing a header file and a source file at the same time. To edit more than one file at a time, invoke VIM using the following command

```
$ vim file1 file2 ...
```

Now you can edit the first file and move onto the next file using the command

```
:n
```

You can jump back using the command

```
:e#
```

It may be useful while coding if you could see both the files at the same time and switch between the two. In other words, it would be useful if the screen was split and you could see the header file at the top and the source file at the bottom. VIM has such a command to split the screen. To invoke i, simply say **:split**

The same file will be displayed in both the windows. Whatever command is invoked, will affect only the window in focus. So one can edit another file in another window by using the command **:e file2**

After executing that command, you'll find that there are two files visible. One window shows the first file and the other shows the second file. To switch between the files one has to use the keystroke **CTRL−W CTRL−W**. To learn more about split windows, just run help on it.

7. Quickfix

When coding in C one often has a edit−compile−edit cycle. Typically you would edit C file using some the things I've mentioned earlier, save the file, compile the code and go to the error(s) and start editing again. VIM helps save the cycle time slightly using a mode called quickfix. Basically, one has to save the compiler errors in a file and open the file with VIM using the command

```
$ vim −q compiler_error_file
```

VIM automatically opens the file containing the error and positions the cursor at the location of the first error.

There is a shortcut to the cycle. Using the command "make", one can automatically compile code and goto the position where the first error occurs. To invoke the make command type the following

```
:make
```

Basically, this command calls make in a shell and goes to the first error. However, if you are not compiling using make and are compiling using a command such as cc, then you have to set a variable called makeprg to the command you want invoked when you use the make command. For eg. **:set makeprg=cc\ foo.c**

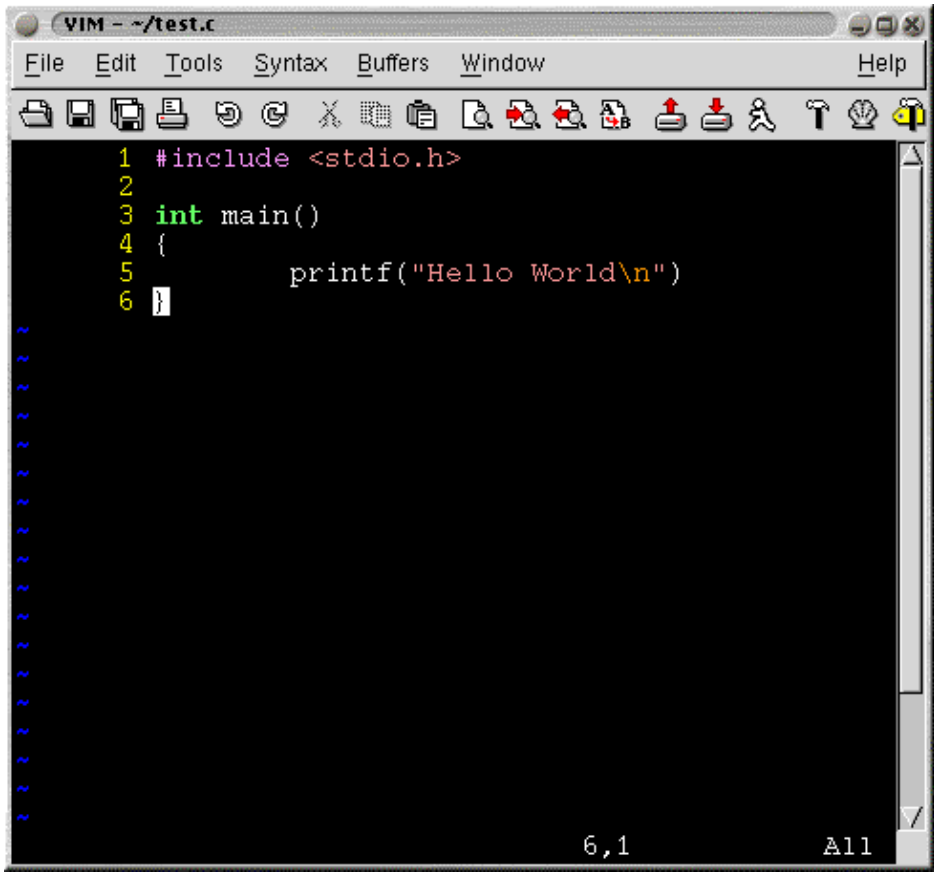
After setting makeprg, you can just call the make command and quickfix will come into play.

After you have corrected the first error, the next thing to do would be go to the next error and correct that. The following command is used go to the next error. **:cn**

To go back, you can use the command **:cN**

Let me demonstrate this using an example. Consider the following code

Figure 11. Quickfile Program Listing

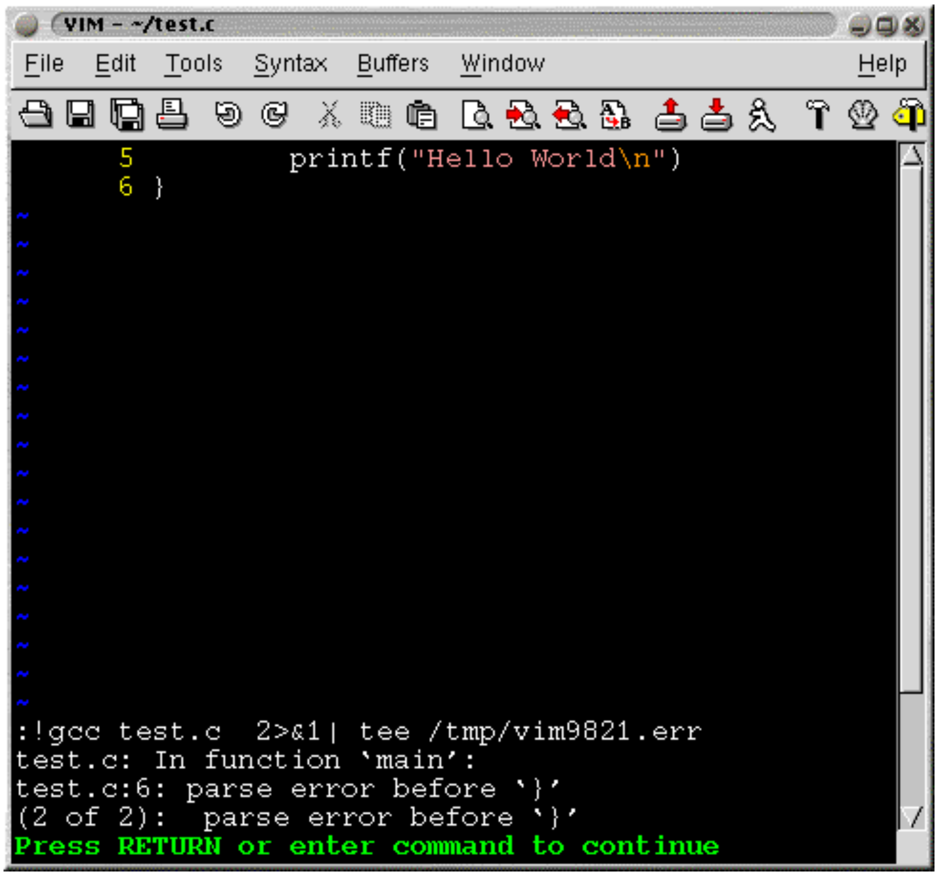


As you can see there is an error on line number 5. The file is saved as test.c and makeprg is set using the command

```
:set makeprg=gcc\ test.c
```

Next the make command is invoked using the command **:make**. gcc gives an error and the output of the make command is something like this

Figure 12. :make error



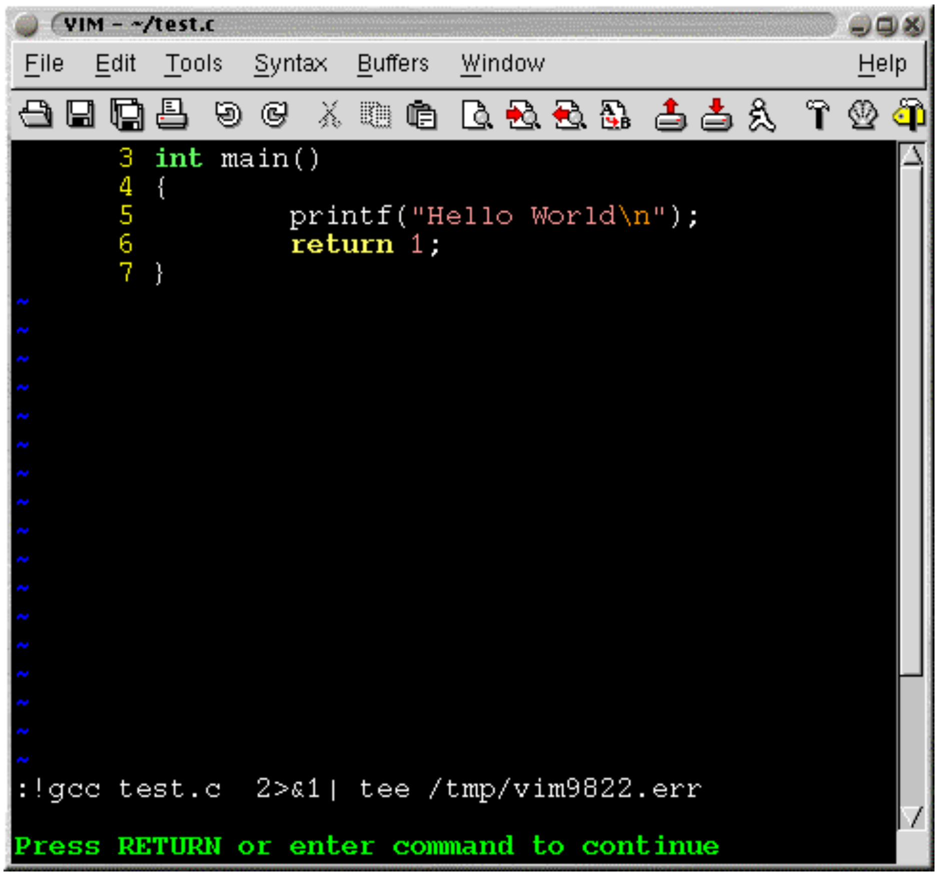
On pressing **RETURN**, the cursor moves to line number 6.

Now, the command **:cn** will move the cursor to the line number 4.

To move back to the previous error, one can use the command **:cN** and the cursor will move back to the line 6.

After correcting the error on line 5 and adding "return 1;", one can run **:make** again and the output will be

Figure 13. No Error



That was just a small example. You can use quickfix to solve your compile time problems and hopefully reduce the edit–compile–edit cycle.

---

8. Copyright

9. References

You can get more information on VIM and download it at http://www.vim.org