Vim for C Programmers

By Girish Venkatachalam
Created 2005-10-31 02:00

You don't have to move to an integrated development environment to get luxury coding features. From variable autocompletions all the way up to integration with ctags and make, Vim makes a C programmer's life easier and more productive.

Vim is an extremely powerful editor with a user interface based on Bill Joy's almost 30-year-old vi, but with many new features. The features that make Vim so versatile also sometimes make it intimidating for beginners. This article attempts to level the learning curve with a specific focus on C programming.

make and the Compile-Test-Edit Cycle

A typical programmer's routine involves compiling and editing programs until the testing proves that the program correctly does the job it is supposed to do. Any mechanism that reduces the rigor of this cycle obviously makes any programmer's life easier. Vim does exactly that by integrating **make** with Vim in such a way that you don't have to leave the editor to compile and test the program. Running `:make` from inside of Vim does the job for you, provided a makefile is in the current directory.

You can change the directory from inside of Vim by running `:cd`. To verify where you are, use `:pwd`. In case you are using FreeBSD and want to invoke gmake instead of make from the command line, all you have to do is enter `:set makeprg=gmake`. Now say you want to give some parameters to make. If, for instance, you want to give CC=gcc296:

```
:set makeprg=gmake\ \CC=gcc296
```

does the job.

Now comes the job of inspecting the errors, jumping to the appropriate line number in the source file and fixing them. If you want to display the line numbers in the source file, `:se nu` turns on this option, and `:se nonu` disables line number display.

Once you compile, Vim automatically takes you to the first line that is causing the error. To go to the next error; use `:cn` to take you to the next line number causing the error. `:cfirst` and `:clast` take you to the first error and the last error, respectively. Once you have fixed the errors, you can compile again. If you want to inspect the error list again, `:clist` displays it. Convenient, isn't it?

If you want to read some other source file, say foo.c, while fixing a particular error, simply type `:e foo.c`.

One shortcut provided by Vim to avoid typing too much to switch back to the previous file is to type `:e #` instead of typing the full path of the file. If you want to see all of the files you have opened in Vim at any point in time, you can use `:ls` or `:buffers`.

If you have a situation in which you have opened too many files and you want to close some of them, you can use `:ls`. It should display something like this:

```
2 #    "newcachain.c"                   line 5
3 %a   "cachain.c"                      line 1
```

If you want to close newcachain.c, `:bd 2` or `:bd newcachain.c` does the job.

While browsing C code, you may have situations in which you want to skip multiple functions fast. You can use the]] key combination for that while in command mode. If you want to browse backward in the file, the command is [[.

You also can use marks to bookmark certain cursor positions. You can use any lowercase alphabet character as a mark. For instance, say you want to mark line number 256 of the source and call it b. Simply go to that line, `:256`, and type `mb` in command mode. Vim never echoes what you type in command mode but silently executes the commands for you.

If you want to go to the previous position, typing `''` (two single-quotation marks) takes you there. Typing `'a` takes you to mark a and so on.

Especially when editing Makefiles, you may want to figure out which of the white spaces are tabs. You can type `:se list`, and whatever is displayed as `^I` in blue are tabs. Another way to do that is to use `/\t`. This highlights the tabs in yellow.

Global searches and replaces are common tasks for programmers, and Vim provides good support for both. Simply type `/` in command mode, and you are taken to the searched keyword. If you prefer incremental searches, à la emacs, you can specify `:se incsearch` before you search. When you want to disable it, type `:se nois`.

Search and replace is a powerful tool in Vim. You can execute it only on a region that you selected using the v command, only between certain line numbers or only in rectangular regions selected with the Ctrl-V command.

Once you select your region or line number ranges, for example using `:24,56` to select lines 24-56 (both inclusive), type `s/foo/bar` to replace all occurrences of the string foo with bar.

But, this command replaces only one instance per line. If you want to do this for multiple occurrences per line, type `s/foo/bar/g`. If you want to replace only some occurrences, you can use the "confirm" option with `s/foo/bag/gc`.

Sometimes the string contains characters that appear as a substring of other keywords. For instance, say you want to replace the variable "in" and not the "in" in inta. To search for whole words, type `/\<in\>/`.

Most commonly, you will want to do a global replace, which is every instance in a given file. You can do that by using either `:1,$s/foo/bar/g` or `:%s/foo/bar/g`. If you then want to replace this in all the files you have open, you can enter `:bufdo %s/foo/bar/g`.

Another way of searching is by going to the keyword and typing * in command mode. The keyword now will be highlighted wherever it occurs in the file. Searching backward is simple too; type ? instead of / while searching.

Once the searching is over, Vim remembers it, so the next time you search for the same keyword, you have to type only / or ?, instead of typing the whole text.

One side effect of searching is that it stays highlighted. This can be a distraction while editing programs. Turn highlighting off by typing `:se nohlsearch`, `:nohlsearch` or `:nohl`

You always can use the Tab key to complete Vim commands you give with a colon. For instance, you can type `:nohl`<Tab>, and Vim completes it for you. This is applicable generically, and you can press Tab to cycle through Vim's commands until Vim finds a unique match.

Vim with Exuberant ctags

Exuberant ctags (see the on-line Resources) is an external program that can generate tags for Vim to navigate source code. If all of your source code is contained in only one directory, simply go to the directory in the shell and enter:

```
$ ctags .
```

This generates a tags file called tags. Vim reads this file for jumping to functions, enums, #defines and other C constructs.

If the source code is distributed across several directories, ctags has to generate tags for all of them relative to a certain directory. To do this, go to the root directory of the source code and execute:

```
$ ctags -R .
```

Check whether the tags file has been generated. You also can open and read the tags file in Vim.

Now, let us move on to navigating the source code using tags. Navigating the source code using ctags is one of the most fascinating tools that a programmer has. You can read the code so nicely and quickly that you wonder how it would have been without ctags.

Once the tags file has been generated, open the file in Vim as normal, except that if the file is deep inside, open it from the root directory. For instance, your source code is organized like this:

```
common
  |
  ----> gui --> wxpython
  |       |
  |       ------>Tk
  |
  ----> backend --> networking
include
user
```

If you want to edit tcp.c under the common/backend/networking directory, you should open it like this:

```
$ vim common/backend/networking/tcp.c
```

instead of like this:

```
$ cd common/backend/networking
```

and:

```
$ vim tcp.c
```

The tags file is situated in the directory above common, and Vim automatically knows the location of the tags file this way.

Alternatively, you can open the file using the second method mentioned above and execute this from inside of Vim:

```
:se tags=../../../tags
```

The first method is easier for navigation. Once you open the file, you can jump from one function definition to another easily by using the key combination Ctrl-].

If you want to go to the definition of anything, be it a function, macro or anything else, simply press Ctrl-] when the cursor is positioned on it. Thus, from invocation, we can move to the definition. It takes you there no matter which file contains it. Assuming that we call drawscreen() from tcp.c, it automatically takes you there, even if the file is contained under common/gui.

If you want to go back to what you were reading, press Ctrl-T, and you return to where you left. You can jump to another invocation from there by pressing Ctrl-] again. You can continue this process ad infinitum, and you can keep coming back by pressing Ctrl-T.

Another way to find a function definition if you know only a part of the name is:

```
:ta /function
```

This command takes you to the first match if there are multiple matches. You can go to the next match with `:tn`.

If there are multiple definitions and you want to choose among them, you can press G Ctrl-] or type `:tselect <tagname>`. This way you can modify the source code by navigating with tags without even knowing which file contains what.

Vim with cscope

cscope is another powerful source code navigation tool with which we can perform a variety of searches. Here is a sample output of the cscope menu:

```
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Now, Vim has integrated cscope into its repertoire, making it convenient for programmers to use the same features in cscope from the cool comfort of Vim. All you have to do is establish a cscope connection by issuing `:cs add cscope.out`.

As we discussed before with ctags, cscope generates an index called cscope.out that can be generated by using the shell command:

```
$ cscope -Rbq
```

This generates the file cscope.out. It is to be executed from the source code root directory à la ctags. You then open the file as before, relative to the source code root directory, and make a cscope connection with the command `:cs add cscope.out`. You can verify existing cscope connections by typing `:cs show`.

What you can search for from inside of Vim can be seen using `:cs<CR>`. For instance, to go to a particular file, or a header of a source file, simply type `:cs f f stdio.h` for opening stdio.h or `:cs f f foo.c`.

For searching for functions called by a function foo.c, type `:cs f d foo.c`. This lists out the functions called by foo.c. For functions calling foo.c, type `:cs f c foo.c`.

To search for an egrep pattern, type `:cs f e varName` and so on. For a list of the available options, type `:cs`. It displays a range of available options.

Now, if you have both ctags and cscope, you can type `:cstag /foo` to search for a function or enum or whatever that contains foo.

Vim and Syntax Highlighting

If there is one feature in Vim for which it wins hands-down compared to any other editor or IDE, it is full-featured syntax highlighting. The colors available in Vim make it a veritable delight to work with source code. It not only makes your life colorful, it also makes it easy to spot errors ahead of compilation. Common errors such as a mismatched),} or] in the code are easy to see. It also reminds you if you have left a string hanging without the closing " or '. It tells you the comment doesn't end with */, or that you are nesting comments. Syntax highlighting is smart when it comes to C syntax.

Typically, you wouldn't have to do anything to enable Vim's syntax highlighting; `:sy on` does the job in case your distribution doesn't enable it by default. As with other commands, you can add this to your ~/.vimrc file.

If colors still don't show up, something is wrong with your terminal. Fix it first. `:se filetype on` is another thing you can try in addition to `:sy enable`.

Let us assume that you have colors displayed correctly. Say you don't like a certain color, or because blue is not visible in dark backgrounds, you can't read C comments. To solve the second problem, a simple `:se background=dark` does the job. If you want to disable syntax highlighting for C comments, type `:highlight clear comment`.

To change colors, first use the `:syntax` command to display all the syntax items for the given buffer. Then, identify the syntax group you want to change. If you want strings displayed in a bright white color, which is easy to read against a black background, simply enter:

```
:highlight String ctermfg=white
```

or, for gvim users, type:

```
:highlight String guifg=white
```

You also can change the syntax color of any group. Typical syntax groups are Statement, Label, Conditional, Repeat, Todo and Special. You can change the attributes of highlighting as well, such as underline and bold. For instance, if you want to display NOTE, FIXME, TODO and XXX with underlining, you can use:

```
:highlight Todo cterm=underline
```

or you can both add bold and change the color:

```
:highlight Repeat ctermfg=yellow cterm=bold
```

You can create your own set of highlight commands and save it in your ~/.vimrc file so that every time you edit your source code, your favorite colors are displayed.

Vim and Variable Name Completion

In addition, Vim has a feature for variable name completion. While typing, simply press Ctrl-N or Ctrl-P in insert mode. Remember, this works only in insert mode. All other commands mentioned above work in command mode. You can cycle through possible completions by pressing Ctrl-N again.

This helps us avoid errors while typing, because structure members and function names often can be misspelled. This works best when Vim can use tags, so make sure a ctags file is in place.

Vim and Source Code Formatting

Vim understands C well enough to be able to indent code automatically. The default indentation style uses tabs, which may not be appropriate for some people. In order to remove tabs completely from the source, enter:

```
:set expandtab
:retab
```

which converts all tabs into spaces in such a way that the indentation is preserved. While typing C text, Vim automatically indents for you. This helps you figure out where you have your matching brace. You can match braces,),] and } with the % command in command mode. Simply take the cursor to a brace and press %, which takes you to the corresponding closing or opening brace. This works for comments as well as for #if, #ifdef and #endif.

After finishing typing the program, if you want to indent the whole file at one go, type `gg=G` in command mode. You then can remove tabs if you want by the above-mentioned method. `gq` is the command sequence for indenting comments. You can select a region and indent it too with the = operator.

If Vim's default tab indentation is painful to use, you can disable it by setting `:se nocindent`. Other indentation options are available. You can indent code between two braces and between certain line numbers. You can learn more by typing `:help indent.txt`.

Conclusion

Vim comes with rich help documentation. Type `:help` from inside of Vim to browse it. To go to a particular topic, press Ctrl-] on the turquoise-colored text. Vim's help documentation uses the navigation mechanism we saw using tags.

Resources for this article: www.linuxjournal.com/article/8455 [1].

Girish Venkatachalam loves to play with open-source operating systems, such as OpenBSD, FreeBSD and Debian GNU/Linux. He also likes to go cycling when not hacking. He can be contacted at girish1729@gmail.com [2].

Links

[1] http://interactive.linuxjournal.com//article/8455
[2] http://interactive.linuxjournal.com/
[3] https://www.ssc.com/lj/subs/NewUSA.html

Source URL: http://interactive.linuxjournal.com/article/8289