# Using (neo)vim for C++ development

**2021-01-08** cpp vim tools

## Background

I’ve been writing code in vim for the past several years, and in this post I’d like to share some tips on configuring a development environment. This post contains some notes on configuration that would have helped me when I first started using vim and working on my own config. I hope that as a result of reading this, you will be able to improve your workflow with some new features and make the development process easier and more convenient.

In this article, we will look at common tasks that occur when editing code and try to automate and improve them using vim. Each section contains a brief description of the problem, a proposed solution, overview of alternatives, a full code listing for the configuration, and a screenshot or animated screencast with a demonstration. At the end, additional links to useful plugins and resources will be provided.

Most of the tasks come down to installing and properly configuring one or more plugins. I assume that you are an experienced vim user and already use one of the plugin managers.

All of these tips are applicable in both vim and neovim. Also, despite the title, some of these tips can be applied not only to C++, but also to any other language.

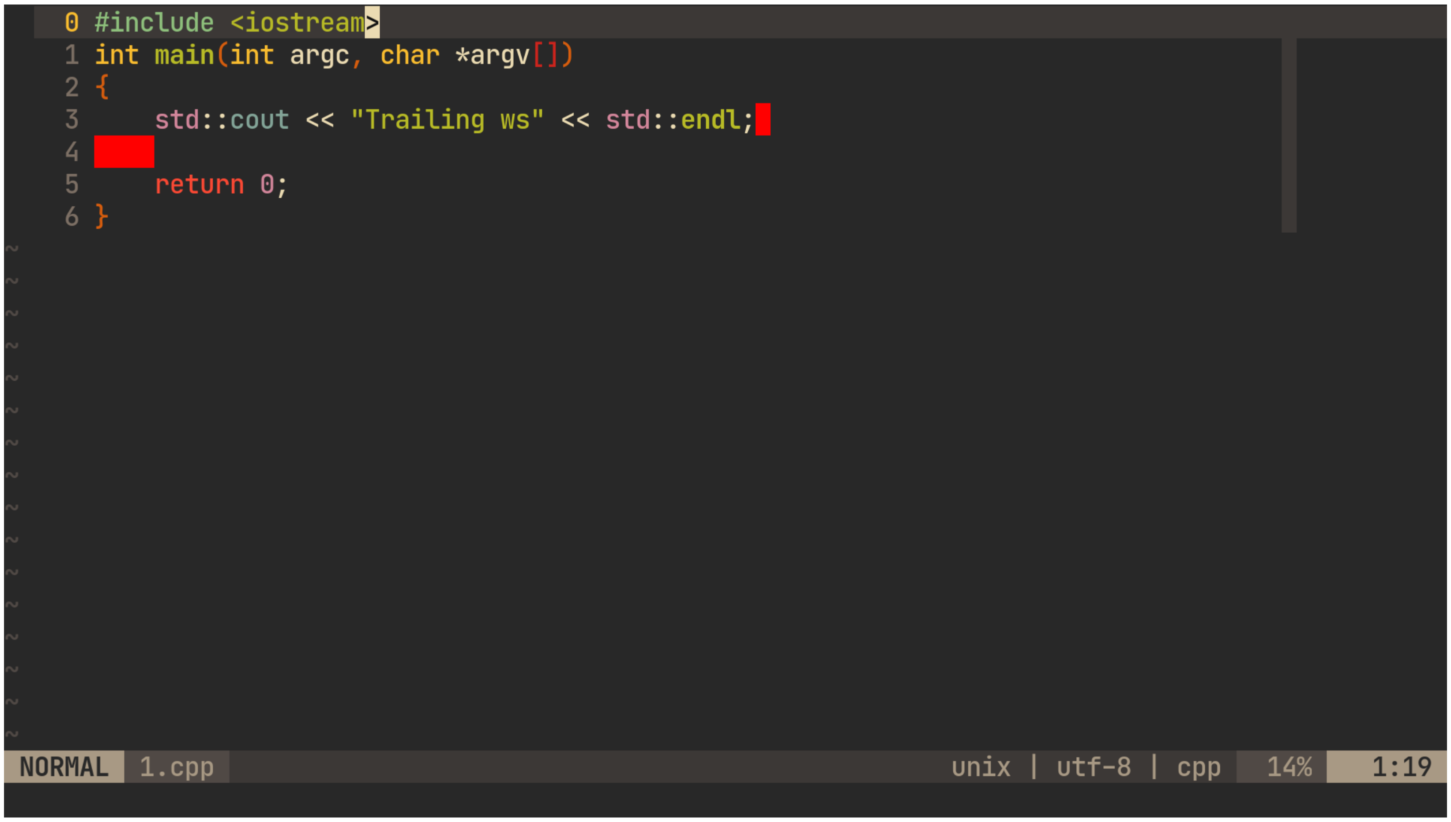
## Removing trailing whitespaces

Let’s start with a very simple problem. Some code conventions restrict the leaving of trailing whitespaces. For example, the Linux Kernel prohibits from doing this.

So we’d like to see when we accidentally leave a space at the end of a line. Let’s add the following lines in your vim configuration file:

```
highlight ExtraWhitespace ctermbg=red guibg=red
match ExtraWhitespace /\s\+$/
au BufWinEnter * match ExtraWhitespace /\s\+$/
au InsertEnter * match ExtraWhitespace /\s\+\%#\@<!$/
au InsertLeave * match ExtraWhitespace /\s\+$/
au BufWinLeave * call clearmatches()
```

As a result, extra whitespaces will be highlighted as follows:



```
#include <iostream>
int main(int argc, char *argv[])
{
    std::cout << "Trailing ws" << std::endl; 
    
    return 0;
}
```

```
NORMAL  1.cpp                                   unix | utf-8 | cpp  14%  1:19
```

We also want to remove extra spaces by the single key binding. Let’s define it:

```
" Remove all trailing whitespaces
nnoremap <silent> <leader>rs :let _s=@/ <Bar> :%s/\s\+$//e <Bar> :let @/=_s <Bar> :
```

So we can solve this simple problem.

Alternatives:

- You can show all whitespaces and line breaks in the source code.
- There is editorconfig plugin for vim that supports the trim_trailing_whitespace option. When it is set to true the plugin will remove any whitespace characters preceding newline characters.
- See clang-format section below.

## Accessing Standard Library documentation using cppman

Vim ships with a man page viewer that works out of the box on any modern *nix via :Man command. But the system man pages do not contain the definitions for C++. Let's fix it by installing Cppman.

Cppman is a set of C++ 98/11/14/17/20 manual pages for Linux, with source from cplusplus.com and cppreference.com. You can easily install using `pip` or the package manager for your distribution according to their Installation guide. Next, let's configure vim.

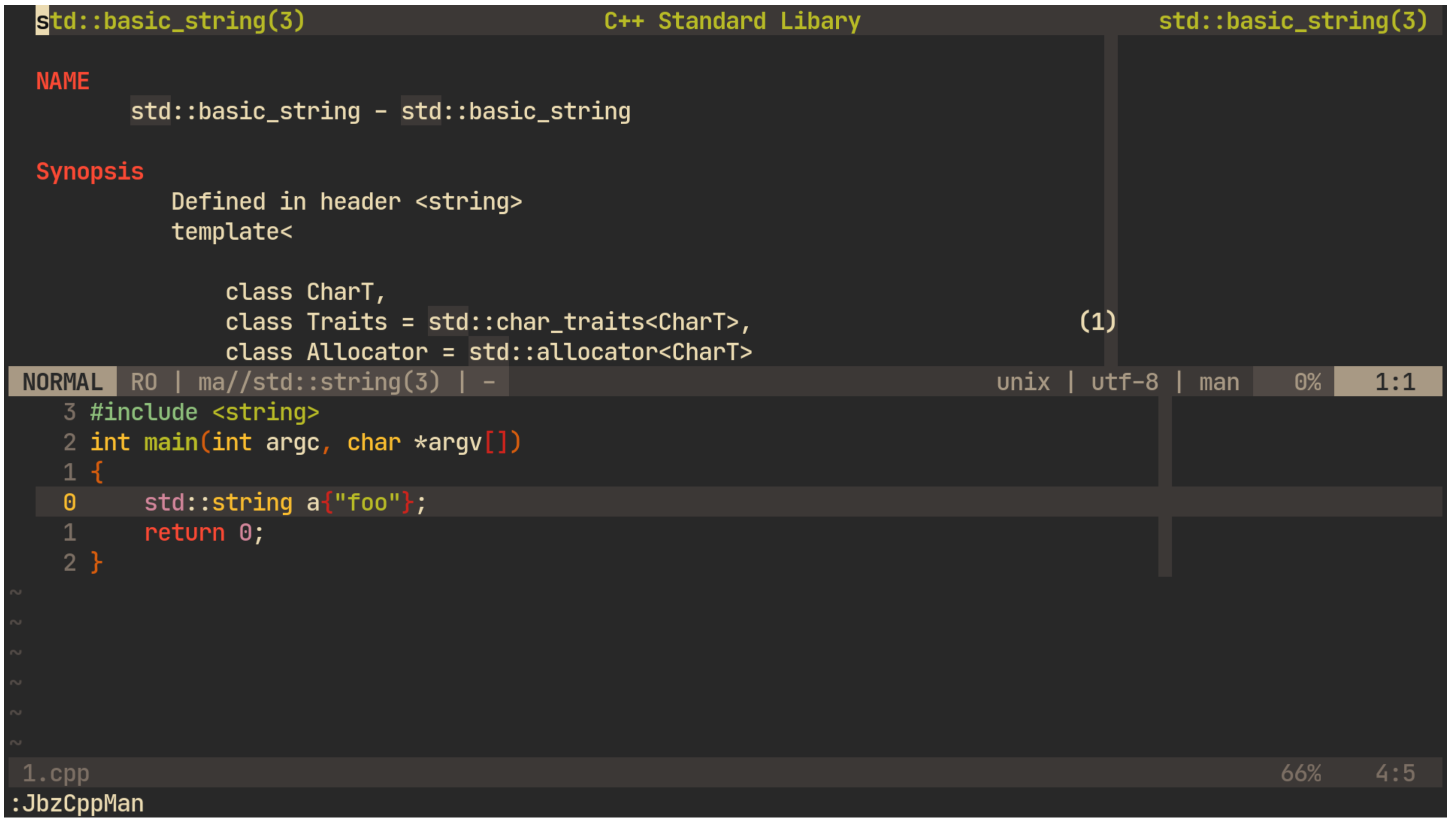
By default, when you press `K`, vim grabs the keyword under the cursor, separated by iskeyword symbols. There is one problem. The `:` symbol often found in C++ definitions is not included in the `iskeyword` array. So, for example, when you press `K` under the keyword `std::string`, vim will try to find the man page for `std`, instead of the full identifier. Let's fix it with this small function:

```
function! s:JbzCppMan()
    let old_isk = &iskeyword
    setl iskeyword+=:
    let str = expand("<cword>")
    let &l:iskeyword = old_isk
    execute 'Man ' . str
endfunction
command! JbzCppMan :call s:JbzCppMan()
```

We also need to remap the default `K` key binding to use our function in C++:

```
au FileType cpp nnoremap <buffer>K :JbzCppMan<CR>
```

Here is a result:



```
std::basic_string(3)          C++ Standard Libary          std::basic_string(3)

NAME
       std::basic_string - std::basic_string

Synopsis
          Defined in header <string>
          template<

              class CharT,
              class Traits = std::char_traits<CharT>,                     (1)
              class Allocator = std::allocator<CharT>
NORMAL  RO | ma//std::string(3) | -                 unix | utf-8 | man    0%   1:1
#include <string>
int main(int argc, char *argv[])
{
    std::string a{"foo"};
    return 0;
}
~
~
~
~
~
~
1.cpp                                                            66%   4:5
:JbzCppMan
```

## Browsing online documentation from vim

But what if you also want to quickly access online documentation for your favorite library or framework? Or may be search for unknown keywords on StackOverflow, Google or similar sites? And it is desirable to make a solution that can be easily extended to support more search engines.

open-browser.vim can help us. This is a plugin that allows you open URLs in your favorite browser using vim.

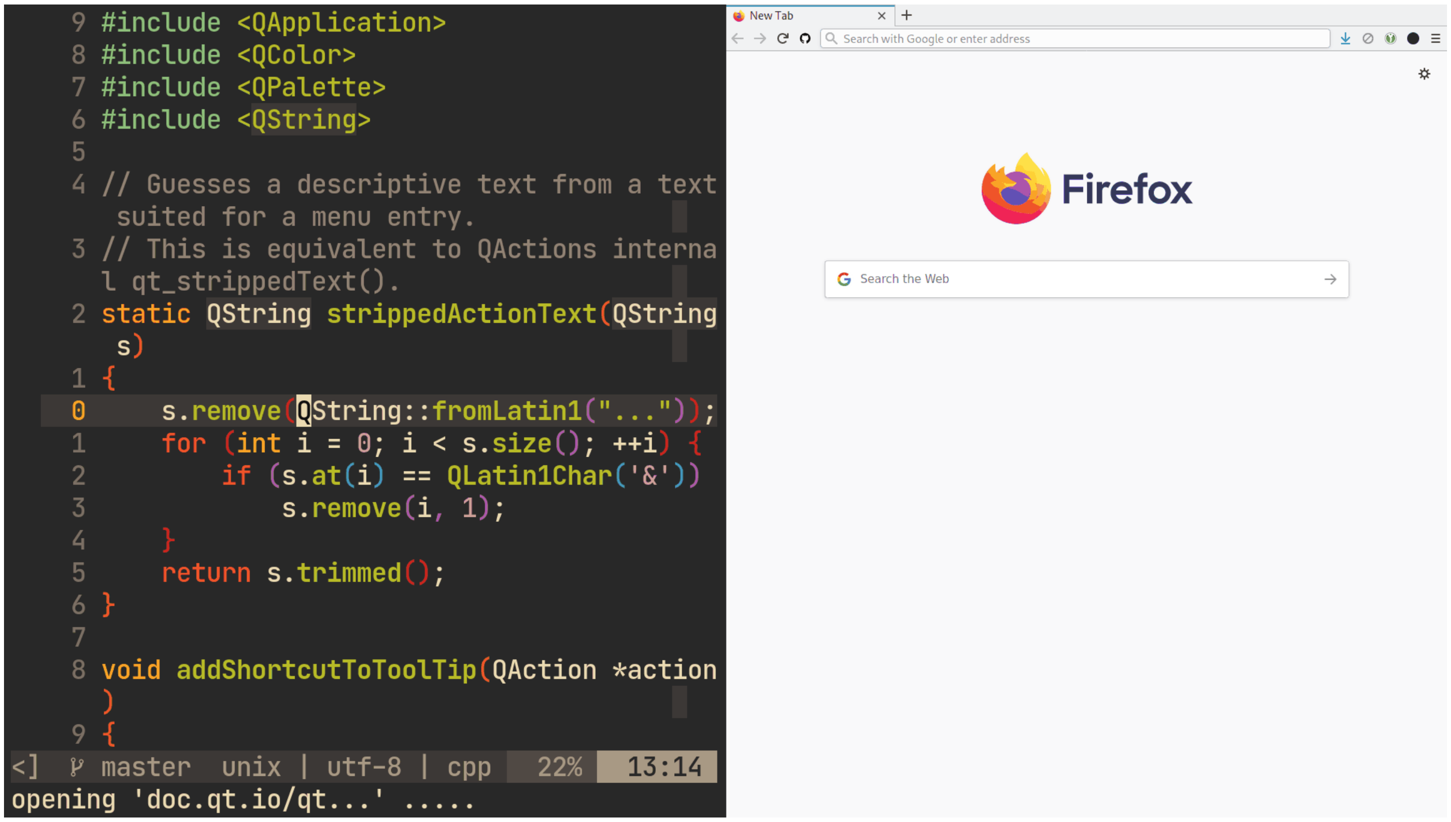
Install it using your package manager. Then let’s add a simple configuration to access cppreference and Qt documentation pages:

```
let g:openbrowser_search_engines = extend(
\ get(g:, 'openbrowser_search_engines', {}),
\ {
\    'cppreference': 'https://en.cppreference.com/mwiki/index.php?title=Special%3ASe
\    'qt': 'https://doc.qt.io/qt-5/search-results.html?q={query}',
\ },
\ 'keep'
\)
```

And add some convenient key bindings by this way:

```
nnoremap <silent> <leader>osx :call openbrowser#smart_search(expand('<cword>'), "cp
nnoremap <silent> <leader>osq :call openbrowser#smart_search(expand('<cword>'), "qt
```

As a result we can quickly browse online documentation for the word under the cursor:



```
#include <QApplication>
#include <QColor>
#include <QPalette>
#include <QString>

// Guesses a descriptive text from a text suited for a menu entry.
// This is equivalent to QActions internal qt_strippedText().
static QString strippedActionText(QString s)
{
    s.remove(QString::fromLatin1("..."));
    for (int i = 0; i < s.size(); ++i) {
        if (s.at(i) == QLatin1Char('&'))
            s.remove(i, 1);
    }
    return s.trimmed();
}

void addShortcutToToolTip(QAction *action)
{
<]  master  unix | utf-8 | cpp   22%   13:14
opening 'doc.qt.io/qt...' .....
```

Further use of this plugin is limited only by your imagination. For example, you can use it to search for similar C++ snippets in open source projects on GitHub:

```
'github-cpp': 'http://github.com/search?l=C%2B%2B&q=fork%3Afalse+language%3AC%2B%2B
```

Or integrate it with grep.app or Debian Code Search. Or add integration with your favorite online translation service, if you work with multiple languages. You get the idea. The main thing is to write the configuration correctly.

Alternatives:

- zeavim.vim is a plugin that implements the integration with Zeal – free and open source offline documentation browser. In fact, it solves the same problem. You may want to use it if you don't have an internet connection, or you need access to documentation for a specific version of the framework that isn't available online.

- devdocs.vim is a plugin for devdocs which also provides multiple API documentations.

## Switching between source and header file

Switching between source and header files is another common operation when working with C++.

After searching and trying many solutions, I settled on vim-fswitch plugin. Personally, I like this plugin for its simple configuration for various file types and the absence of third-party dependencies.

Install it with your favorite package manager. It will perfectly works out of the box, but sometimes you want to configure switch destination files like this:

```
au BufEnter *.h  let b:fswitchdst = "c,cpp,cc,m"
au BufEnter *.cc let b:fswitchdst = "h,hpp"
```

There is also one more nuance (thanks Elias Daler for sharing this!).

A lot of C++ projects follow this convention for splitting headers and sources:

```
include/<project-name>/<path>/some_header.h
src/<path>/some_source.cpp
```

To make FSwitch work with such cases (when switching from header to source), you need to add this to vimrc:

```
au BufEnter *.h let b:fswitchdst = 'c,cpp,m,cc' | let b:fswitchlocs = 'reg:|include
```

It also will be convenient to set up some key bindings:

```
nnoremap <silent> <A-o> :FSHere<cr>
" Extra hotkeys to open header/source in the split
nnoremap <silent> <localleader>oh :FSSplitLeft<cr>
nnoremap <silent> <localleader>oj :FSSplitBelow<cr>
nnoremap <silent> <localleader>ok :FSSplitAbove<cr>
nnoremap <silent> <localleader>ol :FSSplitRight<cr>
```

Alternatives:

- There are many other options described in this vimwiki article. You can try them or write your own solution.

## Using ctags

`ctags` is a convenient indexing tool that allows you to go to symbol definition from your project using `tags` database. This is an old and reliable tool, that is recommended to use for large code bases like Linux Kernel where other tools like LSP may get stuck.

The most actual and maintained implementation of ctags is universal-ctags. It is available in all popular distributions.

Vim has integrated ctags support. But here is one annoying thing. We need to generate `tags` database for each project using something like `ctags -R .`. Moreover, we need re-generate `tags` when editing the project.

This of course can be automated. Install vim-guttentags is a plugin that will asynchronously (re)generate tag files as you work. I also suggest adding a simple configuration to avoid indexing of some unwanted files:

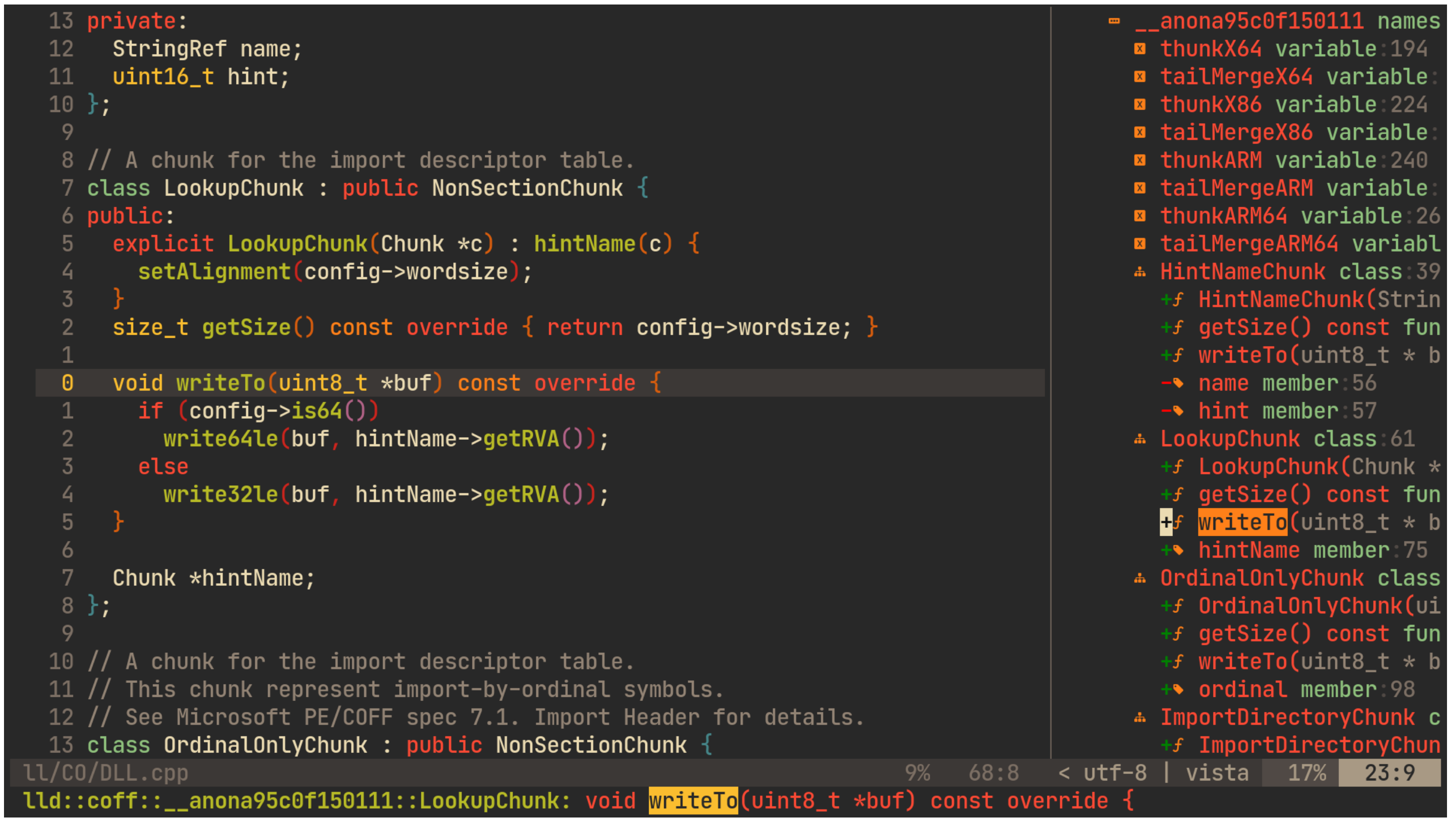
```
set tags=./tags;
let g:gutentags_ctags_exclude_wildignore = 1
let g:gutentags_ctags_exclude = [
  \'node_modules', '_build', 'build', 'CMakeFiles', '.mypy_cache', 'venv',
  \'*.md', '*.tex', '*.css', '*.html', '*.json', '*.xml', '*.xmls', '*.ui']
```

This makes your workflow simpler and more convenient. See also this detailed vimwiki article if you need more information about using ctags.

## Exploring source file structure with vista.vim

When you are working with unfamiliar source code, it may be convenient to browse the structure of the current source file. What classes, functions, macroses are defined here?

vista.vim is a plugin that helps us. It opens a split with the current file definitions. Here is what it looks like:



```
private:
  StringRef name;
  uint16_t hint;
};

// A chunk for the import descriptor table.
class LookupChunk : public NonSectionChunk {
public:
  explicit LookupChunk(Chunk *c) : hintName(c) {
    setAlignment(config->wordsize);
  }
  size_t getSize() const override { return config->wordsize; }

  void writeTo(uint8_t *buf) const override {
    if (config->is64())
      write64le(buf, hintName->getRVA());
    else
      write32le(buf, hintName->getRVA());
  }

  Chunk *hintName;
};

// A chunk for the import descriptor table.
// This chunk represent import-by-ordinal symbols.
// See Microsoft PE/COFF spec 7.1. Import Header for details.
class OrdinalOnlyChunk : public NonSectionChunk {
```

```
__anona95c0f150111 names
  thunkX64 variable:194
  tailMergeX64 variable:
  thunkX86 variable:224
  tailMergeX86 variable:
  thunkARM variable:240
  tailMergeARM variable:
  thunkARM64 variable:26
  tailMergeARM64 variabl
  HintNameChunk class:39
    HintNameChunk(Strin
    getSize() const fun
    writeTo(uint8_t * b
    name member:56
    hint member:57
  LookupChunk class:61
    LookupChunk(Chunk *
    getSize() const fun
    writeTo(uint8_t * b
    hintName member:75
  OrdinalOnlyChunk class
    OrdinalOnlyChunk(ui
    getSize() const fun
    writeTo(uint8_t * b
    ordinal member:98
  ImportDirectoryChunk c
    ImportDirectoryChun
```

```
ll/CO/DLL.cpp                              9%   68:8  < utf-8 | vista  17%  23:9
lld::coff::__anona95c0f150111::LookupChunk: void writeTo(uint8_t *buf) const override {
```

This plugin shows both LSP and ctags symbols. The LSP configuration is beyond the scope of this post, but it is a very useful feature that can give more accurate information.

To use it, install vista.vim with your favorite package manager and add a convenient key binding to toggle Vista split:

```
nnoremap <silent> <A-6> :Vista!!<CR>
```

Alternatives:

- tagbar is another plugin that provides a way to get the overview of the file structure. Unlike vista.vim, it doesn't support LSP symbols and doesn't provide useful utility functions for retrieving the information about the symbol under the cursor. But it's simpler and more reliable.

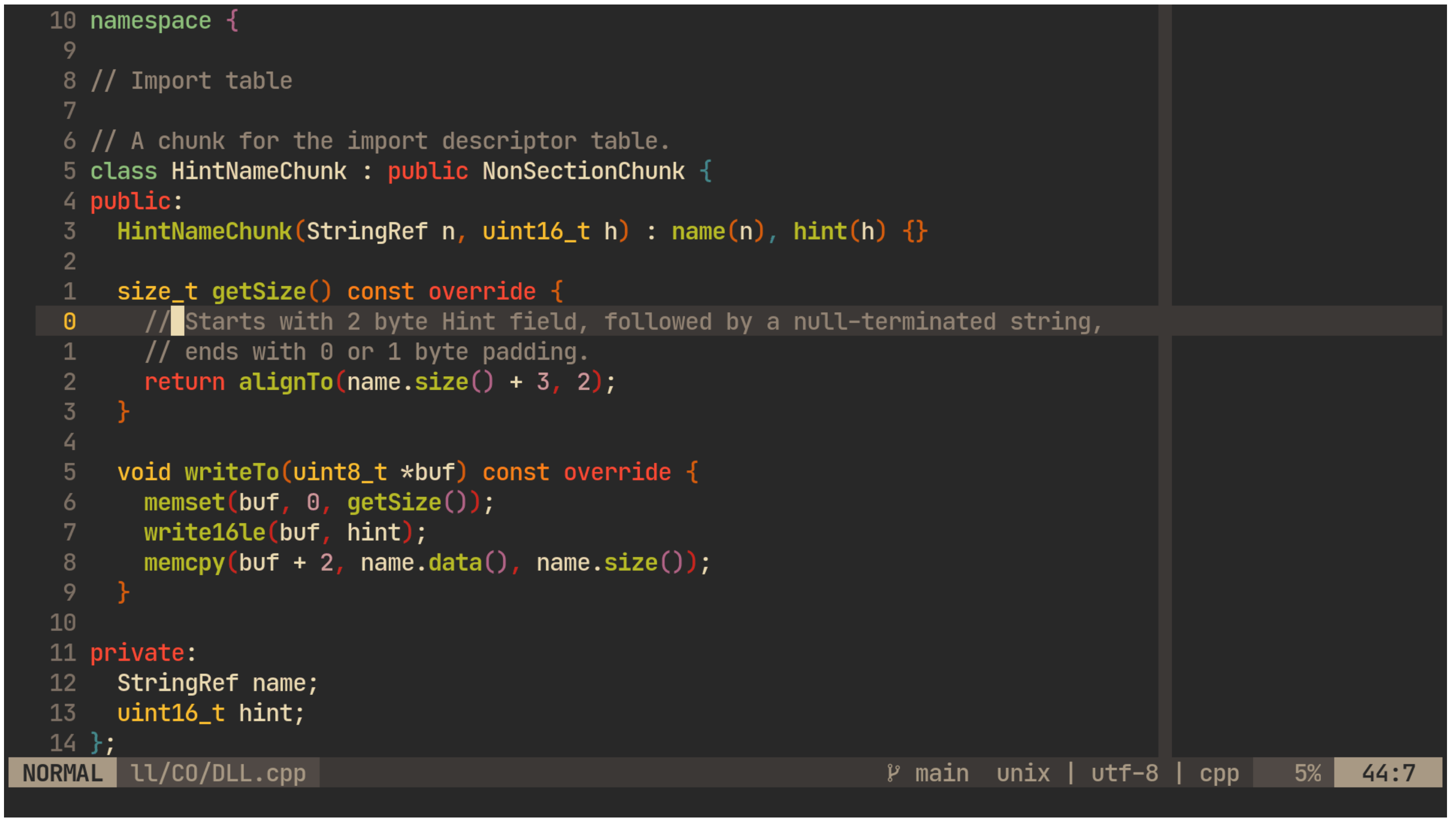
## Display the current function in the status line

Another feature of vista.vim is that it shares information of the symbol under the cursor. Using this we can display the current function name in the status line. This helps us navigate when moving through a large file.

Bellow is an example of configuration for the lightline, but you can easily adapt it to your status bar:

```
function! LightlineCurrentFunctionVista() abort
  let l:method = get(b:, 'vista_nearest_method_or_function', '')
  if l:method != ''
    let l:method = '[' . l:method . ']'
  endif
  return l:method
endfunction
au VimEnter * call vista#RunForNearestMethodOrFunction()
```

Here's what it looks like:



```
namespace {

// Import table

// A chunk for the import descriptor table.
class HintNameChunk : public NonSectionChunk {
public:
  HintNameChunk(StringRef n, uint16_t h) : name(n), hint(h) {}

  size_t getSize() const override {
    // Starts with 2 byte Hint field, followed by a null-terminated string,
    // ends with 0 or 1 byte padding.
    return alignTo(name.size() + 3, 2);
  }

  void writeTo(uint8_t *buf) const override {
    memset(buf, 0, getSize());
    write16le(buf, hint);
    memcpy(buf + 2, name.data(), name.size());
  }

private:
  StringRef name;
  uint16_t hint;
};
NORMAL  ll/CO/DLL.cpp                          main  unix | utf-8 | cpp   5%   44:7
```

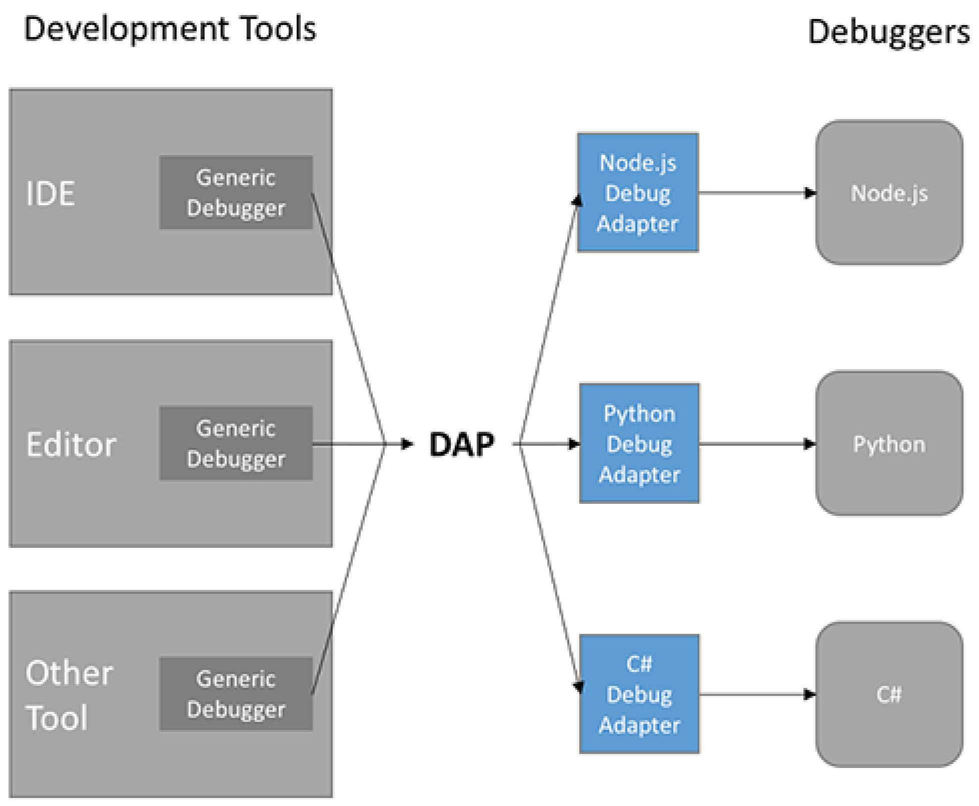
Is you need the complete lightline configuration, see [this snippet]() and take a look at lightline documentation.

Alternatives:

- There are many alternatives provided by popular LSP plugins. For example, [lsp-status.nvim]() can provide the same information for Neovim's built-in LSP server.

## vimspector: Interactive debugging inside vim

One of the interesting features supported by modern text editors and IDEs is the [Debug Adapter Protocol (DAP)](). The idea behind the Debug Adapter Protocol is to standardize the communication between the development tool and a concrete debugger or runtime:



It was originally developed to use with Visual Studio Code, but later it was moved to separate project. Currently, DAP support is implemented in all popular text editors, including vim.

The most popular DAP adapter for vim provided by awesome vimspector plugin.

The installation process is a bit tricky. We will need vscode-cpptools to make it works with C and C++ executables. Fortunately, vimspector ships with convenient script that automates our installation. The installation process is also well documented.

If you are using vim-plug plugin manager as well as I do, write the following in your configuration file:

```
Plug 'puremourning/vimspector', {
  \ 'do': 'python3 install_gadget.py --enable-vscode-cpptools'
  \ }
```

For other plugin managers, the installation will be actually the same. You just need to execute `install_gadget.py` script once after the installation as the documentation says.

Next let’s set up some key bindings:

```
command! -nargs=+ Vfb call vimspector#AddFunctionBreakpoint(<f-args>)
```

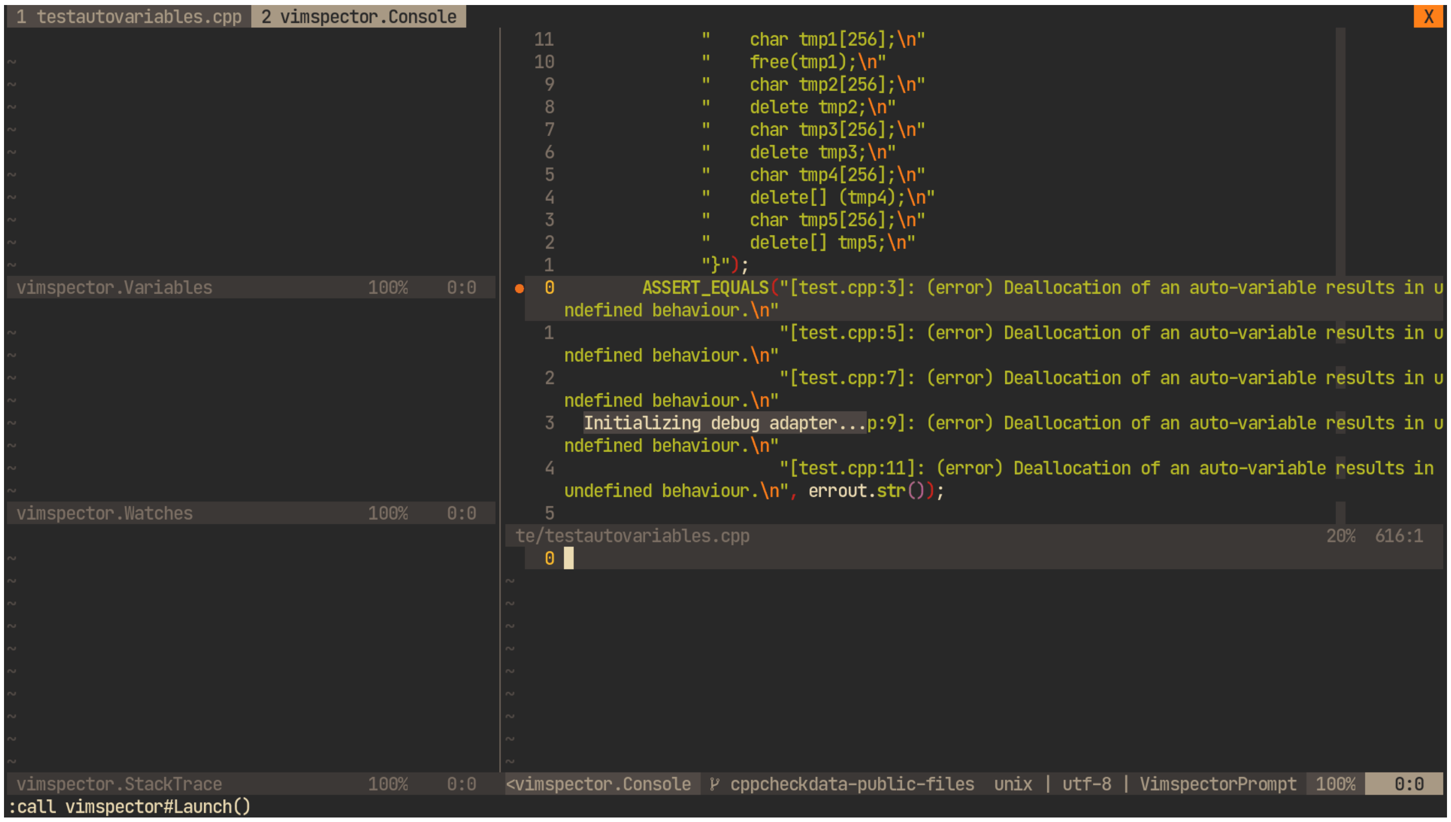
```
nnoremap <localleader>gd :call vimspector#Launch()<cr>
nnoremap <localleader>gc :call vimspector#Continue()<cr>
nnoremap <localleader>gs :call vimspector#Stop()<cr>
nnoremap <localleader>gR :call vimspector#Restart()<cr>
nnoremap <localleader>gp :call vimspector#Pause()<cr>
nnoremap <localleader>gb :call vimspector#ToggleBreakpoint()<cr>
nnoremap <localleader>gB :call vimspector#ToggleConditionalBreakpoint()<cr>
nnoremap <localleader>gn :call vimspector#StepOver()<cr>
nnoremap <localleader>gi :call vimspector#StepInto()<cr>
nnoremap <localleader>go :call vimspector#StepOut()<cr>
nnoremap <localleader>gr :call vimspector#RunToCursor()<cr>
```

After that, our setup is ready to go.

For each project we need to create vimspector.json which contains information about how to start a debugger session. This file is pretty simple, and you will usually just copy it between your projects with minimal changes. Here is an example content:

```
{
  "configurations": {
    "Launch": {
      "adapter": "vscode-cpptools",
      "configuration": {
        "request": "launch",
        "program": "${gadgetDir}/vscode-cpptools/debugAdapters/bin/OpenDebugAD7",
        "externalConsole": true
      }
    }
  }
}
```

Put this file in the project root, add breakpoints and launch the debugger session with defined key binding. Here is a small demo:



```
 1 testautovariables.cpp  2 vimspector.Console                                                          X
                                 "    char tmp1[256];\n"
                                 "    free(tmp1);\n"
                                 "    char tmp2[256];\n"
                                 "    delete tmp2;\n"
                                 "    char tmp3[256];\n"
                                 "    delete tmp3;\n"
                                 "    char tmp4[256];\n"
                                 "    delete[] (tmp4);\n"
                                 "    char tmp5[256];\n"
                                 "    delete[] tmp5;\n"
                                 "}");
 vimspector.Variables   100%  0:0    ASSERT_EQUALS("[test.cpp:3]: (error) Deallocation of an auto-variable results in u
                                ndefined behaviour.\n"
                                          "[test.cpp:5]: (error) Deallocation of an auto-variable results in u
                                ndefined behaviour.\n"
                                          "[test.cpp:7]: (error) Deallocation of an auto-variable results in u
                                ndefined behaviour.\n"
                                  Initializing debug adapter...p:9]: (error) Deallocation of an auto-variable results in u
                                ndefined behaviour.\n"
                                          "[test.cpp:11]: (error) Deallocation of an auto-variable results in
                                undefined behaviour.\n", errout.str());
 vimspector.Watches     100%  0:0
                               te/testautovariables.cpp                                              20%  616:1
 vimspector.StackTrace  100%  0:0  <vimspector.Console  cppcheckdata-public-files  unix | utf-8 | VimspectorPrompt  100%  0:0
:call vimspector#Launch()
```

This plugin supports all common debugger actions like step over/into/out, breakpoints, watchlists, etc. It can also be used for remote debugging, which is very useful in embedded area.

Alternatives:

- Neovim users can be also interested in nvim-dap – DAP client written in Lua.

## Snippets and print-statement-debugging

But sometimes you can’t properly configure your project to use the debugger. Often, this problem occurs in embedded area or when you work with large code base and you don’t have debugging information. In this case, you can use the rudimentary print-statement-debugging method. And the snippets will help you perfectly in this case.

Snippets are smart templates that will insert text for you and adapt it to their context. For those who have never used them, it will be easier to show a small demo:

Snippets are a powerful tool. If you never use it before, I strongly recommend the article [How I'm able to take notes in mathematics lectures using LaTeX and Vim]() by Gilles Castel. This is a comprehensive article about note-taking in Mathematics lectures in Vim with good introduction to snippets. It's just fascinating. Some of his advices can be applied to your development workflow.

But let's back to our use-case. We need to install two plugins: [ultisnips]() and [vim-snippets]().

Open C++ file in vim and execute `:UltiSnipsEdit` to open snippets file. Add the following lines to it:

```
snippet prd
std::cout << __PRETTY_FUNCTION__ << " " << $1 << std::endl; // prdbg
endsnippet
```

Save it and execute `:call UltiSnips#RefreshSnippets()` to apply changes.

Next define a helper function:

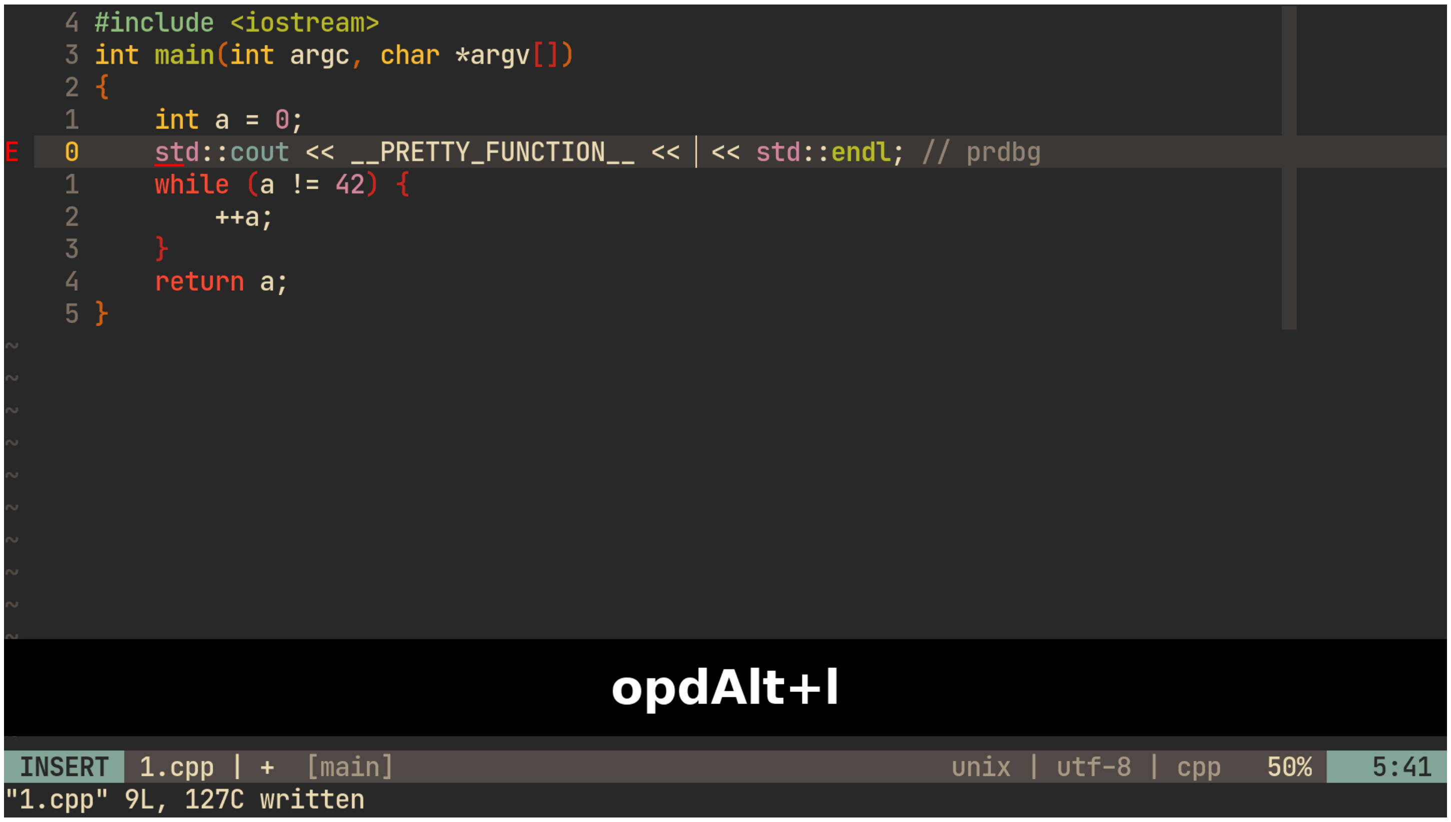
```
function! s:JbzRemoveDebugPrints()
  let save_cursor = getcurpos()
  :g/\/\/\ prdbg$/d
  call setpos('.', save_cursor)
endfunction
```

```
command! JbzRemoveDebugPrints call s:JbzRemoveDebugPrints()
```

This function will remove our print statements. We can also define a convenient key binding to call it:

```
au FileType c,cpp nnoremap <buffer><leader>rd :JbzRemoveDebugPrints<CR>
```

So, we can very quickly add our debug prints and delete them with a single command. Here is a demo:



```
#include <iostream>
int main(int argc, char *argv[])
{
    int a = 0;
    std::cout << __PRETTY_FUNCTION__ << | << std::endl; // prdbg
    while (a != 42) {
        ++a;
    }
    return a;
}
```

```
"1.cpp" 9L, 127C written
```

Improve your prints, add more snippets and use it. However, do not forget that the use of a debugger is preferable in most cases.

## Formatting source code with clang-format

clang-format is a tool to automatically format C/C++/Java/JavaScript/Objective-C/Protobuf/C# code, so that developers don’t need to worry about style issues during code reviews.

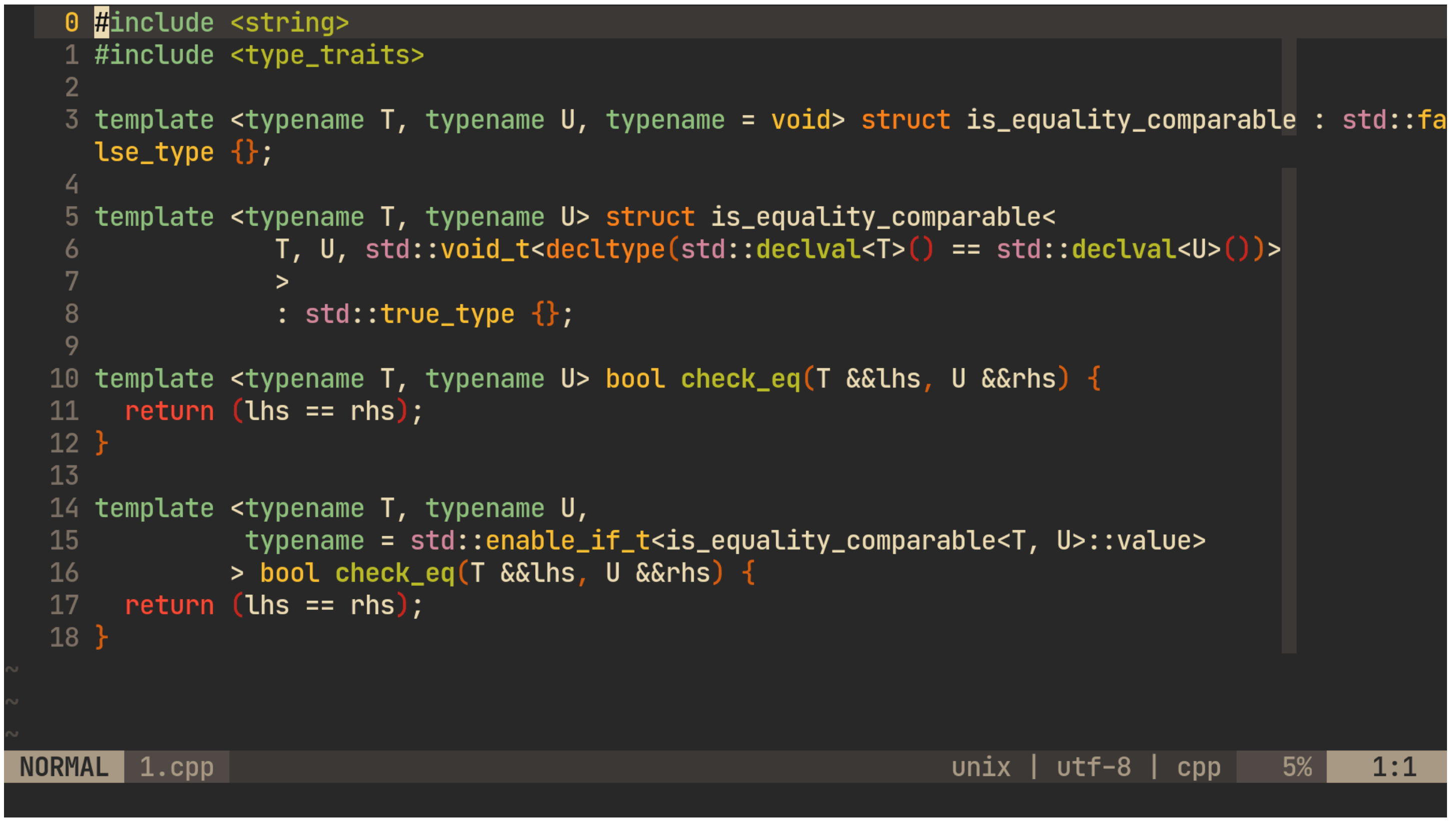
It can be simply integrated into vim. Install `clang-format` tool from your distribution repositories and create this function in the vim configuration:

```
function! s:JbzClangFormat(first, last)
  let l:winview = winsaveview()
  execute a:first . "," . a:last . "!clang-format"
  call winrestview(l:winview)
endfunction
command! -range=% JbzClangFormat call <sid>JbzClangFormat (<line1>, <line2>)
```

Add some key bindings:

```
" Autoformatting with clang-format
au FileType c,cpp nnoremap <buffer><leader>lf :<C-u>JbzClangFormat<CR>
au FileType c,cpp vnoremap <buffer><leader>lf :JbzClangFormat<CR>
```

And let's see how does it work:



```
#include <string>
#include <type_traits>

template <typename T, typename U, typename = void> struct is_equality_comparable : std::fa
lse_type {};

template <typename T, typename U> struct is_equality_comparable<
            T, U, std::void_t<decltype(std::declval<T>() == std::declval<U>())>
            >
            : std::true_type {};

template <typename T, typename U> bool check_eq(T &&lhs, U &&rhs) {
  return (lhs == rhs);
}

template <typename T, typename U,
          typename = std::enable_if_t<is_equality_comparable<T, U>::value>
         > bool check_eq(T &&lhs, U &&rhs) {
  return (lhs == rhs);
}
```

You can also select text regin in visual mode and call clang-format with the same key binding.

Alternatives:

- neoformat is a universal plugin that can run the arbitrary code formatters. Note that it also can be configured to use cmake_format CMakeLists.txt formatter, which can be handy in C++ projects.
- vim-clang-format – advanced plugin for running clang-format. See: What is the difference from clang-format.py? in their README.
- There is also clang-format.py script in LLVM distribution. You can use it according to the documentation.

## Enhancing syntax highlighting for modern C++

By default, vim has very poor syntax highlighting for C++. The problem is that C++ cannot be parsed without complete semantic analysis.

For example, if we have `f(x)` expression, depending on context `f` may be:

- a function (or function pointer or function reference)
- an instance of a class overloading operator()
- an object implicitly convertible to one of the above
- an overloaded function name (at any of multiple scopes)
- the name of one or more templates (at any of multiple scopes)
- the name of one or more template specializations
- … several of the above.

We will also need information about all header files and libraries included in the project.

Thus, we definitely need to get semantic information about the context.

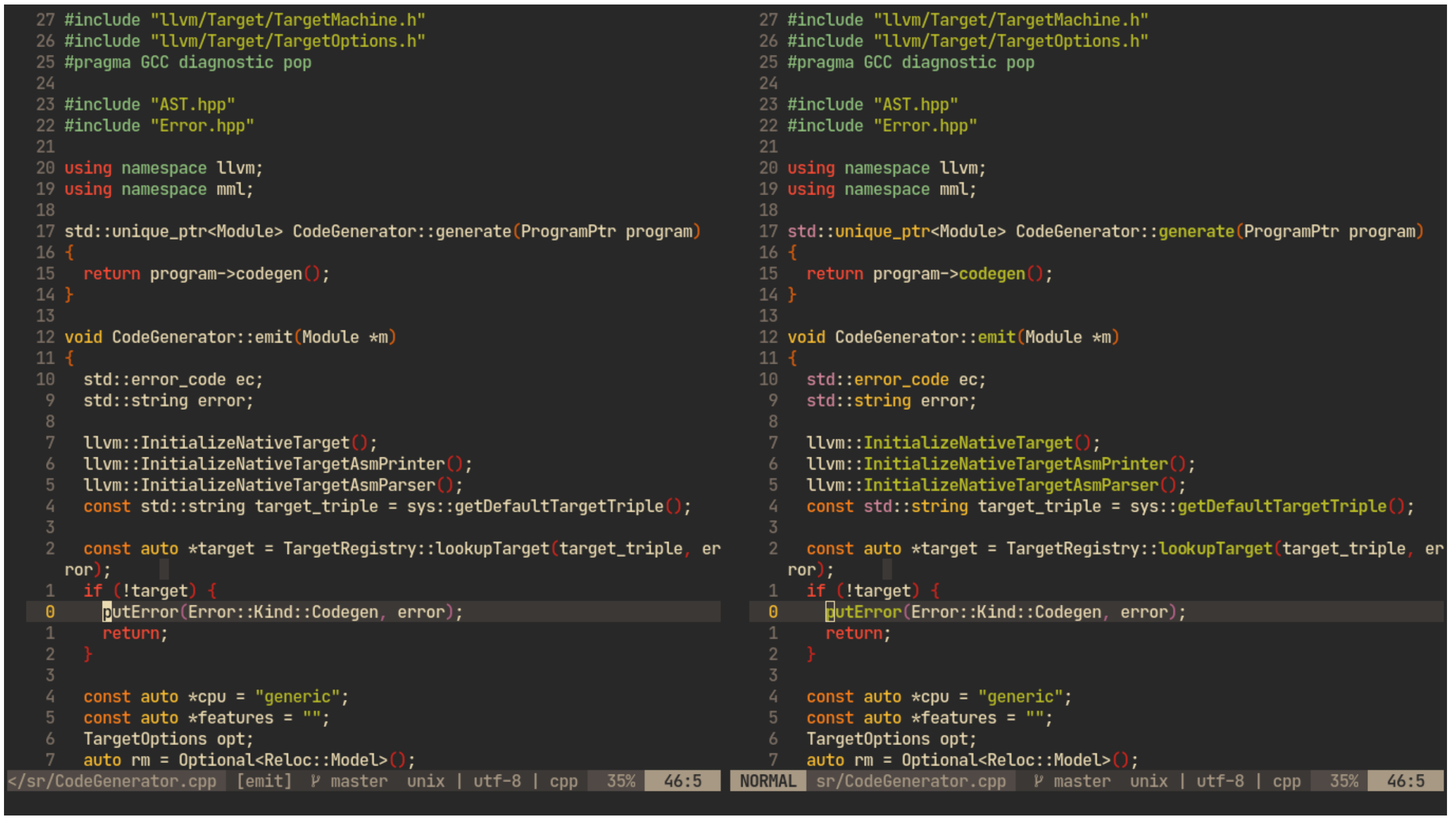
One option is to use semantic highlighting provided by the LSP server. Semantic highlighting is an extension of LSP added to the protocol several years ago.

The problem is that using semantic highlighting significantly complicates the configuration of the LSP server and client. You will also need a properly working LSP server for each project, which can be inconvenient if you don't want to run the build to get the `compile_commands.json` file. However, this solution gives you the most correct syntax highlighting. If you want to use it, consider vim-lsp-cxx-highlight plugin.

I suggest to take a look at a simpler solution. We can use an extended vim syntax file that contains keywords added in the recent standards and common C++ functions from the Standard Library.

To do this, install vim-cpp-modern using your package manager.

Result of plugin activation (on the right):



```
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetOptions.h"
#pragma GCC diagnostic pop

#include "AST.hpp"
#include "Error.hpp"

using namespace llvm;
using namespace mml;

std::unique_ptr<Module> CodeGenerator::generate(ProgramPtr program)
{
  return program->codegen();
}

void CodeGenerator::emit(Module *m)
{
  std::error_code ec;
  std::string error;

  llvm::InitializeNativeTarget();
  llvm::InitializeNativeTargetAsmPrinter();
  llvm::InitializeNativeTargetAsmParser();
  const std::string target_triple = sys::getDefaultTargetTriple();

  const auto *target = TargetRegistry::lookupTarget(target_triple, er
ror);
  if (!target) {
    putError(Error::Kind::Codegen, error);
    return;
  }

  const auto *cpu = "generic";
  const auto *features = "";
  TargetOptions opt;
  auto rm = Optional<Reloc::Model>();
</sr/CodeGenerator.cpp  [emit]   master  unix | utf-8 | cpp  35%  46:5
```

```
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetOptions.h"
#pragma GCC diagnostic pop

#include "AST.hpp"
#include "Error.hpp"

using namespace llvm;
using namespace mml;

std::unique_ptr<Module> CodeGenerator::generate(ProgramPtr program)
{
  return program->codegen();
}

void CodeGenerator::emit(Module *m)
{
  std::error_code ec;
  std::string error;

  llvm::InitializeNativeTarget();
  llvm::InitializeNativeTargetAsmPrinter();
  llvm::InitializeNativeTargetAsmParser();
  const std::string target_triple = sys::getDefaultTargetTriple();

  const auto *target = TargetRegistry::lookupTarget(target_triple, er
ror);
  if (!target) {
    putError(Error::Kind::Codegen, error);
    return;
  }

  const auto *cpu = "generic";
  const auto *features = "";
  TargetOptions opt;
  auto rm = Optional<Reloc::Model>();
NORMAL  sr/CodeGenerator.cpp   master  unix | utf-8 | cpp  35%  46:5
```

It should be noted that sometimes this plugin will highlight keywords inaccurately. For example, a variable named `string` used in your code will be colored as a keyword, since the plugin contains the corresponding rule for the `std::string` class. However, this is the simplest and most reliable solution.

Alternatives:

- vim-lsp-cxx-highlight mentioned above
- Neovim nightly has integration with tree-sitter – an incremental parsing system for programming tools. You can try using it and tell us in the comments how it works in comparison with LSP semantic highlighting and vim syntax files based on the regular expressions.

## Ideas for further configuration

I hope you can get some snippets from my post and simplify your vim-based C++ development environment. Of course, I can't fit all information about vim into one article. Therefore, I tried to write about some useful features that will certainly be useful to most developers.

Some topics deserve a separate writings. However, I consider it necessary to at least list them here:

- LSP plugins. I intentionally skiped this topic, because there are lots of alternatives: vim-lsp, LanguageClient-neovim, coc.nvim, Neovim's built-in LSP server, etc. Each of them contains its own configuration features and the additional plugins, therefore, it is impossible to fully cover this in one article.

- Autocompletion plugins, the choice of which usually depends on your LSP plugin.
- Take a look at handy plugins that improves the visual appearance: indentLine, vim-illuminate, rainbow.

If you have any additions, comments, and suggestions to improve this post, I would be very grateful if you write about this in the comments.

## Comments: 0

*You can leave a comment using this GitHub issue.*

### LATEST POSTS

- An overview of the Flow blockchain and the Float project
- Creating CI pipelines for C++/Qt applications with Buildbot
- Using (neo)vim for C++ development