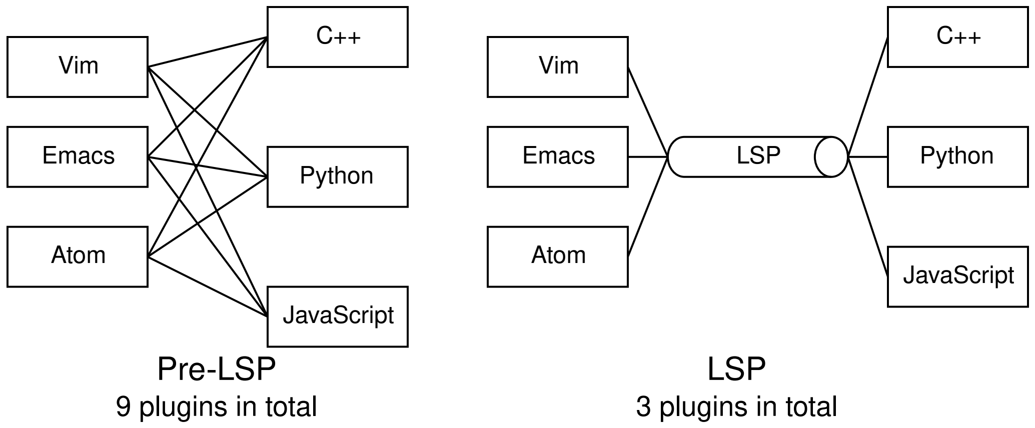
Configure coc.nvim for C/C++ Development

coc.nvim is yet another LSP plugin, which was first released in mid-2018. Because of its powerful functionality comparable to VSCode, coc.nvim soon becomes one of the most popular Vim plugins for 2018.

I have been using YouCompleteMe for many years. And I also once used Neocomplete and Deoplete. They were good for Python and JavaScript, but when it comes to C++, YouCompleteMe was just way better than any of them. However, the first time I tried coc.nvim, I fell in love with it and now it becomes one of the must-have plugins for my Vim configuration.

For those who are not familiar with LSP, I think it's worth a brief intro.

Language Server Protocol



Before LSP were born, the language plugins we developed for Vim couldn't be used for Emacs, which means we had to write a plugin for every pair of editors and languages. Although some like YouCompleteMe provides an editor-independent backend for different languages to some extent, it is not widely accepted by the community and hence the languages that it supports are limited.

To solve this problem, here comes LSP, the Language Server Protocol, a widely accepted protocol that an editor may use to communicate with a language plugin, which supports semantic completion, syntax checking, semantic highlighting, go-to-definition and countless more features. We only need to write one language server for a language, like Python, and without any effort, it can be used in all the editors that support LSP.

LSP is originally the protocol that VSCode uses to communicate with its extensions running in isolated processes. Later on, the VSCode team decided to open-source the protocol and allowed other editors to support it. Since then, many LSP plugins have been written for Vim. What makes coc.nvim outstanding is its powerful extensibility. It enriches Vim with the best part of VSCode, it's stable, and it's fast.

Setting up coc.nvim

Coc.nvim relies on Node.js to be installed in your system. In macOS, you need to run in your terminal:

```
brew install node
npm install -g yarn
```

Then add the following line to your Vim configuration. I use **vim-plugged** as my plugin manager. If you are using a different one, you should use the command that your plugin manager supports.

```
Plug 'neoclide/coc.nvim', {'do': { -> coc#util#install()}}
```

Then add these lines to your Vim configuration to set up coc.nvim:

```
" if hidden is not set, TextEdit might fail.
set hidden

" Some servers have issues with backup files, see #649
set nobackup
set nowritebackup

" Better display for messages
set cmdheight=2

" You will have bad experience for diagnostic messages when it's default 4000.
set updatetime=300

" don't give |ins-completion-menu| messages.
set shortmess+=c

" always show signcolumns
set signcolumn=yes

" Use tab for trigger completion with characters ahead and navigate.
" Use command ':verbose imap <tab>' to make sure tab is not mapped by other plugin.
inoremap <silent><expr> <TAB>
      \ pumvisible() ? "\<C-n>" :
      \ <SID>check_back_space() ? "\<TAB>" :
      \ coc#refresh()
inoremap <expr><S-TAB> pumvisible() ? "\<C-p>" : "\<C-h>"

function! s:check_back_space() abort
  let col = col('.') - 1
  return !col || getline('.')[col - 1]  =~# '\s'
endfunction

" Use <c-space> to trigger completion.
inoremap <silent><expr> <c-space> coc#refresh()

" Use <cr> to confirm completion, `<C-g>u` means break undo chain at current position.
" Coc only does snippet and additional edit on confirm.
inoremap <expr> <cr> pumvisible() ? "\<C-y>" : "\<C-g>u\<CR>"

" Use `[c` and `]c` to navigate diagnostics
nmap <silent> [c <Plug>(coc-diagnostic-prev)
nmap <silent> ]c <Plug>(coc-diagnostic-next)

" Remap keys for gotos
nmap <silent> gd <Plug>(coc-definition)
nmap <silent> gy <Plug>(coc-type-definition)
nmap <silent> gi <Plug>(coc-implementation)
nmap <silent> gr <Plug>(coc-references)

" Use K to show documentation in preview window
```

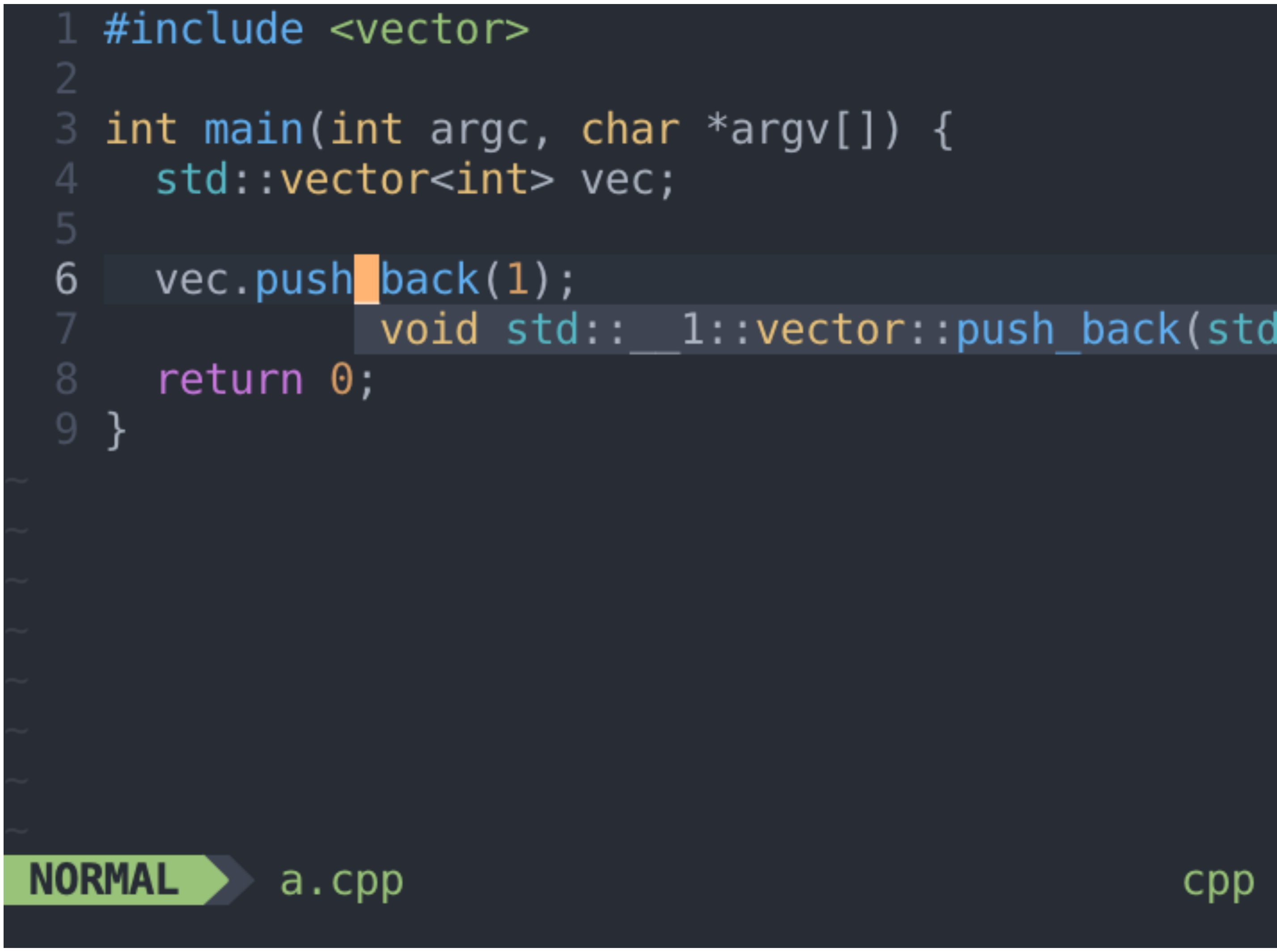
```
nnoremap <silent> K :call <SID>show_documentation()<CR>

function! s:show_documentation()
  if (index(['vim','help'], &filetype) >= 0)
    execute 'h '.expand('<cword>')
  else
    call CocAction('doHover')
  endif
endfunction

" Highlight symbol under cursor on CursorHold
autocmd CursorHold * silent call CocActionAsync('highlight')

" Remap for rename current word
nmap <leader>rn <Plug>(coc-rename)

" Remap for format selected region
xmap <leader>f  <Plug>(coc-format-selected)
nmap <leader>f  <Plug>(coc-format-selected)

augroup mygroup
  autocmd!
  " Setup formatexpr specified filetype(s).
  autocmd FileType typescript,json setl formatexpr=CocAction('formatSelected')
  " Update signature help on jump placeholder
  autocmd User CocJumpPlaceholder call CocActionAsync('showSignatureHelp')
augroup end

" Remap for do codeAction of selected region, ex: `<leader>aap` for current paragraph
xmap <leader>a  <Plug>(coc-codeaction-selected)
nmap <leader>a  <Plug>(coc-codeaction-selected)

" Remap for do codeAction of current line
nmap <leader>ac  <Plug>(coc-codeaction)
" Fix autofix problem of current line
nmap <leader>qf  <Plug>(coc-fix-current)

" Use <tab> for select selections ranges, needs server support, like: coc-tsserver, coc-python
nmap <silent> <TAB> <Plug>(coc-range-select)
xmap <silent> <TAB> <Plug>(coc-range-select)
xmap <silent> <S-TAB> <Plug>(coc-range-select-backword)

" Use `:Format` to format current buffer
command! -nargs=0 Format :call CocAction('format')

" Use `:Fold` to fold current buffer
command! -nargs=? Fold :call     CocAction('fold', <f-args>)

" use `:OR` for organize import of current buffer
command! -nargs=0 OR   :call     CocAction('runCommand', 'editor.action.organizeImport')

" Add status line support, for integration with other plugin, checkout `:h coc-status`
set statusline^=%{coc#status()}%{get(b:,'coc_current_function','')}

" Using CocList
" Show all diagnostics
nnoremap <silent> <space>a  :<C-u>CocList diagnostics<cr>
" Manage extensions
nnoremap <silent> <space>e  :<C-u>CocList extensions<cr>
" Show commands
nnoremap <silent> <space>c  :<C-u>CocList commands<cr>
" Find symbol of current document
nnoremap <silent> <space>o  :<C-u>CocList outline<cr>
" Search workspace symbols
nnoremap <silent> <space>s  :<C-u>CocList -I symbols<cr>
" Do default action for next item.
nnoremap <silent> <space>j  :<C-u>CocNext<CR>
" Do default action for previous item.
nnoremap <silent> <space>k  :<C-u>CocPrev<CR>
" Resume latest coc list
nnoremap <silent> <space>p  :<C-u>CocListResume<CR>
```

If you use delimitMate, you need to be careful about the key binding on `<CR>`, because our key binding will prohibit delimitMate from binding its own expansion function to `<CR>`, even if `delimitMate_expand_cr` is set to 1.

If you use **delimitMate**:

```
imap <expr> <cr> pumvisible() ? "\<C-y>" : "\<C-g>u\<Plug>delimitMateCR"
```

And if you are using Neovim, I would highly recommend you to use the newest nightly-built 4.0 version from Neovim's release page. By the time this post is written (when Neovim 0.3.7 was just released), the stable version has not supported the floating window yet.



```
#include <vector>

int main(int argc, char *argv[]) {
  std::vector<int> vec;

  vec.push_back(1);
                void std::__1::vector::push_back(std
  return 0;
}
```

Setting up ccls

I use ccls as my C/C++ language server.

In macOS, run the following command to install ccls from Homebrew.

```
brew update
brew install ccls
```

In Vim, run `:CocConfig` to open the configuration file of coc.nvim and add the following lines to it.

```
{
    "languageserver": {
        "ccls": {
            "command": "ccls",
            "filetypes": [
                "c",
                "cpp",
                "objc",
                "objcpp"
            ],
            "rootPatterns": [
                ".ccls",
                "compile_commands.json",
                ".vim/",
                ".git/",
                ".hg/"
            ],
            "initializationOptions": {
                "cache": {
                    "directory": "/tmp/ccls"
                }
            }
        }
    }
}
```

Setting up your C/C++ project

There are two ways to tell ccls your compile options.

1. generate `compile_commands.json` and put it to your project root;
2. place `.ccls` to your project root. It is a text file, in which each line is a command line argument passed to the compiler.

Although the first approach seems to be the most precise, but currently there are no perfect tools on macOS to generate it.

Most tools use one of the following two methods to generate `compile_commands.json`:

1. Intercepting the system calls and extracting the arguments passed to the compiler by dynamic library injection (e.g. Bear, scan-build):
   - pros: works for hard-coded compiler path;
   - cons: macOS prohibits dynamic library injection for security reasons if the the program to be injected is system software (e.g. clang from Xcode).
2. Using a compiler wrapper (e.g. scan-build):
   - pros: doesn't violate security policies;
   - cons: the compiler path must not be hard-coded.

Generally speaking, none of the tools solves all these problems and provides a perfect solution. Thus, I prefer putting `.ccls` in the project root.

Here is an example of `.ccls` (each command line argument occupies a line):

```
-I
../include
-I
../vendor/include
-std=c++14
-stdlib=libc++
-fPIC
```

If you are using macOS, then chances are ccls cannot find system headers and as a result reports a bunch of errors.

This is because new macOS systems moves system headers into the macOS SDK directory and no longer places them in `/usr/include`. And the reason why ccls can find the system headers previously is that `/usr/include` is hard-coded into ccls during compilation. But now, since the macOS SDK path is not hard-coded, it cannot find them any more.

I personally solved the issue by manually adding the path of the system headers to **.ccls**. Here is how to get the path:

Run `g++ -E -x c++ - -v < /dev/null` in your terminal and you'll see a list of include paths that the compiler searches. They are between `#include <...> search starts here:` and `End of search list.`. Now put them into your `.ccls` file as `-isystem` options (unlike `-I`, the errors and warnings in the header files found in `-isystem` paths are ignored by the syntax checker).

After manually adding these system header paths, the `.ccls` file might look like this:

```
-isystem
/usr/local/include
-isystem
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1
-isystem
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang/10.0.1/incl
-isystem
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include
-isystem
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.14.sdk/usr/
```