Valloric / YouCompleteMe

A code-completion engine for Vim http://valloric.github.io/YouCompleteMe/

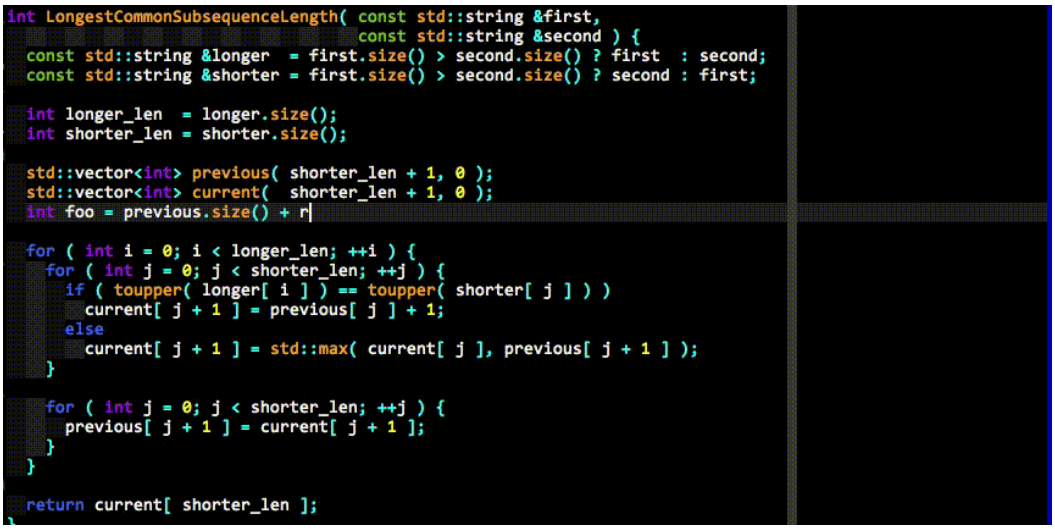
README.md

YouCompleteMe: a code-completion engine for Vim

Intro

YouCompleteMe is a fast, as-you-type, fuzzy-search code completion engine for Vim. It has several completion engines:

- an identifier-based engine that works with every programming language,
- a Clang-based engine that provides native semantic code completion for C/C++/Objective-C/Objective-C++ (from now on referred to as "the C-family languages"),
- a Jedi-based completion engine for Python,
- an OmniSharp-based completion engine for C#,
- a Gocode-based completion engine for Go,
- a TSServer-based completion engine for TypeScript,
- and an omnifunc-based completer that uses data from Vim's omnicomplete system to provide semantic completions for many other languages (Ruby, PHP etc.).



Here's an explanation of what happens in the short GIF demo above.

First, realize that **no keyboard shortcuts had to be pressed** to get the list of completion candidates at any point in the demo. The user just types and the suggestions pop up by themselves. If the user doesn't find the completion suggestions relevant and/or just wants to type, they can do so; the completion engine will not interfere.

When the user sees a useful completion string being offered, they press the TAB key to accept it. This inserts the completion string. Repeated presses of the TAB key cycle through the offered completions.

If the offered completions are not relevant enough, the user can continue typing to further filter out unwanted completions.

A critical thing to notice is that the completion **filtering is NOT based on the input being a string prefix of the completion** (but that works too). The input needs to be a *subsequence match* of a completion. This is a fancy way of saying that any input characters need to be present in a completion string in the order in which they appear in the input. So `abc` is a subsequence of `xaybgc`, but not of

`xbyxaxxc`. After the filter, a complicated sorting system ranks the completion strings so that the most relevant ones rise to the top of the menu (so you usually need to press TAB just once).

All of the above works with any programming language because of the identifier-based completion engine. It collects all of the identifiers in the current file and other files you visit (and your tags files) and searches them when you type (identifiers are put into per-filetype groups).

The demo also shows the semantic engine in use. When the user presses `.`, `->` or `::` while typing in insert mode (for C++; different triggers are used for other languages), the semantic engine is triggered (it can also be triggered with a keyboard shortcut; see the rest of the docs).

The last thing that you can see in the demo is YCM's diagnostic display features (the little red X that shows up in the left gutter; inspired by Syntastic) if you are editing a C-family file. As Clang compiles your file and detects warnings or errors, they will be presented in various ways. You don't need to save your file or press any keyboard shortcut to trigger this, it "just happens" in the background.

In essence, YCM obsoletes the following Vim plugins because it has all of their features plus extra:

- clang_complete
- AutoComplPop
- Supertab
- neocomplcache

YCM also provides semantic go-to-definition/declaration commands for C-family languages & Python. Expect more IDE features powered by the various YCM semantic engines in the future.

You'll also find that YCM has filepath completers (try typing `./` in a file) and a completer that integrates with UltiSnips.

Installation

Mac OS X super-quick installation

Please refer to the full Installation Guide below; the following commands are provided on a best-effort basis and may not work for you.

Install the latest version of MacVim. Yes, MacVim. And yes, the *latest*.

If you don't use the MacVim GUI, it is recommended to use the Vim binary that is inside the MacVim.app package (`MacVim.app/Contents/MacOS/Vim`). To ensure it works correctly copy the `mvim` script from the MacVim download to your local binary folder (for example `/usr/local/bin/mvim`) and then symlink it:

```
ln -s /usr/local/bin/mvim vim
```

Install YouCompleteMe with Vundle.

Remember: YCM is a plugin with a compiled component. If you **update** YCM using Vundle and the ycm_support_libs library APIs have changed (happens rarely), YCM will notify you to recompile it. You should then rerun the install process.

NOTE: If you want C-family completion, you MUST have the latest Xcode installed along with the latest Command Line Tools (they are installed when you start Xcode for the first time).

Install CMake. Preferably with Homebrew, but here's the stand-alone CMake installer.

If you have installed a Homebrew Python and/or Homebrew MacVim, see the *FAQ* for details.

Compiling YCM **with** semantic support for C-family languages:

```
cd ~/.vim/bundle/YouCompleteMe
./install.py --clang-completer
```

Compiling YCM **without** semantic support for C-family languages:

```
cd ~/.vim/bundle/YouCompleteMe
./install.py
```

If you want semantic C# support, you should add `--omnisharp-completer` to the install script as well. If you want Go support, you should add `--gocode-completer`. If you want semantic TypeScript support, install the TypeScript SDK with `npm install -g typescript`

That's it. You're done. Refer to the *User Guide* section on how to use YCM. Don't forget that if you want the C-family semantic completion engine to work, you will need to provide the compilation flags for your project to YCM. It's all in the User Guide.

YCM comes with sane defaults for its options, but you still may want to take a look at what's available for configuration. There are a few interesting options that are conservatively turned off by default that you may want to turn on.

Ubuntu Linux x64 super-quick installation

Please refer to the full Installation Guide below; the following commands are provided on a best-effort basis and may not work for you.

Make sure you have Vim 7.3.598 with python2 support. Ubuntu 14.04 and later have a Vim that's recent enough. You can see the version of Vim installed by running `vim --version`. If the version is too old, you may need to compile Vim from source (don't worry, it's easy).

Install YouCompleteMe with Vundle.

Remember: YCM is a plugin with a compiled component. If you **update** YCM using Vundle and the ycm_support_libs library APIs have changed (happens rarely), YCM will notify you to recompile it. You should then rerun the install process.

Install development tools and CMake: `sudo apt-get install build-essential cmake`

Make sure you have Python headers installed: `sudo apt-get install python-dev`.

Compiling YCM **with** semantic support for C-family languages:

```
cd ~/.vim/bundle/YouCompleteMe
./install.py --clang-completer
```

Compiling YCM **without** semantic support for C-family languages:

```
cd ~/.vim/bundle/YouCompleteMe
./install.py
```

If you want semantic C# support, you should add `--omnisharp-completer` to the install script as well. If you want Go support, you should add `--gocode-completer`. If you want semantic TypeScript support, install the TypeScript SDK with `npm install -g typescript`.

That's it. You're done. Refer to the *User Guide* section on how to use YCM. Don't forget that if you want the C-family semantic completion engine to work, you will need to provide the compilation flags for your project to YCM. It's all in the User Guide.

YCM comes with sane defaults for its options, but you still may want to take a look at what's available for configuration. There are a few interesting options that are conservatively turned off by default that you may want to turn on.

Windows Installation

YCM has **no official support for Windows**, but that doesn't mean you can't get it to work there. See the Windows Installation Guide wiki page. Feel free to add to it.

FreeBSD/OpenBSD Installation

Please refer to the full Installation Guide below; the following commands are provided on a best-effort basis and may not work for you. OpenBSD / FreeBSD are not officially supported platforms by YCM.

Make sure you have Vim 7.3.598 with python2 support.

OpenBSD 5.5 and later have a Vim that's recent enough. You can see the version of Vim installed by running `vim --version`.

FreeBSD 10.x comes with clang compiler but not the libraries needed to install.

```
pkg install llvm35 boost-all boost-python-libs clang35
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/llvm35/lib/
```

Install YouCompleteMe with Vundle.

Remember: YCM is a plugin with a compiled component. If you **update** YCM using Vundle and the ycm_support_libs library APIs have changed (happens rarely), YCM will notify you to recompile it. You should then rerun the install process.

Install dependencies and CMake: `sudo pkg_add llvm boost cmake`

Compiling YCM **with** semantic support for C-family languages:

```
cd ~/.vim/bundle/YouCompleteMe
./install.py --clang-completer --system-libclang --system-boost
```

Compiling YCM **without** semantic support for C-family languages:

```
cd ~/.vim/bundle/YouCompleteMe
./install.py --system-boost
```

If you want semantic C# support, you should add `--omnisharp-completer` to the install script as well. If you want Go support, you should add `--gocode-completer`.

That's it. You're done. Refer to the *User Guide* section on how to use YCM. Don't forget that if you want the C-family semantic completion engine to work, you will need to provide the compilation flags for your project to YCM. It's all in the User Guide.

YCM comes with sane defaults for its options, but you still may want to take a look at what's available for configuration. There are a few interesting options that are conservatively turned off by default that you may want to turn on.

Full Installation Guide

These are the steps necessary to get YCM working on a Unix OS like Linux or Mac OS X. My apologies to Windows users, but I don't have a guide for them. The code is platform agnostic, so if everything is configured correctly, YCM *should* work on Windows without issues (but as of writing, it's untested on that platform).

See the *FAQ* if you have any issues.

Remember: YCM is a plugin with a compiled component. If you **update** YCM using Vundle and the ycm_support_libs library APIs have changed (happens rarely), YCM will notify you to recompile it. You should then rerun the install process.

Please follow the instructions carefully. Read EVERY WORD.

1. **Ensure that your version of Vim is *at least* 7.3.598 *and* that it has support for python2 scripting.**

   Inside Vim, type `:version`. Look at the first two to three lines of output; it should say `Vi IMproved X.Y`, where X.Y is the major version of vim. If your version is greater than 7.3, then you're all set. If your version is 7.3 then look below that where it says, `Included patches: 1-Z`, where Z will be some number. That number needs to be 598 or higher.

   If your version of Vim is not recent enough, you may need to compile Vim from source (don't worry, it's easy).

   After you have made sure that you have Vim 7.3.598+, type the following in Vim: `:echo has('python')`. The output should be 1. If it's 0, then get a version of Vim with Python support.

2. **Install YCM** with Vundle (or Pathogen, but Vundle is a better idea). With Vundle, this would mean adding a `Plugin 'Valloric/YouCompleteMe'` line to your vimrc.

   If you don't install YCM with Vundle, make sure you have run `git submodule update --init --recursive` after checking out the YCM repository (Vundle will do this for you) to fetch YCM's dependencies.

3. [Complete this step ONLY if you care about semantic completion support for C-family languages. Otherwise it's not necessary.]

   Download the latest version of `libclang`. Clang is an open-source compiler that can compile C/C++/Objective-C/Objective-C++. The `libclang` library it provides is used to power the YCM semantic completion engine for those languages. YCM is designed to work with libclang version 3.6 or higher, but can in theory work with any 3.2+ version as well.

   You can use the system libclang *only if you are sure it is version 3.3 or higher*, otherwise don't. Even if it is, we recommend using the official binaries from llvm.org if at all possible. Make sure you download the correct archive file for your OS.

   We **STRONGLY recommend AGAINST use** of the system libclang instead of the upstream compiled binaries. Random things may break. Save yourself the hassle and use the upstream pre-built libclang.

4. **Compile the `ycm_support_libs` libraries** that YCM needs. These libs are the C++ engines that YCM uses to get fast completions.

   You will need to have `cmake` installed in order to generate the required makefiles. Linux users can install cmake with their package manager (`sudo apt-get install cmake` for Ubuntu) whereas other users can download and install cmake from its project site. Mac users can also get it through Homebrew with `brew install cmake`.

   You also need to make sure you have Python headers installed. On a Debian-like Linux distro, this would be `sudo apt-get install python-dev`. On Mac they should already be present.

   Here we'll assume you installed YCM with Vundle. That means that the top-level YCM directory is in `~/.vim/bundle/YouCompleteMe`.

   We'll create a new folder where build files will be placed. Run the following:

   ```
   cd ~
   mkdir ycm_build
   cd ycm_build
   ```

   Now we need to generate the makefiles. If you DON'T care about semantic support for C-family languages, run the following command in the `ycm_build` directory:

   ```
   cmake -G "Unix Makefiles" . ~/.vim/bundle/YouCompleteMe/third_party/ycmd/cpp
   ```

For those who want to use the system version of boost, you would pass `-DUSE_SYSTEM_BOOST=ON` to cmake. This may be necessary on some systems where the bundled version of boost doesn't compile out of the box.

NOTE: We **STRONGLY recommend AGAINST use** of the system boost instead of the bundled version of boost. Random things may break. Save yourself the hassle and use the bundled version of boost.

If you DO care about semantic support for C-family languages, then your `cmake` call will be a bit more complicated. We'll assume you downloaded a binary distribution of LLVM+Clang from llvm.org in step 3 and that you extracted the archive file to folder `~/ycm_temp/llvm_root_dir` (with `bin`, `lib`, `include` etc. folders right inside that folder).

NOTE: This *only* works with a *downloaded* LLVM binary package, not a custom-built LLVM! See docs below for `EXTERNAL_LIBCLANG_PATH` when using a custom LLVM build.

With that in mind, run the following command in the `ycm_build` directory:

```
cmake -G "Unix Makefiles" -DPATH_TO_LLVM_ROOT=~/ycm_temp/llvm_root_dir . ~/.vim/bundle/YouC
```

Now that makefiles have been generated, simply run:

```
make ycm_support_libs
```

For those who want to use the system version of libclang, you would pass `-DUSE_SYSTEM_LIBCLANG=ON` to cmake *instead of* the `-DPATH_TO_LLVM_ROOT=...` flag.

NOTE: We **STRONGLY recommend AGAINST use** of the system libclang instead of the upstream compiled binaries. Random things may break. Save yourself the hassle and use the upstream pre-built libclang.

You could also force the use of a custom libclang library with `-DEXTERNAL_LIBCLANG_PATH=/path/to/libclang.so` flag (the library would end with `.dylib` on a Mac). Again, this flag would be used *instead of* the other flags. **If you compiled LLVM from source, this is the flag you should be using.**

Running the `make` command will also place the `libclang.[so|dylib]` in the `YouCompleteMe/third_party/ycmd` folder for you if you compiled with clang support (it needs to be there for YCM to work).

That's it. You're done. Refer to the *User Guide* section on how to use YCM. Don't forget that if you want the C-family semantic completion engine to work, you will need to provide the compilation flags for your project to YCM. It's all in the User Guide.

YCM comes with sane defaults for its options, but you still may want to take a look at what's available for configuration. There are a few interesting options that are conservatively turned off by default that you may want to turn on.

Quick Feature Summary

General (all languages)

- Super-fast identifier completer including tags files and syntax elements
- Intelligent suggestion ranking and filtering
- File and path suggestions
- Suggestions from Vim's OmniFunc
- UltiSnips snippet suggestions

C-family languages (C, C++, Objective C, Objective C++)

- Semantic auto-completion
- Real-time diagnostic display
- Go to declaration/definition (`GoTo` , etc.)
- Semantic type information for identifiers (`GetType`)
- Automatically fix certain errors (`FixIt`)
- View documentation comments for identifiers (`GetDoc`)

C♯

- Semantic auto-completion
- Real-time diagnostic display
- Go to declaration/definition (`GoTo` , etc.)
- Semantic type information for identifiers (`GetType`)
- Automatically fix certain errors (`FixIt`)
- Management of OmniSharp server instance
- View documentation comments for identifiers (`GetDoc`)

Python 2

- Intelligent auto-completion
- Go to declaration/definition (`GoTo` , etc.)
- View documentation comments for identifiers (`GetDoc`)

Go

- Semantic auto-completion
- Management of `gocode` server instance

TypeScript

- Semantic auto-completion
- Go to definition (`GoToDefinition`)
- Semantic type information for identifiers (`GetType`)
- View documentation comments for identifiers (`GetDoc`)

User Guide

General Usage

- If the offered completions are too broad, keep typing characters; YCM will continue refining the offered completions based on your input.
- Filtering is "smart-case" sensitive; if you are typing only lowercase letters, then it's case-insensitive. If your input contains uppercase letters, then the uppercase letters in your query must match uppercase letters in the completion strings (the lowercase letters still match both). So, "foo" matches "Foo" and "foo", "Foo" matches "Foo" and "FOO" but not "foo".
- Use the TAB key to accept a completion and continue pressing TAB to cycle through the completions. Use Shift-TAB to cycle backwards. Note that if you're using console Vim (that is, not Gvim or MacVim) then it's likely that the Shift-TAB binding will not work because the console will not pass it to Vim. You can remap the keys; see the *Options* section below.

Knowing a little bit about how YCM works internally will prevent confusion. YCM has several completion engines: an identifier-based completer that collects all of the identifiers in the current file and other files you visit (and your tags files) and searches them when you type (identifiers are put into per-filetype groups).

There are also several semantic engines in YCM. There's a libclang-based completer that provides

semantic completion for C-family languages. There's a Jedi-based completer for semantic completion for Python. There's also an omnifunc-based completer that uses data from Vim's omnicomplete system to provide semantic completions when no native completer exists for that language in YCM.

There are also other completion engines, like the UltiSnips completer and the filepath completer.

YCM automatically detects which completion engine would be the best in any situation. On occasion, it queries several of them at once, merges the outputs and presents the results to you.

Client-server architecture

YCM has a client-server architecture; the Vim part of YCM is only a thin client that talks to the `ycmd` HTTP+JSON server that has the vast majority of YCM logic and functionality. The server is started and stopped automatically as you start and stop Vim.

Completion string ranking

The subsequence filter removes any completions that do not match the input, but then the sorting system kicks in. It's actually very complicated and uses lots of factors, but suffice it to say that "word boundary" (WB) subsequence character matches are "worth" more than non-WB matches. In effect, this means given an input of "gua", the completion "getUserAccount" would be ranked higher in the list than the "Fooguxa" completion (both of which are subsequence matches). A word-boundary character are all capital characters, characters preceded by an underscore and the first letter character in the completion string.

General Semantic Completion Engine Usage

- You can use Ctrl+Space to trigger the completion suggestions anywhere, even without a string prefix. This is useful to see which top-level functions are available for use.

C-family Semantic Completion Engine Usage

YCM looks for a `.ycm_extra_conf.py` file in the directory of the opened file or in any directory above it in the hierarchy (recursively); when the file is found, it is loaded (only once!) as a Python module. YCM calls a `FlagsForFile` method in that module which should provide it with the information necessary to compile the current file. You can also provide a path to a global `.ycm_extra_conf.py` file, which will be used as a fallback. To prevent the execution of malicious code from a file you didn't write YCM will ask you once per `.ycm_extra_conf.py` if it is safe to load. This can be disabled and you can white-/blacklist files. See the *Options* section for more details.

This system was designed this way so that the user can perform any arbitrary sequence of operations to produce a list of compilation flags YCM should hand to Clang.

See YCM's own `.ycm_extra_conf.py` for details on how this works. You should be able to use it *as a starting point*. **Don't** just copy/paste that file somewhere and expect things to magically work; **your project needs different flags**. Hint: just replace the strings in the `flags` variable with compilation flags necessary for your project. That should be enough for 99% of projects.

Yes, Clang's `CompilationDatabase` system is also supported. Again, see the above linked example file. You can get CMake to generate this file for you by adding `set( CMAKE_EXPORT_COMPILE_COMMANDS 1 )` to your project's `CMakeLists.txt` file (if using CMake). If you're not using CMake, you could use something like Bear to generate the `compile_commands.json` file.

Consider using YCM-Generator to generate the `ycm_extra_conf.py` file.

If Clang encounters errors when compiling the header files that your file includes, then it's probably going to take a long time to get completions. When the completion menu finally appears, it's going to have a large number of unrelated completion strings (type/function names that are not actually members). This is because Clang fails to build a precompiled preamble for your file if there are any errors in the included headers and that preamble is key to getting fast completions.

Call the `:YcmDiags` command to see if any errors or warnings were detected in your file.

Semantic completion for other languages

Python, C#, Go, and TypeScript are supported natively by YouCompleteMe using the Jedi, Omnisharp, Gocode, and TSServer engines, respectively. Check the installation section for instructions to enable these features if desired.

YCM will use your `omnifunc` (see `:h omnifunc` in Vim) as a source for semantic completions if it does not have a native semantic completion engine for your file's filetype. Vim comes with okayish omnifuncs for various languages like Ruby, PHP etc. It depends on the language.

You can get stellar omnifuncs for Java and Ruby with Eclim. Just make sure you have the *latest* Eclim installed and configured (this means Eclim `>= 2.2.*` and Eclipse `>= 4.2.*`).

After installing Eclim remember to create a new Eclipse project within your application by typing `:ProjectCreate <path-to-your-project> -n ruby` (or `-n java`) inside vim and don't forget to have `let g:EclimCompletionMethod = 'omnifunc'` in your vimrc. This will make YCM and Eclim play nice; YCM will use Eclim's omnifuncs as the data source for semantic completions and provide the auto-triggering and subsequence-based matching (and other YCM features) on top of it.

Writing New Semantic Completers

You have two options here: writing an `omnifunc` for Vim's omnicomplete system that YCM will then use through its omni-completer, or a custom completer for YCM using the Completer API.

Here are the differences between the two approaches:

- You have to use VimScript to write the omnifunc, but get to use Python to write for the Completer API; this by itself should make you want to use the API.
- The Completer API is a *much* more powerful way to integrate with YCM and it provides a wider set of features. For instance, you can make your Completer query your semantic back-end in an asynchronous fashion, thus not blocking Vim's GUI thread while your completion system is processing stuff. This is impossible with VimScript. All of YCM's completers use the Completer API.
- Performance with the Completer API is better since Python executes faster than VimScript.

If you want to use the `omnifunc` system, see the relevant Vim docs with `:h complete-functions`. For the Completer API, see the API docs.

If you want to upstream your completer into YCM's source, you should use the Completer API.

Diagnostic display

YCM will display diagnostic notifications for C-family and C# languages if you compiled YCM with Clang and Omnisharp support, respectively. Since YCM continuously recompiles your file as you type, you'll get notified of errors and warnings in your file as fast as possible.

Here are the various pieces of the diagnostic UI:

- Icons show up in the Vim gutter on lines that have a diagnostic.
- Regions of text related to diagnostics are highlighted (by default, a red wavy underline in `gvim` and a red background in `vim`).
- Moving the cursor to a line with a diagnostic echoes the diagnostic text.
- Vim's location list is automatically populated with diagnostic data (off by default, see options).

The new diagnostics (if any) will be displayed the next time you press any key on the keyboard. So if you stop typing and just wait for the new diagnostics to come in, that *will not work*. You need to press some key for the GUI to update.

Having to press a key to get the updates is unfortunate, but cannot be changed due to the way Vim internals operate; there is no way that a background task can update Vim's GUI after it has finished

running. You *have to* press a key. This will make YCM check for any pending diagnostics updates.

You *can* force a full, blocking compilation cycle with the `:YcmForceCompileAndDiagnostics` command (you may want to map that command to a key; try putting `nnoremap <F5> :YcmForceCompileAndDiagnostics<CR>` in your vimrc). Calling this command will force YCM to immediately recompile your file and display any new diagnostics it encounters. Do note that recompilation with this command may take a while and during this time the Vim GUI *will* be blocked.

YCM will display a short diagnostic message when you move your cursor to the line with the error. You can get a detailed diagnostic message with the `<leader>d` key mapping (can be changed in the options) YCM provides when your cursor is on the line with the diagnostic.

You can also see the full diagnostic message for all the diagnostics in the current file in Vim's `locationlist`, which can be opened with the `:lopen` and `:lclose` commands (make sure you have set `let g:ycm_always_populate_location_list = 1` in your vimrc). A good way to toggle the display of the `locationlist` with a single key mapping is provided by another (very small) Vim plugin called ListToggle (which also makes it possible to change the height of the `locationlist` window), also written by yours truly.

C# Diagnostic Support

Unlike the C-family diagnostic support, the C# diagnostic support is not a full compile run. Instead, it is a simple syntax check of the current file *only*. The `:YcmForceCompileAndDiagnostics` command also is only a simple syntax check, *not* a compile. This means that only syntax errors will be displayed, and not semantic errors. For example, omitting the semicolon at the end of statement will be displayed as a diagnostic error, but using a nonexistent class or variable will not be.

Diagnostic highlighting groups

You can change the styling for the highlighting groups YCM uses. For the signs in the Vim gutter, the relevant groups are:

- `YcmErrorSign`, which falls back to group `SyntasticErrorSign` and then `error` if they exist
- `YcmWarningSign`, which falls back to group `SyntasticWarningSign` and then `todo` if they exist

You can also style the line that has the warning/error with these groups:

- `YcmErrorLine`, which falls back to group `SyntasticErrorLine` if it exists
- `YcmWarningLine`, which falls back to group `SyntasticWarningLine` if it exists

Note that the line highlighting groups only work when gutter signs are turned on.

The syntax groups used to highlight regions of text with errors/warnings:

- `YcmErrorSection`, which falls back to group `SyntasticError` if it exists and then `SpellBad`
- `YcmWarningSection`, which falls back to group `SyntasticWarning` if it exists and then `SpellCap`

Here's how you'd change the style for a group:

```
highlight YcmErrorLine guibg=#3f0000
```

Commands

The `:YcmRestartServer` command

If the `ycmd` completion server suddenly stops for some reason, you can restart it with this command.

The `:YcmForceCompileAndDiagnostics` command

Calling this command will force YCM to immediately recompile your file and display any new diagnostics it encounters. Do note that recompilation with this command may take a while and during

this time the Vim GUI *will* be blocked.

You may want to map this command to a key; try putting `nnoremap <F5> :YcmForceCompileAndDiagnostics<CR>` in your vimrc.

The `:YcmDiags` command

Calling this command will fill Vim's `locationlist` with errors or warnings if any were detected in your file and then open it. If a given error or warning can be fixed by a call to `:YcmCompleter FixIt`, then `(FixIt available)` is appended to the error or warning text. See the `FixIt` completer subcommand for more information.

NOTE: The absense of `(FixIt available)` does not strictly imply a fix-it is not available as not all completers are able to provide this indication. For example, the c-sharp completer provides many fix-its but does not add this additional indication.

The `g:ycm_open_loclist_on_ycm_diags` option can be used to prevent the location list from opening, but still have it filled with new diagnostic data. See the *Options* section for details.

The `:YcmShowDetailedDiagnostic` command

This command shows the full diagnostic text when the user's cursor is on the line with the diagnostic.

The `:YcmDebugInfo` command

This will print out various debug information for the current file. Useful to see what compile commands will be used for the file if you're using the semantic completion engine.

The `:YcmCompleter` command

This command can be used to invoke completer-specific commands. If the first argument is of the form `ft=...` the completer for that file type will be used (for example `ft=cpp`), else the native completer of the current buffer will be used. Call `YcmCompleter` without further arguments for information about the commands you can call for the selected completer.

See the *YcmCompleter subcommands* section for more information on the available subcommands.

YcmCompleter subcommands

[See the docs for the `YcmCompleter` command before tackling this section.]

The invoked subcommand is automatically routed to the currently active semantic completer, so `:YcmCompleter GoToDefinition` will invoke the `GoToDefinition` subcommand on the Python semantic completer if the currently active file is a Python one and on the Clang completer if the currently active file is a C/C++/Objective-C one.

You may also want to map the subcommands to something less verbose; for instance, `nnoremap <leader>jd :YcmCompleter GoTo<CR>` maps the `<leader>jd` sequence to the longer subcommand invocation.

The various `GoTo*` subcommands add entries to Vim's `jumplist` so you can use `CTRL-O` to jump back to where you where before invoking the command (and `CTRL-I` to jump forward; see `:h jumplist` for details).

The `GoToDeclaration` subcommand

Looks up the symbol under the cursor and jumps to its declaration.

Supported in filetypes: `c, cpp, objc, objcpp, python, cs`

The `GoToDefinition` subcommand

Looks up the symbol under the cursor and jumps to its definition.

NOTE: For C-family languages **this only works in certain situations**, namely when the definition of the symbol is in the current translation unit. A translation unit consists of the file you are editing and all the files you are including with `#include` directives (directly or indirectly) in that file.

Supported in filetypes: `c, cpp, objc, objcpp, python, cs, typescript`

The `GoTo` subcommand

This command tries to perform the "most sensible" GoTo operation it can. Currently, this means that it tries to look up the symbol under the cursor and jumps to its definition if possible; if the definition is not accessible from the current translation unit, jumps to the symbol's declaration. For C#, implementations are also considered and preferred.

Supported in filetypes: `c, cpp, objc, objcpp, python, cs`

The `GoToImprecise` subcommand

WARNING: This command trades correctness for speed!

Same as the `GoTo` command except that it doesn't recompile the file with libclang before looking up nodes in the AST. This can be very useful when you're editing files that take long to compile but you know that you haven't made any changes since the last parse that would lead to incorrect jumps. When you're just browsing around your codebase, this command can spare you quite a bit of latency.

Supported in filetypes: `c, cpp, objc, objcpp`

The `ClearCompilationFlagCache` subcommand

YCM caches the flags it gets from the `FlagsForFile` function in your `ycm_extra_conf.py` file if you return them with the `do_cache` parameter set to `True`. The cache is in memory and is never invalidated (unless you restart Vim of course).

This command clears that cache entirely. YCM will then re-query your `FlagsForFile` function as needed in the future.

Supported in filetypes: `c, cpp, objc, objcpp`

The `GetType` subcommand

Echos the type of the variable or method under the cursor, and where it differs, the derived type.

For example:

```
std::string s;
```

Invoking this command on `s` returns `std::string => std::basic_string<char>`

NOTE: Due to limitations of `libclang`, invoking this command on the word `auto` typically returns `auto`. However, invoking it on a usage of the variable with inferred type returns the correct type, but typically it is repeated due to `libclang` returning that the types differ.

For example:

```
const char *s = "String";
auto x = &s; // invoking on x or auto returns "auto";
             // invoking on s returns "const char *"
std::cout << *x; // invoking on x returns "const char ** => const char **"
```

NOTE: Causes reparsing of the current translation unit.

Supported in filetypes: `c, cpp, objc, objcpp, typescript`

The `GetParent` subcommand

Echos the semantic parent of the point under the cursor.

The semantic parent is the item that semantically contains the given position.

For example:

```
class C {
    void f();
};

void C::f() {

}
```

In the out-of-line definition of `C::f`, the semantic parent is the class `C`, of which this function is a member.

In the example above, both declarations of `C::f` have `C` as their semantic context, while the lexical context of the first `C::f` is `C` and the lexical context of the second `C::f` is the translation unit.

For global declarations, the semantic parent is the translation unit.

NOTE: Causes reparsing of the current translation unit.

Supported in filetypes: `c, cpp, objc, objcpp`

The `FixIt` subcommand

Where available, attempts to make changes to the buffer to correct the diagnostic closest to the cursor position.

Completers which provide diagnostics may also provide trivial modifications to the source in order to correct the diagnostic. Examples include syntax errors such as missing trailing semi-colons, spurious characters, or other errors which the semantic engine can deterministically suggest corrections.

If no fix-it is available for the current line, or there is no diagnostic on the current line, this command has no effect on the current buffer. If any modifications are made, the number of changes made to the buffer is echo'd and the user may use the editor's undo command to revert.

When a diagnostic is available, and `g:ycm_echo_current_diagnostic` is set to 1, then the text `(FixIt)` is appended to the echo'd diagnostic when the completer is able to add this indication. The text `(FixIt available)` is also appended to the diagnostic text in the output of the `:YcmDiags` command for any diagnostics with available fix-its (where the completer can provide this indication).

NOTE: Causes re-parsing of the current translation unit.

NOTE: After applying a fix-it, the diagnostics UI is not immediately updated. This is due to a technical restriction in vim, and moving the cursor, or issuing the the `:YcmForceCompileAndDiagnostics` command will refresh the diagnostics. Repeated invocations of the `FixIt` command on a given line, however, *do* apply all diagnostics as expected without requiring refreshing of the diagnostics UI. This is particularly useful where there are multiple diagnostics on one line, or where after fixing one diagnostic, another fix-it is available.

Supported in filetypes: `c, cpp, objc, objcpp, cs`

The `GetDoc` subcommand

Displays the preview window populated with quick info about the identifier under the cursor. This includes, depending on the language, things like:

- The type or declaration of identifier
- Doxygen/javadoc comments
- Python docstrings
- etc.

Supported in filetypes: `c, cpp, objc, objcpp, cs, python, typescript`

The `StartServer` subcommand

Starts the semantic-engine-as-localhost-server for those semantic engines that work as separate servers that YCM talks to.

Supported in filetypes: `cs`

The `StopServer` subcommand

Stops the semantic-engine-as-localhost-server for those semantic engines that work as separate servers that YCM talks to.

Supported in filetypes: `cs`

The `RestartServer` subcommand

Restarts the semantic-engine-as-localhost-server for those semantic engines that work as separate servers that YCM talks to.

Supported in filetypes: `cs`

The `ReloadSolution` subcommand

Instruct the Omnisharp server to clear its cache and reload all files from disk. This is useful when files are added, removed, or renamed in the solution, files are changed outside of Vim, or whenever Omnisharp cache is out-of-sync.

Supported in filetypes: `cs`

The `GoToImplementation` subcommand

Looks up the symbol under the cursor and jumps to its implementation (i.e. non-interface). If there are multiple implementations, instead provides a list of implementations to choose from.

Supported in filetypes: `cs`

The `GoToImplementationElseDeclaration` subcommand

Looks up the symbol under the cursor and jumps to its implementation if one, else jump to its declaration. If there are multiple implementations, instead provides a list of implementations to choose from.

Supported in filetypes: `cs`

Options

All options have reasonable defaults so if the plug-in works after installation you don't need to change any options. These options can be configured in your vimrc script by including a line like this:

```
let g:ycm_min_num_of_chars_for_completion = 1
```

Note that after changing an option in your vimrc script you have to restart Vim for the changes to take effect.

The `g:ycm_min_num_of_chars_for_completion` option

This option controls the number of characters the user needs to type before identifier-based completion suggestions are triggered. For example, if the option is set to `2`, then when the user types a second alphanumeric character after a whitespace character, completion suggestions will be triggered. This option is NOT used for semantic completion.

Setting this option to a high number like `99` effectively turns off the identifier completion engine and just leaves the semantic engine.

Default: `2`

```
let g:ycm_min_num_of_chars_for_completion = 2
```

The `g:ycm_min_num_identifier_candidate_chars` option

This option controls the minimum number of characters that a completion candidate coming from the identifier completer must have to be shown in the popup menu.

A special value of `0` means there is no limit.

NOTE: This option only applies to the identifier completer; it has no effect on the various semantic completers.

Default: `0`

```
let g:ycm_min_num_identifier_candidate_chars = 0
```

The `g:ycm_auto_trigger` option

When set to `0`, this option turns off YCM's identifier completer (the as-you-type popup) *and* the semantic triggers (the popup you'd get after typing `.` or `->` in say C++). You can still force semantic completion with the `<C-Space>` shortcut.

If you want to just turn off the identifier completer but keep the semantic triggers, you should set `g:ycm_min_num_of_chars_for_completion` to a high number like `99`.

Default: `1`

```
let g:ycm_auto_trigger = 1
```

The `g:ycm_filetype_whitelist` option

This option controls for which Vim filetypes (see `:h filetype`) should YCM be turned on. The option value should be a Vim dictionary with keys being filetype strings (like `python`, `cpp` etc) and values being unimportant (the dictionary is used like a hash set, meaning that only the keys matter).

The `*` key is special and matches all filetypes. By default, the whitelist contains only this `*` key.

YCM also has a `g:ycm_filetype_blacklist` option that lists filetypes for which YCM shouldn't be turned on. YCM will work only in filetypes that both the whitelist and the blacklist allow (the blacklist "allows" a filetype by *not* having it as a key).

For example, let's assume you want YCM to work in files with the `cpp` filetype. The filetype should then be present in the whitelist either directly (`cpp` key in the whitelist) or indirectly through the special `*` key. It should *not* be present in the blacklist.

Filetypes that are blocked by the either of the lists will be completely ignored by YCM, meaning that neither the identifier-based completion engine nor the semantic engine will operate in them.

You can get the filetype of the current file in Vim with `:set ft?`.

Default: `{'*' : 1}`

```
let g:ycm_filetype_whitelist = { '*': 1 }
```

The `g:ycm_filetype_blacklist` option

This option controls for which Vim filetypes (see `:h filetype`) should YCM be turned off. The option value should be a Vim dictionary with keys being filetype strings (like `python`, `cpp` etc) and values being unimportant (the dictionary is used like a hash set, meaning that only the keys matter).

See the `g:ycm_filetype_whitelist` option for more details on how this works.

Default: `[see next line]`

```
let g:ycm_filetype_blacklist = {
      \ 'tagbar' : 1,
      \ 'qf' : 1,
      \ 'notes' : 1,
      \ 'markdown' : 1,
      \ 'unite' : 1,
      \ 'text' : 1,
      \ 'vimwiki' : 1,
      \ 'pandoc' : 1,
      \ 'infolog' : 1,
      \ 'mail' : 1
      \}
```

The `g:ycm_filetype_specific_completion_to_disable` option

This option controls for which Vim filetypes (see `:h filetype`) should the YCM semantic completion engine be turned off. The option value should be a Vim dictionary with keys being filetype strings (like `python`, `cpp` etc) and values being unimportant (the dictionary is used like a hash set, meaning that only the keys matter). The listed filetypes will be ignored by the YCM semantic completion engine, but the identifier-based completion engine will still trigger in files of those filetypes.

Note that even if semantic completion is not turned off for a specific filetype, you will not get semantic completion if the semantic engine does not support that filetype.

You can get the filetype of the current file in Vim with `:set ft?`.

Default: `[see next line]`

```
let g:ycm_filetype_specific_completion_to_disable = {
      \ 'gitcommit': 1
      \}
```

The `g:ycm_show_diagnostics_ui` option

When set, this option turns on YCM's diagnostic display features. See the *Diagnostic display* section in the *User Manual* for more details.

Specific parts of the diagnostics UI (like the gutter signs, text highlighting, diagnostic echo and auto location list population) can be individually turned on or off. See the other options below for details.

Note that YCM's diagnostics UI is only supported for C-family languages.

When set, this option also makes YCM remove all Syntastic checkers set for the `c`, `cpp`, `objc` and `objcpp` filetypes since this would conflict with YCM's own diagnostics UI.

If you're using YCM's identifier completer in C-family languages but cannot use the clang-based semantic completer for those languages *and* want to use the GCC Syntastic checkers, unset this option.

Default: `1`

```
let g:ycm_show_diagnostics_ui = 1
```

The `g:ycm_error_symbol` option

YCM will use the value of this option as the symbol for errors in the Vim gutter.

This option is part of the Syntastic compatibility layer; if the option is not set, YCM will fall back to the value of the `g:syntastic_error_symbol` option before using this option's default.

Default: `>>`

```
let g:ycm_error_symbol = '>>'
```

The `g:ycm_warning_symbol` option

YCM will use the value of this option as the symbol for warnings in the Vim gutter.

This option is part of the Syntastic compatibility layer; if the option is not set, YCM will fall back to the value of the `g:syntastic_warning_symbol` option before using this option's default.

Default: `>>`

```
let g:ycm_warning_symbol = '>>'
```

The `g:ycm_enable_diagnostic_signs` option

When this option is set, YCM will put icons in Vim's gutter on lines that have a diagnostic set. Turning this off will also turn off the `YcmErrorLine` and `YcmWarningLine` highlighting.

This option is part of the Syntastic compatibility layer; if the option is not set, YCM will fall back to the value of the `g:syntastic_enable_signs` option before using this option's default.

Default: `1`

```
let g:ycm_enable_diagnostic_signs = 1
```

The `g:ycm_enable_diagnostic_highlighting` option

When this option is set, YCM will highlight regions of text that are related to the diagnostic that is present on a line, if any.

This option is part of the Syntastic compatibility layer; if the option is not set, YCM will fall back to the value of the `g:syntastic_enable_highlighting` option before using this option's default.

Default: `1`

```
let g:ycm_enable_diagnostic_highlighting = 1
```

The `g:ycm_echo_current_diagnostic` option

When this option is set, YCM will echo the text of the diagnostic present on the current line when you move your cursor to that line. If a `FixIt` is available for the current diagnostic, then `(FixIt)` is appended.

This option is part of the Syntastic compatibility layer; if the option is not set, YCM will fall back to the value of the `g:syntastic_echo_current_error` option before using this option's default.

Default: `1`

```
let g:ycm_echo_current_diagnostic = 1
```

The `g:ycm_always_populate_location_list` option

When this option is set, YCM will populate the location list automatically every time it gets new diagnostic data. This option is off by default so as not to interfere with other data you might have placed in the location list.

See `:help location-list` in Vim to learn more about the location list.

This option is part of the Syntastic compatibility layer; if the option is not set, YCM will fall back to the value of the `g:syntastic_always_populate_loc_list` option before using this option's default.

Default: `0`

```
let g:ycm_always_populate_location_list = 0
```

The `g:ycm_open_loclist_on_ycm_diags` option

When this option is set, `:YcmDiags` will automatically open the location list after forcing a compilation and filling the list with diagnostic data.

See `:help location-list` in Vim to learn more about the location list.

Default: `1`

```
let g:ycm_open_loclist_on_ycm_diags = 1
```

The `g:ycm_allow_changing_updatetime` option

When this option is set to `1`, YCM will change the `updatetime` Vim option to `2000` (see `:h updatetime`). This may conflict with some other plugins you have (but it's unlikely). The `updatetime` option is the number of milliseconds that have to pass before Vim's `CursorHold` (see `:h CursorHold`) event fires. YCM runs the completion engines' "file comprehension" systems in the background on every such event; the identifier-based engine collects the identifiers whereas the semantic engine compiles the file to build an AST.

The Vim default of `4000` for `updatetime` is a bit long, so YCM reduces this. Set this option to `0` to force YCM to leave your `updatetime` setting alone.

Default: `1`

```
let g:ycm_allow_changing_updatetime = 1
```

The `g:ycm_complete_in_comments` option

When this option is set to `1`, YCM will show the completion menu even when typing inside comments.

Default: `0`

```
let g:ycm_complete_in_comments = 0
```

The `g:ycm_complete_in_strings` option

When this option is set to `1`, YCM will show the completion menu even when typing inside strings.

Note that this is turned on by default so that you can use the filename completion inside strings. This is very useful for instance in C-family files where typing `#include "` will trigger the start of filename completion. If you turn off this option, you will turn off filename completion in such situations as well.

Default: `1`

```
let g:ycm_complete_in_strings = 1
```

The `g:ycm_collect_identifiers_from_comments_and_strings` option

When this option is set to `1`, YCM's identifier completer will also collect identifiers from strings and comments. Otherwise, the text in comments and strings will be ignored.

Default: `0`

```
let g:ycm_collect_identifiers_from_comments_and_strings = 0
```

The `g:ycm_collect_identifiers_from_tags_files` option

When this option is set to `1`, YCM's identifier completer will also collect identifiers from tags files. The list of tags files to examine is retrieved from the `tagfiles()` Vim function which examines the `tags` Vim option. See `:h 'tags'` for details.

YCM will re-index your tags files if it detects that they have been modified.

The only supported tag format is the Exuberant Ctags format. The format from "plain" ctags is NOT supported. Ctags needs to be called with the `--fields=+l` option (that's a lowercase `L`, not a one) because YCM needs the `language:<lang>` field in the tags output.

See the *FAQ* for pointers if YCM does not appear to read your tag files.

This option is off by default because it makes Vim slower if your tags are on a network directory.

Default: `0`

```
let g:ycm_collect_identifiers_from_tags_files = 0
```

The `g:ycm_seed_identifiers_with_syntax` option

When this option is set to `1`, YCM's identifier completer will seed its identifier database with the keywords of the programming language you're writing.

Since the keywords are extracted from the Vim syntax file for the filetype, all keywords may not be collected, depending on how the syntax file was written. Usually at least 95% of the keywords are successfully extracted.

Default: `0`

```
let g:ycm_seed_identifiers_with_syntax = 0
```

The `g:ycm_extra_conf_vim_data` option

If you're using semantic completion for C-family files, this option might come handy; it's a way of sending data from Vim to your `FlagsForFile` function in your `.ycm_extra_conf.py` file.

This option is supposed to be a list of VimScript expression strings that are evaluated for every request to the `ycmd` server and then passed to your `FlagsForFile` function as a `client_data` keyword argument.

For instance, if you set this option to `['v:version']`, your `FlagsForFile` function will be called like this:

```
# The '704' value is of course contingent on Vim 7.4; in 7.3 it would be '703'
FlagsForFile(filename, client_data = {'v:version': 704})
```

So the `client_data` parameter is a dictionary mapping Vim expression strings to their values at the time of the request.

The correct way to define parameters for your `FlagsForFile` function:

```
def FlagsForFile(filename, **kwargs):
```

You can then get to `client_data` with `kwargs['client_data']`.

Default: `[]`

```
let g:ycm_extra_conf_vim_data = []
```

The `g:ycm_path_to_python_interpreter` option

YCM will by default search for an appropriate Python interpreter on your system. You can use this option to override that behavior and force the use of a specific interpreter of your choosing.

NOTE: This interpreter is only used for the `ycmd` server. The YCM client running inside Vim always uses the Python interpreter that's embedded inside Vim.

Default: `''`

```
let g:ycm_path_to_python_interpreter = ''
```

The `g:ycm_server_use_vim_stdout` option

By default, the `ycmd` completion server writes logs to logfiles. When this option is set to `1`, the server writes logs to Vim's stdout (so you'll see them in the console).

Default: `0`

```
let g:ycm_server_use_vim_stdout = 0
```

The `g:ycm_server_keep_logfiles` option

When this option is set to `1`, the `ycmd` completion server will keep the logfiles around after shutting down (they are deleted on shutdown by default).

To see where the logfiles are, call `:YcmDebugInfo`.

Default: `0`

```
let g:ycm_server_keep_logfiles = 0
```

The `g:ycm_server_log_level` option

The logging level that the `ycmd` completion server uses. Valid values are the following, from most verbose to least verbose:

- `debug`
- `info`
- `warning`
- `error`
- `critical`

Note that `debug` is *very* verbose.

Default: `info`

```
let g:ycm_server_log_level = 'info'
```

The `g:ycm_auto_start_csharp_server` option

When set to `1`, the OmniSharp server will be automatically started (once per Vim session) when you open a C# file.

Default: `1`

```
let g:ycm_auto_start_csharp_server = 1
```

The `g:ycm_auto_stop_csharp_server` option

When set to `1`, the OmniSharp server will be automatically stopped upon closing Vim.

Default: `1`

```
let g:ycm_auto_stop_csharp_server = 1
```

The `g:ycm_csharp_server_port` option

When g:ycm_auto_start_csharp_server is set to `1`, specifies the port for the OmniSharp server to listen on. When set to `0` uses an unused port provided by the OS.

Default: `0`

```
let g:ycm_csharp_server_port = 0
```

The `g:ycm_csharp_insert_namespace_expr` option

By default, when YCM inserts a namespace, it will insert the `using` statement under the nearest `using` statement. You may prefer that the `using` statement is inserted somewhere, for example, to preserve sorting. If so, you can set this option to override this behaviour.

When this option is set, instead of inserting the `using` statement itself, YCM will set the global variable `g:ycm_namespace_to_insert` to the namespace to insert, and then evaluate this option's value as an expression. The option's expression is responsible for inserting the namespace - the default insertion will not occur.

Default: ''

```
let g:ycm_csharp_insert_namespace_expr = ''
```

The `g:ycm_add_preview_to_completeopt` option

When this option is set to `1`, YCM will add the `preview` string to Vim's `completeopt` option (see `:h completeopt`). If your `completeopt` option already has `preview` set, there will be no effect. You can see the current state of your `completeopt` setting with `:set completeopt?` (yes, the question mark is important).

When `preview` is present in `completeopt`, YCM will use the `preview` window at the top of the file to store detailed information about the current completion candidate (but only if the candidate came from the semantic engine). For instance, it would show the full function prototype and all the function overloads in the window if the current completion is a function name.

Default: `0`

```
let g:ycm_add_preview_to_completeopt = 0
```

The `g:ycm_autoclose_preview_window_after_completion` option

When this option is set to `1`, YCM will auto-close the `preview` window after the user accepts the offered completion string. If there is no `preview` window triggered because there is no `preview` string in `completeopt`, this option is irrelevant. See the `g:ycm_add_preview_to_completeopt` option for more details.

Default: `0`

```
let g:ycm_autoclose_preview_window_after_completion = 0
```

The `g:ycm_autoclose_preview_window_after_insertion` option

When this option is set to `1`, YCM will auto-close the `preview` window after the user leaves insert mode. This option is irrelevant if `g:ycm_autoclose_preview_window_after_completion` is set or if no `preview` window is triggered. See the `g:ycm_add_preview_to_completeopt` option for more details.

Default: `0`

```
let g:ycm_autoclose_preview_window_after_insertion = 0
```

The `g:ycm_max_diagnostics_to_display` option

This option controls the maximum number of diagnostics shown to the user when errors or warnings are detected in the file. This option is only relevant if you are using the C-family semantic completion engine.

Default: `30`

```
let g:ycm_max_diagnostics_to_display = 30
```

The `g:ycm_key_list_select_completion` option

This option controls the key mappings used to select the first completion string. Invoking any of them repeatedly cycles forward through the completion list.

Some users like adding `<Enter>` to this list.

Default: `['<TAB>', '<Down>']`

```
let g:ycm_key_list_select_completion = ['<TAB>', '<Down>']
```

The g:ycm_key_list_previous_completion option

This option controls the key mappings used to select the previous completion string. Invoking any of them repeatedly cycles backwards through the completion list.

Note that one of the defaults is `<S-TAB>` which means Shift-TAB. That mapping will probably only work in GUI Vim (Gvim or MacVim) and not in plain console Vim because the terminal usually does not forward modifier key combinations to Vim.

Default: `['<S-TAB>', '<Up>']`

```
let g:ycm_key_list_previous_completion = ['<S-TAB>', '<Up>']
```

The g:ycm_key_invoke_completion option

This option controls the key mapping used to invoke the completion menu for semantic completion. By default, semantic completion is trigged automatically after typing `.`, `->` and `::` in insert mode (if semantic completion support has been compiled in). This key mapping can be used to trigger semantic completion anywhere. Useful for searching for top-level functions and classes.

Console Vim (not Gvim or MacVim) passes `<Nul>` to Vim when the user types `<C-Space>` so YCM will make sure that `<Nul>` is used in the map command when you're editing in console Vim, and `<C-Space>` in GUI Vim. This means that you can just press `<C-Space>` in both console and GUI Vim and YCM will do the right thing.

Setting this option to an empty string will make sure no mapping is created.

Default: `<C-Space>`

```
let g:ycm_key_invoke_completion = '<C-Space>'
```

The g:ycm_key_detailed_diagnostics option

This option controls the key mapping used to show the full diagnostic text when the user's cursor is on the line with the diagnostic. It basically calls `:YcmShowDetailedDiagnostic`.

Setting this option to an empty string will make sure no mapping is created.

Default: `<leader>d`

```
let g:ycm_key_detailed_diagnostics = '<leader>d'
```

The g:ycm_global_ycm_extra_conf option

Normally, YCM searches for a `.ycm_extra_conf.py` file for compilation flags (see the User Guide for more details on how this works). This option specifies a fallback path to a config file which is used if no `.ycm_extra_conf.py` is found.

You can place such a global file anywhere in your filesystem.

Default: `''`

```
let g:ycm_global_ycm_extra_conf = ''
```

The g:ycm_confirm_extra_conf option

When this option is set to `1` YCM will ask once per `.ycm_extra_conf.py` file if it is safe to be loaded. This is to prevent execution of malicious code from a `.ycm_extra_conf.py` file you didn't write.

To selectively get YCM to ask/not ask about loading certain `.ycm_extra_conf.py` files, see the `g:ycm_extra_conf_globlist` option.

Default: `1`

```
let g:ycm_confirm_extra_conf = 1
```

The g:ycm_extra_conf_globlist option

This option is a list that may contain several globbing patterns. If a pattern starts with a `!` all `.ycm_extra_conf.py` files matching that pattern will be blacklisted, that is they won't be loaded and no confirmation dialog will be shown. If a pattern does not start with a `!` all files matching that pattern will be whitelisted. Note that this option is not used when confirmation is disabled using `g:ycm_confirm_extra_conf` and that items earlier in the list will take precedence over the later ones.

Rules:

- `*` matches everything
- `?` matches any single character
- `[seq]` matches any character in seq
- `[!seq]` matches any char not in seq

Example:

```
let g:ycm_extra_conf_globlist = ['~/dev/*','!~/*']
```

- The first rule will match everything contained in the `~/dev` directory so `.ycm_extra_conf.py` files from there will be loaded.
- The second rule will match everything in the home directory so a `.ycm_extra_conf.py` file from there won't be loaded.
- As the first rule takes precedence everything in the home directory excluding the `~/dev` directory will be blacklisted.

NOTE: The glob pattern is first expanded with Python's `os.path.expanduser()` and then resolved with `os.path.abspath()` before being matched against the filename.

Default: `[]`

```
let g:ycm_extra_conf_globlist = []
```

The g:ycm_filepath_completion_use_working_dir option

By default, YCM's filepath completion will interpret relative paths like `../` as being relative to the folder of the file of the currently active buffer. Setting this option will force YCM to always interpret relative paths as being relative to Vim's current working directory.

Default: `0`

```
let g:ycm_filepath_completion_use_working_dir = 0
```

The g:ycm_semantic_triggers option

This option controls the character-based triggers for the various semantic completion engines. The option holds a dictionary of key-values, where the keys are Vim's filetype strings delimited by commas and values are lists of strings, where the strings are the triggers.

Setting key-value pairs on the dictionary *adds* semantic triggers to the internal default set (listed below). You cannot remove the default triggers, only add new ones.

A "trigger" is a sequence of one or more characters that trigger semantic completion when typed. For instance, C++ (`cpp` filetype) has `.` listed as a trigger. So when the user types `foo.` , the semantic engine will trigger and serve `foo` 's list of member functions and variables. Since C++ also has `->` listed as a trigger, the same thing would happen when the user typed `foo->` .

It's also possible to use a regular expression as a trigger. You have to prefix your trigger with `re!` to signify it's a regex trigger. For instance, `re!\w+\.` would only trigger after the `\w+\.` regex matches.

NOTE: The regex syntax is **NOT** Vim's, it's Python's.

Default: `[see next line]`

```
let g:ycm_semantic_triggers =  {
  \   'c' : ['->', '.'],
  \   'objc' : ['->', '.', 're!\[[_a-zA-Z]+\w*\s', 're!^\s*[^\W\d]\w*\s',
  \             're!\[.*\]\s'],
  \   'ocaml' : ['.', '#'],
  \   'cpp,objcpp' : ['->', '.', '::'],
  \   'perl' : ['->'],
  \   'php' : ['->', '::'],
  \   'cs,java,javascript,typescript,d,python,perl6,scala,vb,elixir,go' : ['.'],
  \   'ruby' : ['.', '::'],
  \   'lua' : ['.', ':'],
  \   'erlang' : [':'],
  \ }
```

The `g:ycm_cache_omnifunc` option

Some omnicompletion engines do not work well with the YCM cache—in particular, they might not produce all possible results for a given prefix. By unsetting this option you can ensure that the omnicompletion engine is re-queried on every keypress. That will ensure all completions will be presented, but might cause stuttering and lagginess if the omnifunc is slow.

Default: `1`

```
let g:ycm_cache_omnifunc = 1
```

The `g:ycm_use_ultisnips_completer` option

By default, YCM will query the UltiSnips plugin for possible completions of snippet triggers. This option can turn that behavior off.

Default: `1`

```
let g:ycm_use_ultisnips_completer = 1
```

The `g:ycm_goto_buffer_command` option

Defines where `GoTo*` commands result should be opened. Can take one of the following values: `[ 'same-buffer', 'horizontal-split', 'vertical-split', 'new-tab', 'new-or-existing-tab' ]` If this option is set to the `'same-buffer'` but current buffer can not be switched (when buffer is modified and `nohidden` option is set), then result will be opened in horizontal split.

Default: `'same-buffer'`

```
let g:ycm_goto_buffer_command = 'same-buffer'
```

The `g:ycm_disable_for_files_larger_than_kb` option

Defines the max size (in Kb) for a file to be considered for completion. If this option is set to 0 then no check is made on the size of the file you're opening.

Default: 1000

```
let g:ycm_disable_for_files_larger_than_kb = 1000
```

FAQ

I used to be able to `import vim` in `.ycm_extra_conf.py`, but now can't

YCM was rewritten to use a client-server architecture where most of the logic is in the `ycmd` server. So the magic `vim` module you could have previously imported in your `.ycm_extra_conf.py` files doesn't exist anymore.

To be fair, importing the magic `vim` module in extra conf files was never supported in the first place; it only ever worked by accident and was never a part of the extra conf API.

But fear not, you should be able to tweak your extra conf files to continue working by using the `g:ycm_extra_conf_vim_data` option. See the docs on that option for details.

On very rare occasions Vim crashes when I tab through the completion menu

That's a very rare Vim bug most users never encounter. It's fixed in Vim 7.4.72. Update to that version (or above) to resolve the issue.

I get a linker warning regarding `libpython` on Mac when compiling YCM

If the warning is `ld: warning: path '/usr/lib/libpython2.7.dylib' following -L not a directory`, then feel free to ignore it; it's caused by a limitation of CMake and is not an issue. Everything should still work fine.

I get a weird window at the top of my file when I use the semantic engine

This is Vim's `preview` window. Vim uses it to show you extra information about something if such information is available. YCM provides Vim with such extra information. For instance, when you select a function in the completion list, the `preview` window will hold that function's prototype and the prototypes of any overloads of the function. It will stay there after you select the completion so that you can use the information about the parameters and their types to write the function call.

If you would like this window to auto-close after you select a completion string, set the `g:ycm_autoclose_preview_window_after_completion` option to `1` in your `vimrc` file. Similarly, the `g:ycm_autoclose_preview_window_after_insertion` option can be set to close the `preview` window after leaving insert mode.

If you don't want this window to ever show up, add `set completeopt-=preview` to your `vimrc`. Also make sure that the `g:ycm_add_preview_to_completeopt` option is set to `0`.

It appears that YCM is not working

In Vim, run `:messages` and carefully read the output. YCM will echo messages to the message log if it encounters problems. It's likely you misconfigured something and YCM is complaining about it.

Also, you may want to run the `:YcmDebugInfo` command; it will make YCM spew out various debugging information, including the compile flags for the file if the file is a C-family language file and you have compiled in Clang support.

Sometimes it takes much longer to get semantic completions than normal

This means that libclang (which YCM uses for C-family semantic completion) failed to pre-compile your file's preamble. In other words, there was an error compiling some of the source code you pulled in through your header files. I suggest calling the `:YcmDiags` command to see what they were.

Bottom line, if libclang can't pre-compile your file's preamble because there were errors in it, you're going to get slow completions because there's no AST cache.

YCM auto-inserts completion strings I don't want!

This means you probably have some mappings that interfere with YCM's internal ones. Make sure you don't have something mapped to `<C-p>`, `<C-x>` or `<C-u>` (in insert mode).

YCM *never* selects something for you; it just shows you a menu and the user has to explicitly select something. If something is being selected automatically, this means there's a bug or a misconfiguration somewhere.

I get a `E227: mapping already exists for <blah>` error when I start Vim

This means that YCM tried to set up a key mapping but failed because you already had something mapped to that key combination. The `<blah>` part of the message will tell you what was the key combination that failed.

Look in the *Options* section and see if any of the default mappings conflict with your own. Then change that option value to something else so that the conflict goes away.

I get `'GLIBC_2.XX' not found (required by libclang.so)` when starting Vim

Your system is too old for the precompiled binaries from llvm.org. Compile Clang on your machine and then link against the `libclang.so` you just produced. See the full installation guide for help.

I'm trying to use a Homebrew Vim with YCM and I'm getting segfaults

Something (I don't know what) is wrong with the way that Homebrew configures and builds Vim. I recommend using MacVim. Even if you don't like the MacVim GUI, you can use the Vim binary that is inside the MacVim.app package (it's `MacVim.app/Contents/MacOS/Vim`) and get the Vim console experience.

I have a Homebrew Python and/or MacVim; can't compile/SIGABRT when starting

You should probably run `brew rm python; brew install python` to get the latest fixes that should make YCM work with such a configuration. Also rebuild Macvim then. If you still get problems with this, see issue #18 for suggestions.

Vim segfaults when I use the semantic completer in Ruby files

This was caused by a Vim bug. Update your version of Vim (Vim 7.3.874 is known to work, earlier versions may also fix this issue).

I get `LONG_BIT definition appears wrong for platform` when compiling

Look at the output of your CMake call. There should be a line in it like the following (with `.dylib` in place of `.so` on a Mac):

```
-- Found PythonLibs: /usr/lib/libpython2.7.so (Required is at least version "2.5")
```

That would be the **correct** output. An example of **incorrect** output would be the following:

```
-- Found PythonLibs: /usr/lib/libpython2.7.so (found suitable version "2.5.1", minimum required
```

Notice how there's an extra bit of output there, the `found suitable version "<version>"` part, where `<version>` is not the same as the version of the dynamic library. In the example shown, the library is version 2.7 but the second string is version `2.5.1`.

This means that CMake found one version of Python headers and a different version for the library. This is wrong. It can happen when you have multiple versions of Python installed on your machine.

You should probably add the following flags to your cmake call (again, `dylib` instead of `so` on a Mac):

```
-DPYTHON_INCLUDE_DIR=/usr/include/python2.7 -DPYTHON_LIBRARY=/usr/lib/libpython2.7.so
```

This will force the paths to the Python include directory and the Python library to use. You may need to set these flags to something else, but you need to make sure you use the same version of Python that your Vim binary is built against, which is highly likely to be the system's default Python.

I get `libpython2.7.a [...] relocation R_X86_64_32` when compiling

The error is usually encountered when compiling YCM on Centos or RHEL. The full error looks something like the following:

```
/usr/bin/ld: /usr/local/lib/libpython2.7.a(abstract.o): relocation R_X86_64_32 against `a local
```

It's possible to get a slightly different error that's similar to the one above. Here's the problem and how you solve it:

Your `libpython2.7.a` was not compiled with `-fPIC` so it can't be linked into `ycm_core.so`. Use the `-DPYTHON_LIBRARY=` CMake flag to point it to a `.so` version of libpython on your machine (for instance, `-DPYTHON_LIBRARY=/usr/lib/libpython2.7.so`). Naturally, this means you'll have to go through the full installation guide by hand.

I get `Vim: Caught deadly signal SEGV` on Vim startup

This can happen on some Linux distros. If you encounter this situation, run Vim under `gdb`. You'll probably see something like this in the output when Vim crashes:

```
undefined symbol: clang_CompileCommands_dispose
```

This means that Vim is trying to load a `libclang.so` that is too old. You need at least a 3.2 libclang. Some distros ship with a system `libclang.so` that identifies itself as 3.2 but is not; it was cut from the upstream sources before the official 3.2 release and some API changes (like the addition of the CompileCommands API) were added after their cut.

So just go through the installation guide and make sure you are using a correct `libclang.so`. I recommend downloading prebuilt binaries from llvm.org.

YCM does not read identifiers from my tags files

First, put `let g:ycm_collect_identifiers_from_tags_files = 1` in your vimrc.

Make sure you are using Exuberant Ctags to produce your tags files since the only supported tag format is the Exuberant Ctags format. The format from "plain" ctags is NOT supported. The output of `ctags --version` should list "Exuberant Ctags".

Ctags needs to be called with the `--fields=+l` (that's a lowercase `L`, not a one) option because YCM needs the `language:<lang>` field in the tags output.

NOTE: Mac OS X comes with "plain" ctags installed by default. `brew install ctags` will get you the Exuberant Ctags version.

Also make sure that your Vim `tags` option is set correctly. See `:h 'tags'` for details. If you want to see which tag files YCM will read for a given buffer, run `:echo tagfiles()` with the relevant buffer active. Note that that function will only list tag files that already exist.

`CTRL-U` in insert mode does not work

YCM keeps you in a `completefunc` completion mode when you're typing in insert mode and Vim disables `<C-U>` in completion mode as a "feature." Sadly there's nothing I can do about this.

YCM conflicts with UltiSnips TAB key usage

YCM comes with support for UltiSnips (snippet suggestions in the popup menu), but you'll have to change the UltiSnips mappings. See `:h UltiSnips-triggers` in Vim for details. You'll probably want to change some/all of the following options:

```
g:UltiSnipsExpandTrigger
g:UltiSnipsJumpForwardTrigger
g:UltiSnipsJumpBackwardTrigger
```

Why isn't YCM just written in plain VimScript, FFS?

Because of the identifier completion engine and subsequence-based filtering. Let's say you have *many* dozens of files open in a single Vim instance (I often do); the identifier-based engine then needs to store thousands (if not tens of thousands) of identifiers in its internal data-structures. When the user types, YCM needs to perform subsequence-based filtering on *all* of those identifiers (every single one!) in less than 10 milliseconds.

I'm sorry, but that level of performance is just plain impossible to achieve with VimScript. I've tried, and the language is just too slow. No, you can't get acceptable performance even if you limit yourself to just the identifiers in the current file and simple prefix-based filtering.

Why does YCM demand such a recent version of Vim?

During YCM's development several show-stopper bugs were encountered in Vim. Those needed to be fixed upstream (and were). A few months after those bugs were fixed, Vim trunk landed the `pyeval()` function which improved YCM performance even more since less time was spent serializing and deserializing data between Vim and the embedded Python interpreter. A few critical bugfixes for `pyeval()` landed in Vim 7.3.584 (and a few commits before that).

I get annoying messages in Vim's status area when I type

If you're referring to the `User defined completion <bla bla> back at original` and similar, then just update to Vim 7.4.314 (or later) and they'll go away.

Nasty bugs happen if I have the `vim-autoclose` plugin installed

Use the delimitMate plugin instead. It does the same thing without conflicting with YCM.

Is there some sort of YCM mailing list? I have questions

If you have questions about the plugin or need help, please use the ycm-users mailing list, *don't* create issues on the tracker. The tracker is for bug reports and feature requests.

I get an internal compiler error when installing

This can be a problem on virtual servers with limited memory. A possible solution is to add more swap memory. A more practical solution would be to force the build script to run only one compile job at a time. You can do this by setting the `YCM_CORES` environment variable to `1`. Example:

```
YCM_CORES=1 ./install.py --clang-completer
```

I get weird errors when I press `Ctrl-C` in Vim

Never use `Ctrl-C` in Vim.

Using `Ctrl-C` to exit insert mode in Vim is a bad idea. The main issue here is that `Ctrl-C` in Vim doesn't just leave insert mode, it leaves it without triggering `InsertLeave` autocommands (as per Vim docs). This is a bad idea and is likely to break many other things and not just YCM.

Bottom line, if you use `Ctrl-C` to exit insert mode in Vim, you're gonna have a bad time.

If pressing `<esc>` is too annoying (agreed, it is), we suggest mapping it to something more convenient. On a QWERTY keyboard, a good pick for the `<esc>` map is `inoremap jk <Esc>`. This is right on the home row, it's an incredibly rare digraph in English and if you ever need to type those two chars in sequence in insert mode, you just type `j`, then wait 500ms, then type `k`.

Why did YCM stop using Syntastic for diagnostics display?

Previously, YCM would send any diagnostics it would receive from the libclang semantic engine to Syntastic for display as signs in the gutter, red squiggles etc. Today, YCM uses its own code to do that.

Using Syntastic for this was always a kludge. Syntastic assumes its "checker" plugins behave in a certain way; those assumptions have never fit YCM. For instance, YCM continuously recompiles your code in the background for C-family languages and tries to push new diagnostics to the user as fast as possible, even while the user types.

Syntastic assumes that a checker only runs on file save ("active" mode) or even less frequently, when the user explicitly invokes it ("passive" mode). This mismatch in assumptions causes performance problems since Syntastic code isn't optimized for this use case of constant diagnostic refreshing.

Poor support for this use case also led to crash bugs in Vim caused by Syntastic-Vim interactions (issue #593) and other problems, like random Vim flickering. Attempts were made to resolve these issues in Syntastic, but ultimately some of them failed (for various reasons).

Implementing diagnostic display code directly in YCM resolves all of these problems. Performance also improved substantially since the relevant code is now written in Python instead of VimScript

(which is very slow) and is tailored only for YCM's use-cases. We were also able to introduce new features in this area since we're now not limited to the Syntastic checker API.

We've tried to implement this in the most backwards-compatible way possible; YCM options that control diagnostic display fall back to Syntastic options that control the same concepts if the user has those set.

Still, some Syntastic-specific configuration you might have had might not be supported by the new code. Please file issues on the tracker in such cases; if we find the request to be reasonable, we'll find a way to address it.

Completion doesn't work with the C++ standard library headers

This is caused by an issue with libclang that only affects some operating systems. Compiling with `clang` the binary will use the correct default header search paths but compiling with `libclang.so` (which YCM uses) does not.

Mac OS X is normally affected, but there's a workaround in YCM for that specific OS. If you're not running that OS but still have the same problem, continue reading.

The workaround is to call `echo | clang -v -E -x c++ -` and look at the paths under the `#include <...> search starts here:` heading. You should take those paths, prepend `-isystem` to each individual path and append them all to the list of flags you return from your `FlagsForFile` function in your `.ycm_extra_conf.py` file.

See issue #303 for details.

Install YCM with NeoBundle

NeoBundle can do the compilation for you; just add the following to your vimrc:

```
NeoBundle 'Valloric/YouCompleteMe', {
     \ 'build'      : {
        \ 'mac'     : './install.py --clang-completer --system-libclang --omnisharp-completer',
        \ 'unix'    : './install.py --clang-completer --system-libclang --omnisharp-completer',
        \ 'windows' : './install.py --clang-completer --system-libclang --omnisharp-completer',
        \ 'cygwin'  : './install.py --clang-completer --system-libclang --omnisharp-completer'
        \ }
     \ }
```

But you could have problems with the time needed to get the sub modules and compile the whole thing. To increase the Neobundle timeout to 1500 seconds, add the following to your vimrc:

```
let g:neobundle#install_process_timeout = 1500
```

Contact

If you have questions about the plugin or need help, please use the ycm-users mailing list.

If you have bug reports or feature suggestions, please use the issue tracker.

The latest version of the plugin is available at http://valloric.github.io/YouCompleteMe/.

The author's homepage is http://val.markovic.io.

License