Vim as an IDE

Davis Claiborne

LUG @ NC State

March 11, 2022

Linux Users Group
at NC State University

Presentation Overview

Introduction

QuickFix List

Reading / Writing Aids

Miscellaneous

Main Idea

"Unix Philosophy"

Let other tools do the dirty work; parse their output

"Unix Philosophy"
Let other tools do the dirty work; parse their output

- Even if you don't use Vim, you can takeaway some things from this presentation
- That's because Vim's main philosophy is to let other tools do the dirty work, and just parse their output and provide it in a way that is convenient for the user to use
- This mindset can be thought of as an extension of the Unix philosophy
- While largely applicable, Vim definitely has outgrown this to some extent, especially in the most literal sense, e.g. built-in search instead of using grep, mostly in order to make certain functionality easier to use or in order to provide a better experience in any environment

Assumed Familiarity

- Focus of this presentation: C/C++
- Why?
  - Supported by default
    - More generalized
    - More consistent
  - More personal experience
- Other languages are supported

Assumed Familiarity

- Focus of this presentation: C/C++
- Why?
  - Supported by default
    - More generalized
    - More consistent
  - More personal experience
- Other languages are supported

- Before I get into the content itself, I should note that the majority of this presentation has to do with the C/C++ way of doing things, i.e. using GCC, GDB, makefiles, etc.
- I also assume some familiarity with these tools - while extremely useful, covering them could be several presentations in and of themselves. It shouldn't be required to understand what's going, but some background knowledge will be helpful
- The presentation is like this for two main reasons: first, it's what Vim supports and expects by default, which in turn means it will be more broadly applicable and also less reliant on plugins, which may fall out of favor or popularity; second, it's just what I have the most experience in
- If you don't like the C way of doing things, don't worry: Vim has a huge ecosystem of plugins you can use for other languages, some of which I'll show off later in the presentation

Vim 7 vs Vim 8 vs Neovim

- Focused on Vim 8
- Vim 7
  - Mostly compatible
  - Lacks some nicer features
- Neovim
  - Probably better
  - Haven't tried it

Vim 7 vs Vim 8 vs Neovim

- Focused on Vim 8
- Vim 7
  - Mostly compatible
  - Lacks some nicer features
- Neovim
  - Probably better
  - Haven't tried it

- Additionally, this presentation is on Vim 8, not Vim 7 or Neovim
- Compatibility between Vim 7 and Vim 8 is pretty good, but Vim 8 introduced a lot of fancy new features which you won't be able to make use of if you're stuck on 7. I'll make a note of this when applicable, and suggest some alternatives.
- For personal uses, this shouldn't be a huge issue, but can be an issue if you're stuck developing using older servers. Hopefully this will become less and less of an issue over time, as sysadmins update to releases with more modern versions of Vim.
- Finally, I'll try and preempt a question many of you are probably asking - "Will this work for Neovim?" The answer is, probably yes, and there might even be better plugins available for it as well that I'm not familiar with (though I haven't personally tested anything on Neovim).
- And to preempt another question - why I'm not using Neovim - I've been meaning to check out Neovim for a while, and have just been busy/lazy.

QuickFix List Overview

- QuickFix List
  - Introduction
  - Navigation
  - Advanced Usage

What is the QuickFix List?

- List of specific points in specific files
- Designed for compile-edit cycle
  - More generally useful
  - For compilation: `:make`[1]
- Vim must be compiled with `+quickfix`

[1] Assuming Vim's current directory contains a makefile - see `:help :make` for more

What is the QuickFix List?

- List of specific points in specific files
- Designed for compile-edit cycle
  - More generally useful
  - For compilation: `:make`[1]
- Vim must be compiled with `+quickfix`

[1] Assuming Vim's current directory contains a makefile - see `:help :make` for more

- The first thing I'd like to talk about is the QuickFix list
- At a high level, the QuickFix list is a list of points, i.e. a line number and/or column number, in specific files that you can easily move back and forth between (more on this later)
- The QuickFix list was specifically designed with the idea of capturing and displaying error messages from compilation, allowing for quick and easy navigation to the specified lines, but are actually designed in such a way to be much more generally useful
- To invoke it in what can be thought of as "compilation mode," use the command `:make` (assuming that there is a makefile in Vim's current directory)
- Note that, in order to make use of these features, Vim must be compiled with the `+quickfix` feature

Basic Example

If you have the following code:

```
int main() {
        // Did not declare a or b
        a = 1;
        b = 2;
}
```

With a basic makefile in the current directory, then run `:make`

The make command will run and it will show the shell and its output. You can press "Enter" to continue.

At the bottom of the window, you'll now see the following text:

```
(3 of 10): error: `a' undeclared (first use in this function)
```

When you press "Enter" again, your cursor will also be moved to this location

- Quick demo of basic QF functionality

Basic Example

If you have the following code:

```
int main() {
        // Did not declare a or b
        a = 1;
        b = 2;
}
```

With a basic makefile in the current directory, then run `:make`

The make command will run and it will show the shell and its output. You can press "Enter" to continue.

At the bottom of the window, you'll now see the following text:

```
(3 of 10): error: `a' undeclared (first use in this function)
```

When you press "Enter" again, your cursor will also be moved to this location

Understanding the Output

What does “3 of 10” mean?

- 3: Line number containing filename, row, column, and syntax error
- 10: The number of lines output from the makefile

```
cc    -c -o example.o example.c
example.c: In function ‘main’:
example.c:3:9: error: ‘a’ undeclared (first use in this function)
    3 |         a = 1;
      |         ^
example.c:3:9: note: each undeclared identifier is reported only once for each function...
example.c:4:9: error: ‘b’ undeclared (first use in this function)
    4 |         b = 2;
      |         ^
make: *** [<builtin>: example.o] Error 1
```

Understanding the Output

What does "3 of 10" mean?

- 3: Line number containing filename, row, column, and syntax error
- 10: The number of lines output from the makefile

```
cc     -c -o example.o example.c
example.c: In function 'main':
example.c:3:9: error: 'a' undeclared (first use in this function)
    3 |         a = 1;
      |         ^
example.c:3:9: note: each undeclared identifier is reported only once for each function...
example.c:4:9: error: 'b' undeclared (first use in this function)
    4 |         b = 2;
      |         ^
make: *** [<builtin>: example.o] Error 1
```

- Okay, so that was a pretty basic example, but even now you can see some basic utility
- But you may be wondering something about the previous output: what does "3 of 10" mean? There's only 2 errors, and why is the first one number 3?
- The answer lies in the output from the Makefile itself:

Moving Between QuickFix Items

- Basic:
  - `:cn[ext]` / `:cp[revious]`
  - `:caf[ter]` / `:cbe[fore]`

Moving Between QuickFix Items

- Basic:
  - :cn[ext] / :cp[revious]
  - :caf[ter] / :cbe[fore]

- Now that you've seen some basic usage, you might be thinking, "That's great and all, but what if I want to look at something other than the first error?"
- You're in luck - there's a number of ways to move to the next error, the most basic being `:cn[ext]` and `:cp[revious]`, both of which can take a `count` argument and jump forwards or backwards in the list by that amount
- One thing to note for `:cn` and `:cp` is that their output doesn't totally line up with the line numbers, because the QuickFix list is smart enough to only count messages with line numbers
- You may find `:cafter` and `:cbefore` easier to use, since they let you move up or down in the QF list relative to the current QF location

Moving Between QuickFix Items

- Basic:
  - `:cn[ext]` / `:cp[revious]`
  - `:caf[ter]` / `:cbe[fore]`
  - `:cc [nr]`
    - Show line number `[nr]`
    - Move cursor to corresponding line

- If you do want to see a specific line number, you can use the `:cc` command, which does two things:
- First, in the command-line (the bottom of the window, where `:` commands are input) it shows the makefile output line number specified, e.g. `:cc 3` echos the third line of output
- Additionally, your cursor will also be moved to the file, column, and row corresponding to that error

Moving Between QuickFix Items

- Basic:
  - `:cn[ext]` / `:cp[revious]`
  - `:caf[ter]` / `:cbe[fore]`
  - `:cc [nr]`
    - Show line number `[nr]`
    - Move cursor to corresponding line
- Show all: `:cl[ist]`

Moving Between QuickFix Items

- Basic:
  - `:cn[ext]` / `:cp[revious]`
  - `:caf[ter]` / `:cbe[fore]`
  - `:cc [nr]`
    - Show line number `[nr]`
    - Move cursor to corresponding line
- Show all: `:cl[ist]`

- To show the entire list of items in the QuickFix window, the simplest way to do this would be to run `:clist`
- This shows all the items in the QuickFix list as a quick pop-up at the bottom of the window, and goes away when you press any key
- Additionally, you can specify a range, offsets, etc.

Moving Between QuickFix Items

- Basic:
  - `:cn[ext]` / `:cp[revious]`
  - `:caf[ter]` / `:cbe[fore]`
  - `:cc [nr]`
    - Show line number `[nr]`
    - Move cursor to corresponding line
- Show all: `:cl[ist]`
- Time travel:
  - Go backwards/forwards: `:col[der]` / `:cnew[er]`
  - Show surrounding history: `:chi[story]`

Moving Between QuickFix Items

- Basic:
  - `:cn[ext]` / `:cp[revious]`
  - `:caf[ter]` / `:cbe[fore]`
  - `:cc [nr]`
    - Show line number `[nr]`
    - Move cursor to corresponding line
- Show all: `:cl[ist]`
- Time travel:
  - Go backwards/forwards: `:col[der]` / `:cnew[er]`
  - Show surrounding history: `:chi[story]`

- Finally, you may be thinking, that this would be way more useful if you could go back and use a previous QuickFix list from earlier
- Good news: you can!
- `:colder` and `:cnewer` allow you to go backwards and forwards through the QuickFix list, while `:chistory` allows you to see this history surrounding the current list

The QuickFix Window

- Open the QuickFix window: `:cope[n]`
- New window with QuickFix contents
  - Not editable[1]
- `<CR>` to jump to corresp. line
  - Changes the window above the QuickFix window (if not already open in current tab)
  - Open a new window: `CTRL-W_<Enter>`

[1] By default; see Modifying the QuickFix List

The QuickFix Window

- Open the QuickFix window: `:cope[n]`
- New window with QuickFix contents
  - Not editable[1]
- <CR> to jump to corresp. line
  - Changes the window above the QuickFix window (if not already open in current tab)
  - Open a new window: `CTRL-W_<Enter>`

[1] By default; see Modifying the QuickFix List

- You might consider that a bit clunky, however, or you might just prefer to see a listing with all the Makefile's output. For this, you might prefer to use `:copen`, which opens a new window containing the QuickFix output
- You can navigate this file just like any other one - using j, k, searching, etc. Note that, by default, you can't modify this file. I'll talk about this more in a bit.
- In addition to the regular navigation, you can hit "enter," to jump to that error
- If that file is already on the current tab, the cursor will jump to that location; if not, one of the windows above the QuickFix window will be changed to that location
- If you'd like to avoid this, you can use `CTRL-W + Enter`

Vimgrep and Grep

- Put search outputs to QuickFix

[1] See examples in Search Examples

- Put search outputs to QuickFix

[1] See examples in Search Examples

- Vim features two commands for searching that populate the QuickFix list: `:vimgrep` and `:grep`
- Both of these commands offer a major advantage over searching using the command-line, or with other more typical search commands like `/` or `?`: they put all their results into the QuickFix list
- You may be wondering: what's the difference between `:grep` and `:vimgrep`? When should I use one or the other?
- The short answer is: use whichever one is more comfortable/useful for you.
- `:vimgrep` uses Vim's grep functionality and patterns, while `:grep` uses the `grep` command by default, but can be configured to use any program (that formats its output in a QuickFix-compatible way)

Vimgrep and Grep

- Put search outputs to QuickFix
- `:vimgrep /pattern/flags files`[1]
  - Use Vim's search functionality
  - Consistent
  - Works on any file Vim can read[2]

[1] See examples in Search Examples

[2] See `:help grep`

Vimgrep and Grep

- Put search outputs to QuickFix
- `:vimgrep /pattern/flags files`[1]
  - Use Vim's search functionality
  - Consistent
  - Works on any file Vim can read[2]

[1] See examples in Search Examples
[2] See `:help grep`

- In terms of advantages, `vimgrep`'s major advantages are that it uses Vim's search functionality, which means you don't have to learn another syntax
- It's also consistent across systems, so you don't need to worry about installing another package, since it will work wherever Vim works
- Finally, it works on any file Vim can read, including zipped files and remote files (see `:help grep` for more on that)

Vimgrep and Grep

- Put search outputs to QuickFix
- `:vimgrep /pattern/flags files`[1]
  - Use Vim's search functionality
  - Consistent
  - Works on any file Vim can read[2]
- `:grep [args]`
  - Use external functionality
  - `grep` by default
  - Can be configured[3]

[1] See examples in Search Examples

[2] See `:help grep`

[3] See `:help grepprg`

Vimgrep and Grep

- Put search outputs to QuickFix
- `:vimgrep /pattern/flags files`[1]
  - Use Vim's search functionality
  - Consistent
  - Works on any file Vim can read[2]
- `:grep [args]`
  - Use external functionality
  - `grep` by default
  - Can be configured[3]

[1] See examples in Search Examples
[2] See `:help grep`
[3] See `:help grepprg`

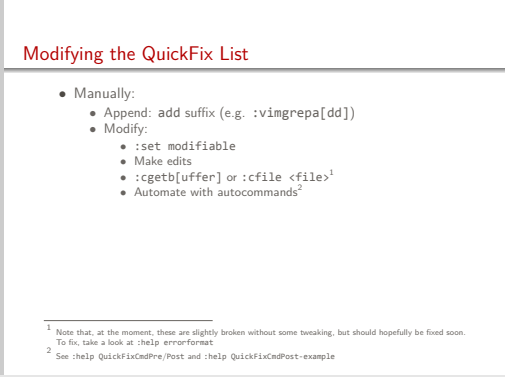
- Meanwhile, the main advantage to using `:grep` is that you can use other search utilities for whatever reason you want, be it that you find the syntax easier to use, it's faster, it supports additional options, etc.
- Virtually any search tool should be able to be dropped in and work - some, like silver searcher, have default flags intended for use with Vim's QuickFix list you can use. Others have popular and existing plugins you can use.
- Even if it doesn't natively support Vim's QuickFix list, so long as the search tool can report at least the file and line number to go to, any search tool can be used with a bit of fiddling

Modifying the QuickFix List

- Manually:
  - Append: `add` suffix (e.g. `:vimgrepa[dd]`)
  - Modify:
    - `:set modifiable`
    - Make edits
    - `:cgetb[uffer]` or `:cfile <file>`[1]
    - Automate with autocommands[2]

[1] Note that, at the moment, these are slightly broken without some tweaking, but should hopefully be fixed soon. To fix, take a look at `:help errorformat`

[2] See `:help QuickFixCmdPre/Post` and `:help QuickFixCmdPost-example`



- The easiest way to modify the QuickFix list is by appending. I won't discuss them all here just for the sake of brevity, but most commands have a corresponding "add" command for easy appending
- Modifying the QuickFix list in other ways is a bit more challenging
- By default, the QuickFix list is not modifiable, but there are still a few ways you can change the contents
- Probably one of the more obvious ways to do this is to do the following: make the file modifiable, make your edits, then either reload the buffer with `cgetbuffer`, or write and reload the file
- You may think this is a bit clunky. And you'd be right, which is why you can also use autocommands to automate this to some extent.

Modifying the QuickFix List

- Manually:
  - Append: `add` suffix (e.g. `:vimgrepa[dd]`)
  - Modify:
    - `:set modifiable`
    - Make edits
    - `:cgetb[uffer]` or `:cfile <file>`[1]
    - Automate with autocommands[2]
- Cfilter:
  - Introduced in 8.1
  - Enable: `:packadd cfilter`
  - Filter: `:Cfilter /pat/`

[1] Note that, at the moment, these are slightly broken without some tweaking, but should hopefully be fixed soon. To fix, take a look at `:help errorformat`

[2] See `:help QuickFixCmdPre/Post` and `:help QuickFixCmdPost-example`

Modifying the QuickFix List

- Manually:
  - Append: `add` suffix (e.g. `:vimgrepa[dd]`)
  - Modify:
    - `:set modifiable`
    - Make edits
    - `:cgetb[uffer]` or `:cfile <file>`[1]
    - Automate with autocommands[2]
- Cfilter:
  - Introduced in 8.1
  - Enable: `:packadd cfilter`
  - Filter: `:Cfilter /pat/`

[1] Note that, at the moment, these are slightly broken without some tweaking, but should hopefully be fixed soon. To fix, take a look at `:help errorformat`
[2] See `:help QuickFixCmdPre/Post` and `:help QuickFixCmdPost-example`

- Vim also has a native way of filtering the QuickFix list, as of Vim 8.1, called the Cfilter plugin
- It's currently not enabled by default, but you can enable it in your vimrc with `packadd cfilter`
- This plugin allows you to use Vim regexes to further filter your QuickFix window using the `:Cfilter` command

Modifying the QuickFix List

- Manually:
  - Append: `add` suffix (e.g. `:vimgrepa[dd]`)
  - Modify:
    - `:set modifiable`
    - Make edits
    - `:cgetb[uffer]` or `:cfile <file>`[1]
    - Automate with autocommands[2]
- Cfilter:
  - Introduced in 8.1
  - Enable: `:packadd cfilter`
  - Filter: `:Cfilter /pat/`
- Many plugins: [5] [7] [8]

[1] Note that, at the moment, these are slightly broken without some tweaking, but should hopefully be fixed soon. To fix, take a look at `:help errorformat`

[2] See `:help QuickFixCmdPre/Post` and `:help QuickFixCmdPost-example`

Modifying the QuickFix List

- Manually:
  - Append: add suffix (e.g. `:vimgrepa[dd]`)
  - Modify:
    - `:set modifiable`
    - Make edits
    - `:cgetb[uffer]` or `:cfile <file>`[1]
    - Automate with autocommands[2]
- Cfilter:
  - Introduced in 8.1
  - Enable: `:packadd cfilter`
  - Filter: `:Cfilter /pat/`
- Many plugins: [5] [7] [8]

[1] Note that, at the moment, these are slightly broken without some tweaking, but should hopefully be fixed soon. To fix, take a look at `:help errorformat`
[2] See `:help QuickFixCmdPre/Post` and `:help QuickFixCmdPost-example`

- Finally, there are a ton of plugins you can use to allow modifying the QuickFix list with a bit less customization
- I haven't used any of them personally, but if you want something that "just works," these may be worth looking in to

Configuring Other Tools

- Not comprehensive[1]

[1] See `:help quickfix.txt`

- Not comprehensive[1]

[1] See :help quickfix.txt

- For the sake of keeping this from turning into a "Vim QuickFix list presentation," I'll close with some of the more "IDE-like" features you can configure, but I highly recommend reading more about the QuickFix list - see `:help quickfix.txt` for more

Configuring Other Tools

- Not comprehensive[1]
- Change compiler: `:compiler <name>`
  - (Selected) native support:[2]
    - GCC
    - PyUnit
    - TeX
  - Unsupported: Plugins / Hack your own![3]

[1] See `:help quickfix.txt`
[2] See `:help compiler-select` for more
[3] See `:help write-compiler-plugin`

Configuring Other Tools

- Not comprehensive[1]
- Change compiler: `:compiler <name>`
  - (Selected) native support:[2]
    - GCC
    - PyUnit
    - TeX
  - Unsupported: Plugins / Hack your own![3]

[1] See :help quickfix.txt
[2] See :help compiler-select for more
[3] See :help write-compiler-plugin

- One of the more useful commands is `compiler`, which lets you set the compiler, either locally or globally, and lets you set compiler-specific options
- For supported compilers, Vim can parse their output and populate the QuickFix list automatically
- For unsupported compilers, it's relatively easy to add support - this actually shows off the power of the flexibility of the QuickFix list - for instance, note that in the supported compilers, PyUnit isn't actually compiled, but is still supported. As long as it outputs information in a consistent and includes the file and line number at a minimum, you can get it to work with Vim relatively easily

Configuring Other Tools

- Not comprehensive[1]
- Change compiler: `:compiler <name>`
  - (Selected) native support:[2]
    - GCC
    - PyUnit
    - TeX
  - Unsupported: Plugins / Hack your own![3]
- Change make: `makeprg`
  - Change what executes on `:make`
  - Set default flags, targets, etc.

[1] See `:help quickfix.txt`

[2] See `:help compiler-select` for more

[3] See `:help write-compiler-plugin`

Configuring Other Tools

- Not comprehensive[1]
- Change compiler: `:compiler <name>`
  - (Selected) native support:[2]
    - GCC
    - PyUnit
    - TeX
  - Unsupported: Plugins / Hack your own![3]
- Change make: `makeprg`
  - Change what executes on `:make`
  - Set default flags, targets, etc.

[1] See :help quickfix.txt
[2] See :help compiler-select for more
[3] See :help write-compiler-plugin

- You can also change the make program using the `makeprg` setting
- You can use this to change what's executed when `:make` is run, for instance if you're not using a makefile, or if you want to add default flags
- Ultimately, the QuickFix list gives you a lot of flexibility and control, and, when combined with configuration or plugins, presents a very powerful tool for working with many files and other programs

Reading / Writing Aids Overview

- Reading / Writing Aids
  - Find / Replace
  - File Navigation
  - Autocomplete / Code Navigation

Searching

- Basic searching (within a given buffer):
  - Plain search: `/` / `?`[1]
  - Word under cursor: `*` / `#`

[1] For info on Vim's regexes, see `:help 03.9`, `:help usr_27`, and `:help regex`

Searching

- Basic searching (within a given buffer):
  - Plain search: / / ?[1]
  - Word under cursor: * / #

[1] For info on Vim's regexes, see :help 03.9, :help usr_27, and :help regex

- I'll start with some basic things I've already discussed or you may already know. These aren't exactly "IDE-like" in the sense that you still have to do some work to get what you want, or basic, since some of them are fairly complex, but they'll lead you in the direction and can generally serve as a decent starting point or fallback.
- For instance, you can use / to search. But one you may not have heard of is ?, which is like / but goes backwards
- By the way, if you're unfamiliar with Vim's regex conventions, these help documents serve as a great starting point
- Additionally, * and # can perform searching for whatever word is under the cursor
- One problem is that these searches are localized entirely to your current buffer, and to expand them, you have to search to another buffer

Searching

- Basic searching (within a given buffer):
  - Plain search: `/` / `?`[1]
  - Word under cursor: `*` / `#`
- Global searching: `:vimgrep /pat/ files`
  - Recursive: `**`[2]
  - All files in arglist[3]: `##`
  - Any Vim filename[4]

[1] For info on Vim's regexes, see `:help 03.9`, `:help usr_27`, and `:help regex`

[2] See `:help wildcards`

[3] See `:help arglist`

[4] See `:help file-searching`, `:help cmdline-special`, and `:help filename-modifiers`

Searching

- Basic searching (within a given buffer):
  - Plain search: / / ?[1]
  - Word under cursor: * / #
- Global searching: `:vimgrep /pat/ files`
  - Recursive: **[2]
  - All files in arglist[3]: ##
  - Any Vim filename[4]

[1] For info on Vim's regexes, see :help 03.9, :help usr_27, and :help regex
[2] See :help wildcards
[3] See :help arglist
[4] See :help file-searching, :help cmdline-special, and :help filename-modifiers

- To solve that, there are some handy global search tools. Some of which I've already talked about. That's right - I still won't shut up about the QuickFix list!
- For instance, with Vimgrep, you can specify which files to include in your search
- This gives you can incredibly flexible yet powerful searching scheme
- By the way, I'll give some more examples of these in a bit, so don't worry if this doesn't make any sense right now
- Vim supports file globbing similar to bash's, including the "globstar" pattern for recursive looking.
- You can also use the special double pound sign (or double octothorpe if you're feeling swanky), which matches your entire arglist
- Finally, Vim has a pretty featureful filename modification syntax, which you can use here as well

Search Examples

- Search for "Linux" in all files in the current directory[1] ending in `.c` or `.h`:[2]

```
:vimgrep /\<Linux\>/g **/*.c **/*.h
```

- Search for "Lua" in all files relative to the current buffer's path:

```
:vimgrep /\<Lua\>/g %:h/**/*
```

- Search all open buffers:[3]

```
:bufdo vimgrepadd /pat/ %
```

[1] Vim has its own current directory; see `:help current-directory`

[2] `\<` and `\>` represent word boundaries; see `:help /\<`

[3] Assuming all buffers are named

Search Examples

- Search for "Linux" in all files in the current directory[1] ending in .c or .h:[2]
  `:vimgrep /\<Linux\>/g **/*.c **/*.h`
- Search for "Lua" in all files relative to the current buffer's path:
  `:vimgrep /\<Lua\>/g %:h/**/*`
- Search all open buffers:[3]
  `:bufdo vimgrepadd /pat/ %`

[1] Vim has its own current directory; see :help current-directory
[2] \< and \> represent word boundaries; see :help /\<
[3] Assuming all buffers are named

- So, let's get into some examples. This first one searches for the text "Linux" recursively for all files ending in .c and .h in Vim's current directory
- Remember that the double asterisks mean "0 or more directories"
- By the way, the funky bracket notation just is Vim's way of specifiying a "word boundary"
- Next, I'll show off using a path modifier by searching for the word "Lua" recursively in all files at or below the level of the current buffer's path
- `%:h` represents the "head" of the path, i.e. everything but the filename
- Finally, what if you want to search all open buffers? Here, the handy `:bufdo` command becomes useful
- This executes a command, in this case `:vimgrepadd`, on each each open buffer; in this context, the % represents the buffer name
- This simple version will give you errors if you have unnamed buffers

Replace

- Buffer: `:%s/pat/sub/g`[1]
- Beyond:
  - Populate QuickFix list
  - `:cdo %s/pat/sub/flags | update`
- External tools:
  - Buffer: `:%!sed s/pat/sub/flags`
  - Beyond: Varies

[1] See `:help :range {address}` and `:help :su`

Replace

- Buffer: `:%s/pat/sub/g`[1]
- Beyond:
  - Populate QuickFix list
  - `:cdo %s/pat/sub/flags | update`
- External tools:
  - Buffer: `:%!sed s/pat/sub/flags`
  - Beyond: Varies

[1] See :help :range {address} and :help :su

- As far as replacing goes, it's more of the same, really
- There's a special command, which you've likely seen as `:%s` before, for doing it in the current buffer
- Going beyond the current buffer, you have a few options:
- You can, once again, use the QuickFix list, this time with the `:cdo` command, which is like the `:bufdo` command from before, but works on items in the QuickFix list
- The `| update` writes those changes - you can leave this off if you just want to change the text but don't want to save yet.
- Finally, you can, of course, always you an external tool like sed if you'd like.
- Vim specifically lets you pass the text of a buffer to a command using the syntax shown, or you can just use the tool from the command-line however you like.

Jumping to Files

- `path`
  - List of paths to search
  - See path: `:set path`
  - Add: `:set path+=/path/to/dir/`
  - Check: `:checkpath`

- Under cursor:
  - Change buffer: `gf`
  - Open window: `CTRL-W_gf`

Jumping to Files

- path
  - List of paths to search
  - See path: `:set path`
  - Add: `:set path+=/path/to/dir/`
  - Check: `:checkpath`
- Under cursor:
  - Change buffer: `gf`
  - Open window: `CTRL-W_gf`

- Often when you're working on code, you'll want to open some other file. Vim offers a number of ways to do this.
- In addition to Vim's current directory, there's a `path` setting that can be modified to have Vim look at additional paths included, like libraries for instance.
- You can see the path and add to it like so; additionally, Vim has a built-in command that can look for include files and check their path as well
- Files added to your path are then used by other commands, like `gf`, which goes to the filename under the cursor
- Additionally, control w + f can be used to open the path under the cursor in a new window

File Explorer

- Netrw
- Launch: `e /path/to/explore/`
- Supports:
  - Compressed files
  - Remote files (ssh/ftp)[1]

[1] Only really good for quick edits in my opinion, though it does hide latency

File Explorer

- Netrw
- Launch: `e /path/to/explore/`
- Supports:
  - Compressed files
  - Remote files (ssh/ftp)[1]

[1] Only really good for quick edits in my opinion, though it does hide latency

- You might be surprised to learn that Vim has its own built-in keyboard-driven file explorer
- This file explorer, called `netrw`, can be invoked by trying to edit a directory, for instance, the current one.
- Even more impressively, it supports in-place editing for many compressed file types, like tar and zip, and even editing remote files via several different network protocols
- I will say that, while remote editing support is pretty handy, I often find it easier just to remote in to the specific machine I want to use instead
- The main advantages of netrw are convenience and the fact that, since you're editing a local copy of the file and just transferring it on save, you're less affected by network latency.
- The main downsides are that, even with keyless auth, it can still feel a bit slow and clunky to open new files, and especially to use netrw to explore files on the remote machine. Additionally, since you'll often want a remote terminal, you'll usually have to ssh in anyways.

Basic Completion

- "Insert completion"[1]
- Prefix: `CTRL-X`
  - `CTRL-F`: File names
  - `CTRL-N`: Keywords (local buffer)
  - `CTRL-I`: Keywords (included files)
  - ...
  - `CTRL-O`: "Omni" (Smart guess)
  - `CTRL-]`: Tag
- Cycle: `CTRL-N` / `CTRL-P`

[1] See `:help ins-completion`

Basic Completion

- "Insert completion"[1]
- Prefix: CTRL-X
  - CTRL-F: File names
  - CTRL-N: Keywords (local buffer)
  - CTRL-I: Keywords (included files)
  - ...
  - CTRL-O: "Omni" (Smart guess)
  - CTRL-]: Tag
- Cycle: CTRL-N / CTRL-P

[1] See :help ins-completion

- Vim provides pretty basic completion on its own, no external tools required
- While not the most powerful, these still can be useful, for instance if you're not working with tag information, and are still good enough to get the job done for simple cases
- Vim calls this "insert completion," and it can be accessed by using CTRL-X while in insert mode
- Vim offers a variety of insert completion options, including file names, keywords, and a "smart" mode
- The most important of these is "tag", which I'll discuss soon
- Once you've selected an options, you can use control N and P to cycle through the options you're given
- One of the keyword options is what you're most likely to want usually

Ctags

- Category of tools that produce "tag files" [14]
- **Tag files**: `tagname\t tagfile\t tagaddress`
  - `tagname`: Keyword
  - `tagfile`: File path
  - `tagaddress`: `ex` command (regex)
- Lacks context

Ctags

- Category of tools that produce "tag files" [14]
- **Tag files**: `tagname\t tagfile\t tagaddress`
  - `tagname`: Keyword
  - `tagfile`: File path
  - `tagaddress`: `ex` command (regex)
- Lacks context

- If you want something a bit better, though, you can try out Ctags
- Ctags really just the name for an overarching style of files, called "tag files," created by external tools.
- Universal Ctags is the most up-to-date version, and contains support for many languages
- These tools produce fairly simplistic files, which contain many, often sorted, lines with the following structure:
- A tag name, which is the keyword you can use as lookup
- Next is a literal tab character, then the filename to look for
- Finally, after another tab is what's called the "address," which is an ex command to locate the text; usually this takes the form of a regex
- While these are useful, and much better for insertion completion, they still leave a lot to be desired
- By the way, why do you think a regex is used instead of line numbers? It's so that the tag file can still be useful after minor changes.

Navigating Tags

- Jump to definition:
  - First: `:tag <name>`[1] / `CTRL-]`
  - Pick: `:tselect <name>` / `g]`
- Tag stack:[2][3]
  - Following a tag pushes it to a stack
  - Pop a tag and jump back: `:pop` / `CTRL-T`
- "Preview": `:ptag <name>` / `CTRL-}`
- More[4]

[1] Name can be a regex; see `:help tag-regexp`

[2] See `:help tagstack`

[3] See also: the jump stack; `:help jump-motions`

[4] See `usr_29` and `:help tagsrch.txt`

Navigating Tags

- Jump to definition:
  - First: `:tag <name>`[1] / `CTRL-]`
  - Pick: `:tselect <name>` / `g]`
- Tag stack:[2][3]
  - Following a tag pushes it to a stack
  - Pop a tag and jump back: `:pop` / `CTRL-T`
- "Preview": `:ptag <name>` / `CTRL-}`
- More[4]

[1] Name can be a regex; see `:help tag-regexp`
[2] See `:help tagstack`
[3] See also: the jump stack; `:help jump-motions`
[4] See usr_29 and `:help tagsrch.txt`

- Using these tags, you can navigate through your code a bit more easily
- Specifically, you can jump to a matching symbol's definition with the `:tag name` command, or the keyword under the cursor with `CTRL-]`
- If you have many matches, you can see a list of them with `:tselect` or `g]` and pick from that list
- One useful concept is the tag stack, which keeps track of what tags you've jumped to, and where you've jumped from
- When you follow a tag, your previous location gets pushed to the stack; by using `:pop` and `CTRL-T`, you can pop that value from the stack and go back to that location
- One final thing I find useful that I'll discuss is the preview window, which opens the location of a tag, but then moves your cursor back to the previous window
- Despite their simplicity, you can do a lot with tags - I recommend reading the help file for more of what you can do with them

Cscope

- Still lacks context
- Initiailze: [11]
  - Generate: `cscope -b -R`
  - Tell Vim: `:cscope add cscope.out`
- Main features:
  - Find symbol: `:cs find s <symbol>`
  - Find definition: `:cs find g <symbol>`
  - Find child funcs: `:cs find d <symbol>`
  - More...[1]
- Populate QuickFix: `:set cscopequickfix`

[1] See `:help cscope`

Cscope

- Still lacks context
- Initiailze: [11]
  - Generate: `cscope -b -R`
  - Tell Vim: `:cscope add cscope.out`
- Main features:
  - Find symbol: `:cs find s <symbol>`
  - Find definition: `:cs find g <symbol>`
  - Find child funcs: `:cs find d <symbol>`
  - More...[1]
- Populate QuickFix: `:set cscopequickfix`

[1] See `:help cscope`

- Cscope is still limited in the same ways as ctags, namely that it lacks context, so it can't be "smart," but it is more powerful than plain Ctags
- You can initialize it using the external `cscope` program
- Next, you tell Vim about it with `:cscope add`
- Once you've done that, you can use a variety of builtin `find` functions with different prefixes to get whatever information you've specified
- Additionally, you can get it to populate the QuickFix list as well, using the `cscopequickfix` setting

Alternatives

- GNU Global [13]
  - Has Cscope mode → Vim compatible
  - Also has Vim plugins
  - Supports 30+ languages
  - Incremental builds
- Eclim [12]
  - Eclipse functionality
  - Requires a headless Eclipse client
- Many more [16, "Tagging systems"]

Alternatives

- GNU Global [13]
  - Has Cscope mode → Vim compatible
  - Also has Vim plugins
  - Supports 30+ languages
  - Incremental builds
- Eclim [12]
  - Eclipse functionality
  - Requires a headless Eclipse client
- Many more [16, "Tagging systems"]

- Cscope and Ctags aren't the only things tools to use, though. I haven't tried any of these yet, but they could be good for your use cases.
- GNU Global is one of the most stable alternatives; it looks promising and should be natively compatible with Vim
- For all you Eclipse fans out there, you can try out Eclim, which has Vim support and works by running a headless version of Eclpise
- There are tons of tagging systems to choose from

Language Servers

- Pre-Vim 8+/Neovim:
  - Linting: Syntastic [6]
  - Completion: YouCompleteMe [10]
- Absolutely tons of plugins [15] [18]
- Most popular: **CoC** [1]
  - **Praise**: "Just works," "most complete"
  - **Criticism**: "Bloated," "Own ecosystem"

Language Servers

- Pre-Vim 8+/Neovim:
  - Linting: Syntastic [6]
  - Completion: YouCompleteMe [10]
- Absolutely tons of plugins [15] [18]
- Most popular: **CoC** [1]
  - **Praise**: "Just works," "most complete"
  - **Criticism**: "Bloated," "Own ecosystem"

- Finally, we get to the most powerful aid of them all - language servers, often called "LSPs" due to the protocol they use
- LSPs do an absolute ton of things, from linting to semantic analysis. No one plugin (that I'm aware of) covered all this functionality pre-Vim 8, but there were alternatives you could use for each one that I've listed
- Frankly, there's almost too many LSP plugins for me too choose from (at the moment, there are at least 5 with significant user bases), and definitely far too many to cover here, not to mention the fact that the server you choose is just as important, if not more so, than the frontend
- Frankly, I don't have enough experience with these to even discuss them, but they look really cool and are the next thing I plan on looking in to when I get the time.
- CoC is the most popular plugin from what I've seen, and it's so popular because of how complete it is, while also apparently being the most easy to set up. Some criticize it for being NodeJS based and trying to reinvent too much of what Vim already does

Miscellaneous Overview

- Miscellaneous
  - Interactive Debugging
  - Per-Project Config
  - Persistence
  - Plugins

Termdebug

- In-Vim Debugging:
  - Pre-Vim 8: Plugins [2]
  - Vim 8+: `:Termdebug`[1]
- GDB interface
- Usage:
  - Initialize: `:packadd termdebug`
  - Start: `:Termdebug <executable>`
- Windows
  - **Code**: Shows current line, breakpoints, etc.
  - *Assembly*: (Optional) Assembly view
  - **Program**: Displays program output
  - **GDB interface**: Standard GDB terminal

[1] See `:help terminal-debug`

Termdebug

- In-Vim Debugging:
  - Pre-Vim 8: Plugins [2]
  - Vim 8+: `:Termdebug`[1]
- GDB interface
- Usage:
  - Initialize: `:packadd termdebug`
  - Start: `:Termdebug <executable>`
- Windows
  - **Code**: Shows current line, breakpoints, etc.
  - *Assembly*: (Optional) Assembly view
  - **Program**: Displays program output
  - **GDB interface**: Standard GDB terminal

[1] See `:help terminal-debug`

- Prior to Vim 8, Vim lacked native built-in debugger functionality, so you had to either use external tools or plugins to debug
- Since Vim 8, you have Vim's "Termdebug" plugin
- Like much of the rest of Vim's builtin tooling, this is primarily a C/C++ tool, as it acts as a wrapper around GDB. Though of course, any languages supported by GDB can be used with it.
- Usage is pretty intuitive if you're familiar with GDB already: once it's initialized, you launch it by running `:Termdebug` followed by the program name
- This command creates 3 windows by default: a code window at the top, a stdout window beneath that, and a GDB window at the bottom
- While you can just treat it like GDB inside Vim, Vim has special commands available to make things easier

Other Debuggers

- **GDB-compatible**: `g:termdebugger`
- **Debug Adapter Protocol**: LSP for Debugging
  - Vimspector [9]
- Other plugins: [17]
- External tools: `:terminal` (Vim 8+)

Other Debuggers

- **GDB-compatible**: `g:termdebugger`
- **Debug Adapter Protocol**: LSP for Debugging
  - Vimspector [9]
- Other plugins: [17]
- External tools: `:terminal` (Vim 8+)

- If you want to use a GDB-compatible debugger, you can using the `g:debugger` setting
- Unfortunately, Vim currently only supports GDB-compatible debuggers; what happens if you want to use something else?
- Thanks to same new features in Vim, it is possible to do, and has been done, but it's definitely not as trivial as adding support for a new compiler
- That being said, in the same way that LSPs are changing linting and other writing functionalities, a new idea, called DAPs, or Debugging Adapter Protocol, is also being developed
- The largest DAP plugin I'm aware of for Vim is Vimspector, but I haven't looked into this area a lot, as GDB meets all my needs
- If you can't or don't want to use a DAP, there are plenty of other Vim debugging plugins available, though
- Finally, if a command-line tool exists, you can use, simply via command line. Since Vim 8, you can open a terminal directly in Vim and run the command that way if you like, or you can use your favorite window/display manager instead

Configuring Individual Projects

- Enable: `:set exrc`[1]
- Recommended: `:set secure`
- Even more recommended: Project-specific autocommands[2]

[1] See `:help 'exrc'`

[2] See `:help trojan-horse`

- Enable: `:set exrc`[1]
- Recommended: `:set secure`
- Even more recommended: Project-specific autocommands[2]

[1] See `:help 'exrc'`
[2] See `:help trojan-horse`

- Often you'll want to have project-specific settings that you'll want to configure; for instance, a project may enforce tabs instead of spaces, or you may want to use a specific compiler
- Changing these settings every time would be awfully tedious. Luckily, Vim has settings that let you do just that
- Vim has an option, called `exrc`, that, if set, will look for a `.vimrc` file and, if found, execute those commands
- For security reasons, it's also recommended to set the secure option as well
- Additionally, the Vim documentation says that, really, you should use autocommands for each directory you want to configure, since this will be much more secure

Sessions

- Save workspace: `:mksession`
- Reload workspace:
  - `vim -S Session.vim`
  - `:source Session.vim`

[1] See `:help swap-file`

Sessions

- Save workspace: `:mksession`
- Reload workspace:
  - `vim -S Session.vim`
  - `:source Session.vim`

[1] See :help swap-file

- Often when you're working, you'll have to pause, and maybe even shut down your computer you're working on
- Many people like to be able to jump right back into things to get back to work
- That's where Vim's sessions come in. Vim's sessions can essentially be thought of as a "saved checkpoint" of where you left off (for the most part - more on that soon)
- You can create a session with `:mksession`, then when you launch Vim, you can just run `vim -S Session.vim`, or use the `:source` command, to restore that session

Sessions

- Save workspace: `:mksession`
- Reload workspace:
  - `vim -S Session.vim`
  - `:source Session.vim`
- Gotchas:
  - Terminal
  - Unsaved changes[1]

[1] See `:help swap-file`

Sessions

- Save workspace: `:mksession`
- Reload workspace:
  - `vim -S Session.vim`
  - `:source Session.vim`
- Gotchas:
  - Terminal
  - Unsaved changes[1]

[1] See :help swap-file

- This does overall a very good job, though it does have some potential "gotchas"
- The main ones have to do with the terminal features - namely, the contents of the terminal are not saved, and termdebug restoration gets a bit messed up
- Unsaved changes also aren't saved, nor are they reapplied at the start of a new session. This means you'll have to save all your work, or use swap files

Undo Persistence

- Undo tree[1]
- Enable: `:set undofile`
- Increase/decrease size: `undolevels`

[1] See `:help undo-tree`

Undo Persistence

- Undo tree[1]
- Enable: `:set undofile`
- Increase/decrease size: `undolevels`

[1] See `:help undo-tree`

- You may be aware of Vim's powerful tree structure for storing changes
- But what you may not know is that you can have these persist even after reboot using `undofiles`
- Note that you can still "lose" changes, especially with undo persistence, as Vim limits how large the undo tree can become. If you'd like to avoid this, you can increase the `undolevels` setting

Plugins

- Git:
  - Fugitive [3]
  - gitgutter [4]
- Fuzzy finders
- More

Plugins

- Git:
  - Fugitive [3]
  - gitgutter [4]
- Fuzzy finders
- More

- Finally, I'll close with this: because of Vim's popularity, unless you're looking for some particularly advanced functionality, there's a very good chance what you're looking for exists as a plugin that can be used
- For instance, many people want in-editor Git control and information; fugitive and gitgutter, are two programs that handle each of those, respectively.
- There are also tons of plugins to add fuzzy finding, as well as many other features. Git was just the main one I wanted to talk about, since it's a commonly requested IDE feature

References I

Plugins

[1] **CoC** (Node-based VSCode-like plugin ecosystem for Vim)
https://github.com/neoclide/coc.nvim

[2] **Conque-GDB** (GDB CLI and terminal emulator for Vim)
https://github.com/vim-scripts/Conque-GDB

[3] **Fugitive** (A Git-wrapper for Vim)
https://github.com/tpope/vim-fugitive

[4] **gitgutter** (Show git difs in the column)
https://github.com/airblade/vim-gitgutter

[5] **quickfix-reflector** (Edit text in the quickfix window)
https://github.com/stefandtw/quickfix-reflector.vim

[6] **Syntastic** (Vim syntax checker)
https://github.com/vim-syntastic/syntastic

[7] **vim-editqf** (Make quickfix entries editable)
https://github.com/jceb/vim-editqf

[8] **vim-qfedit** (Edit the quickfix list freely)
https://github.com/itchyny/vim-qfedit

[9] **vimspector** (A multi-language debugging system for Vim)
https://github.com/puremourning/vimspector

[10] **YouCompleteMe** (Vim completion engine)
https://github.com/ycm-core/YouCompleteMe

References II

Software

[11] **Cscope** (Source code viewing tool)
http://cscope.sourceforge.net/

[12] **Eclim** (Eclipse for Vim)
http://eclim.org/index.html

[13] **GNU Global** (Source code tagging system; compatible with Cscope)
https://www.gnu.org/software/global/

[14] **Universal Ctags** (A modern Ctags implementation)
https://ctags.io/

Websites

[15] **Langserver.org** (Collection of LSP resources)
https://langserver.org/

[16] **'Source Code Reading' related sites** (GNU Global Links)
https://www.gnu.org/software/global/links.html

[17] **Vim Awesome - Debug** (List of Vim debug plugins)
https://vimawesome.com/?q=debug

[18] **Vim Awesome - LSP** (List of Vim LSP plugins)
https://vimawesome.com/?q=lsp