Vim

Making text editing fun and efficient

Alec Gibson

BlueCat Networks

agibson@bluecatnetworks.com

November 24, 2020

Presentation Details

- This presentation was created in Neovim 0.5.0 using the Beamer Latex package
- The source code is available in a repo on my GitHub
- For discussions about Vim, please post in the company #vim-geeks slack channel!

Disclaimer

The target audience of this talk is not seasoned users of Vi-family editors (though you are more than welcome to stay if this describes you). Many of the features I will refer to as Vim features were Vi features first. I never used Vi, and clarifying when features were introduced in Vim's lineage is outside the scope of this talk. So for the purpose of this talk they are Vim features.

Furthermore, almost everything I say in this talk applies equally to Neovim as it does to Vim. Neovim is a fork of Vim with slightly tweaked default options, no Benevolent Dictator For Life, and which tends to implement new features more quickly. I personally use Neovim instead of Vim, but both operate extremely similarly.

Goal

After learning Vim's basics in university, I decided to start using it full-time when I started working at BlueCat at the beginning of May. At first, my workflow was quite inefficient — all my knowledge came from vimtutor, and several Reddit posts I had read. Since then, my knowledge of Vim has expanded massively, and my editing has become much more fluent, so I wanted to share what I've learned with other interested developers.

This talk is meant to make it easier for other developers to make the same transition as me, from a standard point-and-click text editor or IDE to Vim. I try to cover many of Vim's core features, so that you have some idea what it is capable of (which is a considerable amount) instead of immediately reaching for plugins. Where possible, I try to mention what features I have found particularly useful, and to outline why I feel learning Vim has been a net positive to my work as a developer. No section of this talk is exhaustive - Vim's feature set is so huge that covering any topic exhaustively would take forever and be very boring. Instead I try to cover the most important stuff, and reference documentation so you can learn the rest at your leisure.

Overview

About Vim

What is Vim

According to vim.org

Vim is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as “vi” with most UNIX systems and with Apple OS X.

Vim is rock stable and is continuously being developed to become even better. Among its features are:

- persistent, multi-level undo tree
- extensive plugin system
- support for hundreds of programming languages and file formats
- powerful search and replace
- integrates with many tools

Here are some important features I think were missed:

- Vim is very lightweight, meaning it runs smoothly on any modern computer and performs well over SSH.
- Vim's startup time is nearly instantaneous.
- Because Vim runs in a terminal it works nicely with other terminal utilities (like tmux), and you can pipe the output of scripts directly into Vim.
- Vim's configuration is scriptable, so you can define custom functions then use them in commands and keybindings.
- Once you learn Vim you can use it everywhere - its keybindings are supported in most other editors either natively or through plugins (including VSCode, Emacs, and IntelliJ to name a few)

And most importantly....

If you have not used them before, learning Vim's keybindings will provide you with an extremely efficient way to edit text files.

Who Should Try Vim?

You may appreciate Vim if you:

- Spend a large amount of your day editing plain text files (common in software development and IT)
- Make frequent use of your terminal emulator
- Appreciate the value of keyboard shortcuts
- Like to customize your tools to suit your workflow
- Want to make an investment in learning a single editor which works for every programming language
- **And especially** if you need a way to automate performing boring, repetitive edits to file

A Minimal Vim Workflow

Modal Editing

Modal Editing

Vim is a modal text editor, meaning keypresses in Vim perform different actions depending upon the editor's current "mode".

When you edit a file in Vim, the editor starts in "normal" mode. This can confuse new users, because Vim's normal mode treats every key on the keyboard as a binding for a shortcut. To start off learning Vim, let's look at the smallest possible set of features you need to edit files.

Basic Vim File Operations

- To open a file in Vim, type `vim {filename}` in your terminal emulator.
- To change the current open file, type `:e /path/to/file<CR>` (including the colon at the start).
- To save the current file, type `:w<CR>`.
- Finally, to exit Vim type `:q<CR>`.

What's That CR Symbol?

`<CR>` is how you represent the Enter/Return key in Vim keybindings.

Popular Variants

Popular variants of these commands include `:wq<CR>` to save and quit, and `:wq!<CR>` to force Vim to save and quit (ignoring any warnings while doing so).

Basic Vim Editing

The simplest (though probably the slowest) way to navigate a file in Vim is using the arrow keys. To make changes in the current file, press `i` to enter "insert" mode. In insert mode, keys behave the way they would in any other text editor - letter and number keys insert their corresponding characters, and `<BS>` (backspace) deletes the previous character.

If you want to run any of the file operations from the previous slide, just press `<ESC>` (escape) to return to normal mode first.

Using Your Mouse in Vim!?

It's true, Vim supports using your mouse. Just set the required option by typing `:set mouse=a` in normal mode.

This is enough Vim to edit config files on servers over SSH (albeit slowly), or to use while setting up a minimal Linux installation on a PC.

However, if this was the most efficient way to edit files in Vim, **the editor would have died out long ago**.

Vim Can Do More

Why The Minimal Workflow Isn't Enough

The minimal Vim workflow I described in the previous section is seriously inefficient. Here are a few reasons why:

- Using the mouse requires you to take your right hand off the keyboard
- Using the arrow keys requires you to take your right hand off the home row
- You can only move one character at a time using the arrow keys
- You can only delete one character at a time using <BS>
- If your cursor is in the middle of some text, you have to move to the end to delete it
- We don't have any way to copy and paste
- We don't have a way to undo mistakes
- This workflow doesn't include any way to search the current file
- Repeated edits need to be executed manually each time

Vim's normal mode has features which solve all these issues.

Movement Keybindings

Problem

- Using the mouse requires you to take your right hand off the keyboard
- Using the arrow keys requires you to take your right hand off the home row

Vim solves these issues by using the keys `h, j, k, and l` as alternatives to the arrow keys while in normal mode. These behave in the following manner:

h Move left one character

j Move down one character

k Move up one character

l Move right one character

These keybindings are well-known enough that other programs which also use hjkl for navigation often refer to them as "Vim keys".

Moving Longer Distances

Problem

- You can only move one character at a time using the arrow keys

Vim's normal mode contains many keybindings for moving more than one character at a time. Here are some I use frequently:

- `w`/`b`: Move forward/back by one word at a time
- `0`/`$`: Move to the start/end of the current line (use `^` instead of 0 to move to the first non-whitespace character of the line)
- `<C-d>`/`<C-u>`: Move down/up by half a screen at a time
- `gg`/`G`: Move to the start/end of the current file

Useful Keybindings: f and F

Typing f{char} searches the current line for the chosen character, and using F searches backwards. Pressing ";" jumps to the next occurrence, and pressing "," goes to the previous occurence (very useful if you overshoot your target). This is often the fastest way to jump to a specific location on the current line.

Repeating Movements Multiple Times

Most Vim commands allow you to prepend a number to multiply their effects. For example:

- typing `5j` in normal mode moves your cursor down 5 lines.
- typing `3w` moves your cursor forward 3 words
- typing `10fe` moves your cursor to the tenth occurence of the character “e” following it on the current line

Keybindings for Editing

Vim has a number of keybindings for editing the current file's contents. We can start with keybindings which operate on single characters:

- `x`: delete the character under the cursor
- `r{char}`: replace the character under the cursor with {char}
- `~`: swap the case of the character under the cursor
- `<C-a>`/`<C-x>`: increment / decrement the number under the cursor

Operating on Chunks of Text

Problem

- You can only delete one character at a time using <BS>

Operating on chunks of text is a place where Vim's keybindings shine. Vim separates these edits into keybindings it calls "operators" and "motions".

Operators

Operators are the verbs of the edit - they are things like "delete", "change", "yank" (Vim language for "copy"), "indent", "format", etc.

Motions

Motions are like nouns - they define what the operator should operate on. Examples of motions would be "a word", "until the end of the line", or "to the end of the file".

Operators:

- `d`: delete
- `c`: change (delete, then enter insert mode)
- `y`: yank (copy)
- `<`/`>`: increase / decrease indent
- `=`: format (works well for C-style languages, but is configurable)

Example Edits:

- `dl`: delete to the right (same as x)
- `dh`: delete to the left
- `dw`: delete word
- `c$`: change to the end of the line
- `yG`: yank to the end of the file
- `dt_`: delete to the next underscore $*$

$*$ Note: `t` and `T` act like `f` and `F`, but they stop one character before the character being searched for. These are extremely useful for operating on all the text before a certain target character. `Snake_case`, `kebab-case` and `camelCase` are used frequently in source code files, so it often works well to make your target character the next underscore, hypen, or a specific capital letter.

Shortcuts for Common Operations

There are also some special editing keybindings to make frequently required editing operations more convenient. Repeating the operator twice means to apply it to the current line (`dd` deletes the current line, `yy` yanks the current line, etc.). There are also shortcuts for when you capitalize the operators - `D` deletes to the end of the line, `Y` is equivalent to `yy`, etc.

For most motions, the easiest way to understand the behaviour of Vim edits is to imagine the behaviour of the cursor if you typed the motion without the operator, then the operator will be applied to the characters spanned by that motion. This is why, for example, `dw` doesn't delete the entire word the cursor is currently inside, but rather deletes from the current cursor position to the beginning of the next word.

Sometimes this is not the behaviour we want - sometimes we want to delete the entire word the cursor is currently inside. We will explore how to do this using "text objects" next.

Text Objects

Problem

- If your cursor is in the middle of some text, you have to move to the end to delete it

Up until now, the motions we have applied to operators have also been normal mode navigation keybindings. This is not always the case. After typing the operator but before the motion, Vim is in a special mode called "operator-pending" mode, and this mode has some of its own unique keybindings. Text objects are an example of keybindings for operator-pending mode. These keybindings let operators work on an area of text the cursor is already inside - for example the current word or the surrounding set of quotation marks.

examples of text objects:

- `iw/aw`: inner / around word
- `is/as`: inner / around sentence
- `ip/ap`: inner / around paragraph
- `i"/a"`: inner / around quoted string
- `i(/a(`: inner / around `()` block
- `i[/a[`: inner / around `[]` block
- `i{/a{`: inner / around `{}` block

Language plugins often add text objects, for example to operate on the contents of classes and functions.

Each of the “inner” variants of the provided text objects ignore whitespace and surrounding delimiters. The “around” variants include the surrounding delimiters and whitespace.

For example, `di"` will delete the contents of the quotation marks the cursor is currently inside, but will not touch the quotation marks themselves. In contrast, `da"` will delete the contents of the quotation marks, the quotation marks, and surrounding whitespace.

To frame this differently, if you had a series of quoted strings separated by spaces, `di"` would delete the contents of the current string, but repeating it again would do nothing. In contrast, `da"` would delete the current quoted string so that the cursor ends on the next quoted string, so if you repeated the edit it would delete each quoted string in turn.

Pasting, or "Putting"

Problem

- We don't have any way to copy and paste

We've seen how to yank text already using the `y` operator. Now all you need to do to "put" it (Vim language for pasting) is to press `p`.

It is important to know, deleting in Vim does not just delete the chosen text, but behaves more like the "cut" operation in other popular editors. That is to say, when you use the `d` operator, the deleted text will be the next thing "put" when the user presses `p`.

Searching

Problem

- This workflow doesn't include any way to search the current file

Searching is very quick in Vim, and is often the fastest way to navigate a file. Press `/` to initiate a search, then type in the pattern to search for (patterns are regular expressions so, among other things, they can include wildcards) and press `<CR>` to start the search. To move to the next search result press `n` and to move to the previous result press `N`. Here are a few other useful keybindings for searching:

- `?`: start a search in the opposite direction
- `q/`: view a history of previous search patterns (press `<CR>` on one to search for it in the current file)
- `/<CR>`: search again using the most recently used search pattern

Undo and Redo

Problem

- We don't have a way to undo mistakes

You can undo the most recent edit by pressing `u`, and you can press `u` multiple times to continue undoing edits. An example of a single "edit" in Vim would be a single operator + motion combination. Everything you type from pressing `i` to enter insert mode until returning to normal mode is considered a single edit, so pressing `u` once will undo all of it.

You can press `<C-r>` to "redo" the most recently undone edit. Like with `u`, you can repeatedly press `<C-r>` to redo more undone edits.

If you accidentally undo too much then edit the document, it is still possible to recover - Vim keeps track of your branching undos. We'll discuss this later in the Vim Power Tools section when we discuss the Undo Tree.

Repeating Edits

Problem

- Repeated edits need to be executed manually each time

We discussed on the previous slide what a single "edit" is in Vim. You can repeat the most recent edit by pressing the `.` key.

Effectively using `.` takes a lot of practice, and requires that you plan your edits to be repeatable. For example, if you run `diw` to delete a word, then `i` to enter insert mode and type its replacement, subsequent presses of `.` will perform the insertion but not the deletion. If you want the deletion to be included in the repeated edit, you could use `ciw` instead.

Vim supports many ways to repeat edits. I'll discuss macros, `:substitute` and `:global` in the "Power Tools" section of this talk. All of these are more flexible than the `.` key.

How to Find Help

Vim comes with extensive documentation built-in, accessible using the `:h` command (short for `:help`, which works too if you want to be verbose). If you want to look up what a particular keybinding does, you can run `:h {key}<CR>` to look up the corresponding help file. If you don't know the name of the help file you are looking for, you can run `:helpgrep {phrase}<CR>` to search all Vim's help files for a particular phrase. Helpgrep fills Vim's "quickfix" list with its results - we will discuss the quickfix list more later, but for now just know you can run `:cnext<CR>` to go to the next result and `:cprev<CR>` to go to the previous result.

How to Learn Vim

If you want more thorough materials to learn Vim, you can start by running `vimtutor` in your terminal (it usually comes with your installation of Vim). After completing vimtutor, you can read Vim's user manual for a lengthy but readable guide to all of Vim's most important features. To read the user manual, run `:h user-manual<CR>` in Vim.

Power Tools

Now that we've covered most of the basic features of Vim, you have seen one of the reasons many developers swear by it as their editor of choice - the ability to modularly compose edits using operators and navigation keybindings gives you a very terse language for quickly editing documents. However, there are probably still some people wondering **why would anyone want to use this editor**? Hopefully the features I discuss in this section will help make a stronger case in favour of Vim.

To start with, I'll cover Command-Line Mode and Visual Mode to round out our understanding of Vim's modes. Then we'll get into some of (what I consider) Vim's killer features - the jump list, registers, macros, `:substitute`, `:global`, and the undo tree. By the end of this section I'll have explained enough of Vim's features that, if you became fluent in their usage, you could edit single files in Vim quite efficiently.

After this section, we'll cover ways to manage editing multiple files in a single Vim session, then we'll finish up with a short introduction to Vim configuration.

Command-Line Mode

Many of Vim's features are accessible using keybindings in normal mode. However, we can also access features by writing out the names of commands to execute using Vim's command-line mode. Press `:` to enter command-line mode, then you can type in the name of a command and hit `<CR>` to execute it and return to normal mode. If you change your mind, you can press `<ESC>` to go back to normal mode without executing a command.

We've Seen This Before

We have already learned several Vim command-line mode commands. The file operations we learned (`:w`, `:q` and `:e`) all execute in command-line mode, as do the `:h` and `:helpgrep` commands. Vim doesn't require you to type the whole name of a command to execute it - you just need to type enough to disambiguate it from other commands. The file operations we learned are actually short forms for `:write`, `:quit` and `:edit`.

How Do I See My Command-Line History?

When you are in command-line mode, you can press the up and down arrow keys (or <C-p> and <C-n>, emacs style) to select previously run commands. You can view your command-line history by typing `q:` in normal mode. This will open a window showing past commands you've run, where you can edit a previous command, then execute the edited version using <CR>.

Visual Mode

One of the disadvantages of Vim's operator/motion syntax for defining an edit, is that you need to decide on the appropriate motion for an edit on the spot after already commiting to an operator. If you would prefer to make your selection of text to operate on first, then specify the operator after, Vim contains three different "visual modes" which allow you to do just that.

Character-Wise Visual Mode

The simplest visual mode Vim offers is character-wise visual mode, often just refered to as visual mode. You can enter this visual mode by pressing `v`, and exit to normal mode using <ESC>. While in any visual mode, your current text selection will be highlighted. You can move the cursor to change where the visual selection ends, or press `o` to instead change where the selection starts. After making your selection you can use operators on the selected text, like `d` to delete it, or `y` to yank it.

Line-Wise Visual Mode

You can press `V` (capital V) to enter line-wise visual mode - a variant of visual mode which only selects entire lines. I often use this version of visual mode to delete or yank data which is formatted into separate lines, such as a function, a JSON object, or a chunk of YAML. It is also useful for constraining the scope of some command-line commands, something we'll explore more when we discuss the `:substitute` command.

Block-Wise Visual Mode

You can press `<C-v>` to enter block-wise visual mode, which lets you select a rectangular region of text. I have found this most useful when I want to add, remove or modify a prefix for several lines. For example, if I wanted to create a bulleted list, I would use block-wise visual mode to select the first character of several lines, press `I` to insert at the start of each line's selection in the block (a special block-wise visual mode mapping), type `-`, then press `<ESC>`. Upon returning to normal mode, all my selected lines would be updated to begin with a hyphen.

The Jump List

This is a simple feature, but it is **constantly** useful, and I wish every text editor had it. Vim keeps track of every time you use a motion to move a significant distance, and stores these locations you've jumped to in its "jump list". You can view this jump list by running the `:jumps` command.

The useful part is that Vim lets you jump back to older locations you've been using `<C-o>` and newer locations using `<C-i>`. This means for example, if you created a tags file using ctags, you could jump to a function's definition using `<C-]>`. Then, using the jump list you could use `<C-o>` to jump back to where the function is called when you're done looking at it. I use this functionality constantly when reading new code I haven't seen before.

Note: the `<C-]>` keybinding also lets you jump to referenced help pages inside `:help`.

Registers

Instead of using the system clipboard, Vim uses a large collection of "registers" to store text for use elsewhere. Many operators can be prefixed with a register name, to make them use that register. To tell Vim you are specifying a register, you start with " - a quotation mark - followed by a single key for the register name. For example:

- `"ayy`: yank the current line, and put it into register `a`
- `"add`: delete the current line, and put it into register `a`
- `"ap`: insert the contents of register `a` at the cursor's location

You can also use a capital letter to tell Vim to append to a register instead of replacing the register's contents. Many of the non-character keys on the keyboard are also used for registers with special purposes (" is the default register, _ is no register, + can be configured to use the system clipboard, etc). You can learn more about these by reading `:help registers`.

Macros

Vim allows you to execute arbitrary strings of characters stored in a register as a macro! For an easy way to record a macro, Vim provides a convenient keybinding - `q` - which allows you record a sequence of keypresses into a register. For example:

- `qadwq`: record the characters `dw` into register `a`
- `qAdwq`: record the characters `dw` into register `a`, appending to the current contents of the register

Once you have recorded a macro into a register, you can the execute the macro using the `@` keybinding, followed by the name of the register to execute:

- `@a`: execute the contents of register `a` as a macro
- `@@`: execute the last executed macro again

Repeating Macros

Like most of Vim's commands, you can prefix the execution of a macro with a number to specify the number of times to repeat that macro. If Vim reaches the end of the current file while executing a macro, it stops executing it (this is useful, because you can include a search in your macro to find the next place to execute it).

Because macros just execute the contents of a register, there are many ways to record a macro - you don't have to use q! If you want, you could write the macro in your current file, then yank it into a register to execute it. Or you could record part of your macro in register a, stop recording, then append to that same register using `qA`. Vim's macro system is extremely flexible, and is a good way for repeating edits which are too complex to use the `.` key.

:substitute

One of the more common features people use in a text editor is the ability to search for and replace a sequence of characters. To do this, Vim provides the `:substitute` command which has the following syntax:

:substitute syntax

```
:[range]s[ubstitute]/{pattern}/{replacement}/[flags]
```

This syntax provides a great deal of flexibility, allowing you to match strings using a regular expression, use matched subexpressions in the pattern's replacement, and limit what lines to perform the substitution on. But before we get into these more advanced (but very much worth learning) features, we'll explore some examples of basic `:substitute` usage.

:substitute examples

- `:s/asdf/qwerty`: replace the first occurence of `asdf` with `qwerty` on the current line
- `:s/asdf/qwerty/g`: replace every occurence of `asdf` with `qwerty` on the current line (the `g` flag means replace every occurence instead of just the first one)
- `:1,10s/asdf/qwerty/g`: replace every occurence of `asdf` with `qwerty` on lines 1 to 10 of the current file
- `:%s/asdf/qwerty/g`: replace every occurence of `asdf` with `qwerty` on every line in the current file (Vim uses the range `%` as a synonym for `1,$`, meaning every line in the current file)

Patterns

One of the best things about Vim's `:substitute` command is that it doesn't just match verbatim strings of characters, but instead uses a full system of regular expressions. Obviously, outlining all the regular expression rules would be too much for this presentation, but I'll list some of the features I use most often (this assumes the "magic" option is set, which effects what characters need to be escaped to take on a special meaning):

- `.`: any character
- `*`: any number of repetitions
- `^`: the start of a line
- `$`: the end of a line
- `\(\)`: group contents into a subexpression (this subexpression can be used with `*`, and can be referenced in the pattern's replacement)

There is much more that you can do with Vim's regular expression system. To learn more, read `:help pattern`.

Using Matched Patterns in the Replacement

Often you want to perform a substitution, but provide some context so that you avoid the pattern matching false-positives. You can do this using subexpressions in your `:substitute` command's pattern. After using a pattern containing subexpressions, you can reference them in the replacement using `\0` to mean the whole matched pattern, and `\{num}` to mean the `{num}`th subexpression. Here is an example which hopefully illustrates how useful this can be:

```
:%s/func \(.*\)(/func (s *MyStruct) \1(
```

This command's pattern matches "func (", followed by an arbitrary string of characters (captured in a subexpression), followed by an opening bracket — this is the pattern followed by a function definition in Golang. The replacement takes the function name, and replaces its prefix with "func (s *MyStruct) (". This substitute command could be used to turn all the functions in a file into methods for a struct you defined!

Restricting a Substitution's Range

Sometimes I find that I want to perform a substitution only in a select portion of a file. For example, I might want to rename a variable inside a single function, but not touch variables of the same name in other functions. I have already shown that you can restrict `:substitute` by providing `{start},{end}` as the range. I never do this because Vim works so nicely with line-wise visual mode for setting ranges. Just select the lines you want to use as the range for your `:substitute`, then press `:` and Vim will automatically insert `'<,'>` as the range. This range means the current visually selected lines, and it uses a pair of "marks" to denote these.

Marks

Marks are a feature Vim uses to store locations in files. I am going to annoy some Vim users by skipping discussing their usage in this talk, purely to save time and because I don't use them much personally. A good introduction to marks is provided in Vim's user manual, and you can read it by running `:help 03.10`.

One More Substitute Trick

You have seen the g flag for substitutions to make them replace every occurence, instead of just the first one on each line. Another flag I use frequently is c, which makes `:substitute` ask for permission before performing each substitution. This is great if you only want to perform the substitution in some cases, and you're feeling too lazy to define a more complex regular expression to only match those specific cases.

:global

When using macros, a fairly common workflow is to search for the next place to execute your macro, and include this search in the macro recording so that when you execute the macro many times, it automatically finds its next place to execute each time. A similar workflow can also be performed with the `:global` command, which uses the following syntax:

:global syntax

```
:[range]g[lobal]/{pattern}/[command]
```

:global commands

`:global` is extremely flexible — it just runs the command you specify on matches for your provided pattern, using the same regular expression syntax as the `:substitute` command. If you don't provide a command, global uses `:p[rint]` by default, which prints the matching lines for you to view them. You could use `:d[elete]` to delete matching lines, or even `:s[ubstitute]` to perform a substitution only on lines which match the provided pattern. For the ultimate in flexibility, you can use the `:norm[al]` command to execute a string of keybindings in normal mode on matching lines.

:global!

If you want to run a command starting at the beginning of lines which match your pattern, just use `global!` instead of `global`.

:normal and :execute

The `:normal` command provides a way for you to use normal mode keybindings in command-line mode. This is useful for some commands which run other command-line commands, such as `:global`, or `:cdo` (which we'll learn about in the next section).

One difficulty with the `:normal` command is when you want to use a special character, such as `<ESC>` or `<CR>`. When you want to do this, just wrap it in an `:execute` command, which takes a string containing a command and executes it in command-line mode, and allows you to escape special characters. For example:

```
:exe "norm ifunc \<ESC>A{"
```

This command enters insert mode, types "func ", hits `<ESC>` to go back to normal mode, then inserts an opening curly brace at the end of the line.

Interacting With The Shell

Vim supports calling external commands using the `:!{cmd}` command. For example you could run `!ls` to run `ls` in a piped non-interactive shell and view the contents of the current directory. When running shell commands, Vim lets you reference the current filename using %.

If you want to read the output of a shell command into the current file, you can run `:r[ead] !{cmd}`. You can also filter several lines of the current buffer through an external command using `:{range}!{cmd}`. You can use line-wise visual mode to select a range of lines to run a filter on like with `:substitute`, or you can use the normal mode operator `!` for running a filter on an area of text defined using a motion. `!!` is a shortcut for running a filter command on the current line.

The Undo Tree

Here's a perilous situation which can occur in almost any editor: you edit a file for a while, and after making your edits, you realize you deleted something which you should not have deleted. To get that deleted text back, you "undo" multiple times, intending to copy the deleted text, so you can "redo" your changes and paste. However, you accidentally edit the file while in the older state! Now you can't redo your changes, and you've just accidentally lost all the work you had done!

Thankfully, Vim keeps track of your entire "undo tree" to avoid exactly this issue. The easiest way to use this feature is using the g- and g+ normal mode keybindings. These keybindings go to "older" and "newer" text states respectively, regardless of whether there were "undo" operations in the middle. In the situation I outlined, even though you'd performed an edit after undo-ing, you can just press g- to go to older text states until you have your edits back.

Making the Undo Tree More Usable

I find the full power of Vim's undo tree is really unlocked with plugins for visualizing it, so you can easily walk to any node in the undo tree. The one I use is undotree by mbbill.

Managing and Editing Multiple Files

Multiple Files in Vim

The following summary of Vim's terminology regarding viewing files can be found by running `:help window`:

- A buffer is the in-memory text of a file.
- A window is a viewport on a buffer.
- A tab page is a collection of windows.

Buffers

Again quoting from Vim's help files: "A buffer is a file loaded into memory for editing. The original file remains unchanged until you write the buffer to the file."

When you open a new file in Vim, a new buffer is created for that file and made active, and the current buffer is hidden. You can run `:ls` to view a list of buffers together with their state and buffer numbers. Then to switch to another buffer, run `:b[uffer] {num}` with the buffer number you want to switch to.

NOTE: I rarely use `:e` for opening files, and I rarely use `:b` for switching buffers. Often if I know I'll want to edit a file again, I keep it open in a split window or a tab page. I'll discuss my workflow for switching to files I don't already have open in splits or tabs when I discuss **Finding Files Quickly** at the end of this section.

Windows

All of Vim's window management commands are prefixed with `<C-w>`. By default, you can use `<C-w> s` to split your current window into two, one on-top of the other, and `<C-w> v` to split your window side-by-side. You can also perform these operations in command-line mode using `:sp[lit]` and `:vs[plit]`. To switch between open windows, you can use `<C-w> [hjkl]` to switch to the window in the indicated direction. To round out the basic window management commands, you can use `<C-w> c` to close the current window, and `<C-w> o` to close every window but the current one.

How I Manage Windows

In the next section we will discuss configuring Vim, and I'll show how to remap keys. I have rebound splitting and switching windows because I do these actions frequently and I find it annoying to have to press `<C-w>` every time. I have switching windows bound to `<C-[hjkl]>` to switch to the window in the indicated direction, and splitting bound to `<SPC> [hjkl]` (pressed one after another) to split in the indicated direction.

Closing vs Quitting Windows

The difference between `<C-w> c` and `:q` is that closing a window leaves the buffer open in memory, and will fail if you try to close the last window open in Vim. In contrast, quitting a window also deletes the current buffer, will close Vim if this is the last window open, and fails if there are unwritten changes to the current buffer. `<C-w> q` is synonymous with `:q`.

Tabs

As stated in Vim's documentation, a tab page is a collection of windows. To create a new tab page run `:tabnew` (you'll notice the tab bar appears at the top of the window), then you can use `gt` and `gT` to go forwards and backwards through your open tab pages. If you have many tab pages open, you can jump to a specific one using `{num}gt` which jumps to tab page number `{num}` (as opposed to jumping `{num}` tab pages forwards as you might expect). Closing the last open window on a tab page also closes the tab page.

Vimgrep

If you want to search files by their contents, Vim comes with a built-in grep implementation callable using the `:vimgrep` command. Vimgrep has the following syntax:

```
:vimgrep /{pattern}/ {files}
```

The patterns used by vimgrep are the same as those used in Vim when searching and substituting, including all the regular expression goodies. The syntax for specifying files can be looked up using `:help {file}`. Here are the basics which I use most often:

- `./foo/bar` — paths to files
- `*` — wildcards which match anything (including nothing)
- `**` — wildcards which match anything (including nothing), and recurses into subdirectories

Grep

The problem with vimgrep is that it reads every file it searches into memory in Vim, which allows it to use the powerful Vim pattern syntax, but makes it quite slow. If you want a faster grep implementation, you can use Vim's `:grep` command, which interacts with an external grep command specified by the 'grepprg' option. Both vimgrep and grep load their results into a quickfix list. We'll discuss how to use these lists on the next slide.

Quickfix Lists

Quickfix lists are Vim's way of showing lists of locations in files. These lists are used by `:vimgrep` and `:grep`, as well as `:make` which is used for compiling files then showing errors, and several plugins. You can open the quickfix list in a window using the `:copen` command and close it with `:cclose`. The quickfix list contains one location per line (filename, line + column number and line contents). You can navigate the quickfix list just like any other buffer, and pressing `<CR>` on a line takes you to that location in the specified file.

There are also commands for jumping between file locations in the quickfix list without navigating the list manually and selecting an entry. You can use the `:cnext` and `:cprev` commands to jump to the next and previous locations in the list respectively (I use these frequently enough that they're mappend to `<C-n>` and `<C-p>`). There are also `:cfirst` and `:clast` commands which jump to the start and end of the list.

Cdo

One of the killer features of Vim's quickfix lists is the command `:cdo {command}`. This command goes through the files in the quickfix list one-by-one, and executes the provided command on each entry in the quickfix list. This allows you to perform project-wide search and replace by first grepping for your pattern, then running the following combination of commands:

```
:cdo s/{pattern}/{replacement}/g | update
```

You need to include the `| update` portion of the command. In Vim's command-line mode a vertical bar separates two commands to run one-after-another. You need to run update after the command used with `:cdo`, because otherwise Vim will fail to close the current buffer when it changes files. Note, `:update` is just like `:w[rite]`, but it only writes the file if its contents changed.

Multiple Quickfix Lists

Vim keeps track of the last 10 quickfix lists, so if you perform another grep or make command you don't lose the previous one's results. To switch to the previous quickfix list, you can execute `:colder`, and to switch to the next one you can run `:cnewer`.

Because `:cdo` can be used with any Vim command, it is extremely flexible. The next slide is an example of a time I used `:cdo` in my work with BlueCat to perform a task which would have been much more tedious without it:

Cdo With Normal

In our Golang repositories, we frequently propagate errors including messages and error codes, so that we can identify what error occurred higher in the callstack. The resulting error message from a chain of these `stacktrace.Propagate` calls is several error messages concatenated together, so we decided to make the propagated error messages start with lowercase letters, otherwise they just look funny. To convert all existing error messages in the current project to start with lowercase letters (with Vim open in the project's base directory), I ran the following commands $*$:

- `:vimgrep /stacktrace\.Propagate/ ./**`
- `:cdo exe "norm f\"lvu" | update`

$*$ NOTE: I have changed the commands slightly from the originals (posted in the #vim-geeks channel), so they would work in a vanilla Vim installation without depending upon any files already being open.

Related Topics to Look Into

Vim keeps track of several other lists similar to the quickfix list. The argument list keeps track of files provided to Vim from the command line when Vim was started up. It comes with its own `:argdo` command, and `:next` and `:prev` commands, similar to the quickfix list equivalents. For more information, see `:help argument-list`.

Vim also keeps track of lists called "location lists", which are window-local quickfix lists. There are equivalent commands to the quickfix list commands which start with "l" instead of "c" to use a location list instead of a quickfix list. To read more, see `:help location-list`.

There are many other commands similar to `:cdo` which act on different sets of lines or buffers. See the help pages for `:bufdo`, `:tabdo`, `:argdo`, `:windo`, `:ldo`, `:cfdo` and `:lfdo`.

Finding Files Quickly

When I first started using Vim full-time, one of the biggest pain points for me was finding and opening files. Even with tab-completion, I always found `:e` painfully slow. Vim comes with a file-manager plugin called `netrw` preinstalled, so that when you open a path which is a directory in Vim, it opens it in netrw so you can navigate to the file you want to open. Even this felt slow. Vim provides a `:find` command which is actually pretty good when combined with the 'wildmenu' option to turn on tab-completion, but it requires setting the 'path' option correctly, and it doesn't interactively show you matches as you type. I ended up settling on a pair of plugins I have installed, which I think would be sufficient for most Vim users.

Fuzzy Finder

Most of the time when I want to open a file, I do so using fzf.vim — a fuzzy-finder plugin which interfaces with the `fzf` command-line tool. This plugin provides a `:Files` command, which lists files recursively from the current directory, then narrows the list as I type. I can select a file to open from the list at any time using `<CR>`, and I can move up or down the list using the arrow keys or `<C-n>` and `<C-p>`. I also frequently use the `:Rg` command as an interactive interface for ripgrep to do fast searches of project file contents, and `:Helptags` to interactively search the names of `:help` subjects.

Note, I always open my Vim instance in the base directory of the project I'm working on, so that my fzf commands search all the project files. This is also important for my project drawer (discussed on the next slide) because it makes sure the full file tree is displayed.

Project Drawer

This is something of a controversial topic in online Vim discussions, but I also like to use a project drawer plugin (to see the argument against this, read the Vimcasts "Oil and Vinegar" article). I have a keyboard shortcut to toggle my project drawer open and closed, and I have it set to show the location of the current file in the project when I open it. I mainly use this plugin to view the structure of a repository I'm not very familiar with, to quickly switch to another file in the same package, and to quickly create new files without having to type the full file path into an `:e` command. The project drawer plugin that I use is fern.vim, but NERDTree is definitely the most popular one.

Configuration

The Vimrc

Vim's configuration file is located at `~/.vimrc` or `~/.vim/vimrc` on Unix systems (neovim's config file is at `~/.config/nvim/init.vim`). This configuration file contains a set of commands which Vim runs every time it starts up. The lines in your vimrc are executed in a special Vim mode called "ex mode", which executes commands like the ones you execute in command-line mode, except it executes them one after another without requiring a colon before each command. To read more about ex mode, you can refer to `:help Ex-mode`.

Setting Options

Vim has a bunch of options you can either toggle on and off or set values for to change the behaviour of the editor. If you're curious, you can see a quick listing of options Vim accepts at `:help option-list`. To toggle an option on use the command `:set {option}`, to turn it off use `:set no{option}`, and to change the value of a non-boolean option you can use the command `:set {option}={value}`.

Learning About Options

On the next slide I'll go over some options I like to have set. This is a very small subset of the options you can set in Vim, so please explore Vim's help files if you want to learn more - a great place to start would be `:help 05.8` and `:help 05.9`, which are sections in the user manual which deal with options. You can find my full neovim configuration in my dotfiles repo hosted on GitHub.

Looking Up Option Documentation

When you are looking up an option in Vim's help files, the convention is that help topics for options are surrounded with single quotation marks. For example, running `:help undofile` shows you documentation for the vimscript undofile() function, but `:help 'undofile'` shows the documentation for the 'undofile' option.

Useful Options

1. `lazyredraw`: increases performance by stopping Vim from redrawing the screen while executing macros.
2. `showmatch`: when the cursor is on a bracket, highlight the matching bracket.
3. `smartindent`: automatically indent lines - works well for C-like languages.
4. `number`: turn on line numbers.
5. `relativenumber`: number lines relative to the current line (useful for `{num}j` and `{num}k` to jump a specific number of lines up or down).
6. `cursorline`: highlight the current line to make it more visible.
7. `undofile`: use persistent undo, so you can close and reopen Vim and still keep the file's undo history.

Options Specifically For Searching

1. `ignorecase`: makes Vim ignore case by default when searching.
2. `smartcase`: only used in combination with `ignorecase` - overrides `ignorecase` if you include a capital letter in your search pattern.
3. `incsearch`: highlight search results as you are typing your pattern.
4. `hlsearch`: after starting your search, highlight search results (you can disable highlighting for the current search by running `:noh`, and it will turn back on when you search again).
5. `inccommand`: **Neovim Only**, this options lets you see the effects of a `:substitute` command while you are typing it.

Keybindings

If you find there's a command, or a sequence of normal mode shortcuts which you run frequently, you can bind it to a keymapping to make it faster. The simplest command for setting up a custom keymapping is the `:map` command which uses the following syntax:

```
:map {lhs} {rhs}
```

For example, if you wanted to remap `gw` so that it saves the currently file, you could put the following line into your vimrc:

```
map gw :w<cr>
```

You Usually Shouldn't Use :map

The `:map` command has a couple major flaws:

- The keybinding is recursive, meaning if any of the keys in the `{rhs}` of the mapping are themselves the `{lhs}` of another mapping, that keymapping will also be applied.
- The keybinding it sets up applies in every Vim mode. Usually this is not what you want, because the `{rhs}` of your mapping will usually be mode-specific.

To solve the first issue, Vim provides a `:noremap` command which behaves exactly like `:map` but ignores any keymappings in the `{rhs}`. To solve the second issue, Vim allows you to add prefixes to specify what mode the keymapping applies in, for example `:nmap` for normal mode, `:imap` for insert mode and `:cmap` for command-line mode. Both of these can be (and often should be) combined, for example in the `:nnoremap`, `:inoremap` and `:cnoremap` commands. Most of the time, `:nnoremap` is the command you should be using for remapping keys in normal mode.

The Leader Key

Many people find it convenient to start many of their custom keybindings with a common prefix. To facilitate this, Vim allows you to set a "leader" key by setting a variable called "mapleader". The following command which sets my leader key to <SPC> is in my neovim config file:

```
let mapleader = " "
```

After setting this variable, the leader key can be referred to as `<leader>` in keymappings. For example, I have the following mapping in my neovim configuration to map `<SPC> e` to run the `:Files` command from the fzf plugin, to let me find files by name quickly:

```
nnoremap <leader>e :Files<cr>
```

Abbreviations

Very similar to mappings, Vim supports a feature called abbreviations, which allow you to set short forms which will expand into their full form after typing a "non-keyword" character. You can explore setting the 'iskeyword' option, however I just remember that <SPC> and <CR> are both non-keyword characters. This means that, when I type a command-line mode abbreviation, I can just hit <CR> to run it after expanding, or I can type a space if I want to edit the expanded string.

The place where I have found abbreviations most useful is in command-line mode, for long commands which I might sometimes want to edit before running. For example, I have the following abbreviation for running all the Golang tests in the current repository in a tmux split (this uses the vimux plugin for interacting with another tmux split from inside Vim):

```
    cnoreabbrev gta exec "call VimuxRunCommand(
‘cd ".getcwd()." && clear && go test ./pkg/...’)"
```

(this is all on one line in my config, but had to be split to fit the slide)

Plugins

Sometimes, despite Vim's expansive feature set, there are some features which are nice to have, but are not built into Vim by default. For these features, there is an expansive plugin library available, some of which add features to Vim, and others make Vim's built-in features more accessible.

Vim has a built-in way of managing plugins, which allows you to clone plugins into a particular directory where Vim picks them up and loads them automatically. Personally I have never tried managing plugins in this way — I adopted vim-plug by junegunn (the same author who wrote fzf) as my plugin manager quite early on, and it has worked very well for me.

Using Vim-Plug

To install vim-plug, you can download a single vimscript file, and put it in Vim's autoload directory (`~/.vim/autoload` by default on unix systems). Once vim-plug is installed, you can make a list of plugins for vim-plug to manage in your vimrc. Here is an example to show the syntax:

```
call plug#begin()
    Plug 'morhetz/gruvbox'
    Plug 'lambdalisue/fern.vim'
    ...
call plug#end()
```

Each of these plugins correspond with the a username and repository name on GitHub. Once you have set this up in your vimrc, you can use the command `:PlugInstall` to install all the plugins you have listed, and `:PlugUpdate` to update all your plugins to their latest versions. If you remove a plugin from the list in your vimrc, you can run `:PlugClean` to remove unused plugins from your filesystem.

fugitive.vim

fugitive.vim is one of Vim's most popular plugins, written by one of the most prolific and popular Vim plugin authors — tpope. This plugin provides an interactive git workflow inside Vim, largely through a single `:G` command. When run on its own, this command opens a split window which shows the current status of the git repository Vim was opened inside. This status window has several custom keybindings which let you perform convenient git operations:

- `<C-n>/<C-p>`: go to the next/previous changed file, or changed hunk inside an expanded file diff
- `=`: expand or collapse the current file diff
- `-`: stage or unstage the file that the cursor is currently over
- `cc`: create a commit containing the staged changes, and start writing the commit message (you can quit the commit window to cancel the commit, or write it to finalize the commit).

This plugin also allows you to view git logs, blames, and execute arbitrary git commands by using the `:G` command as if it is `git` on the command line.

fzf.vim

fzf.vim is another extremely popular plugin. It provides you with a standard user interface for fuzzily matching entries in a list. This plugin allows you to implement your own custom commands using this fuzzy match feature, however it also comes with several very useful commands out of the box:

- `:Files`: open a file by fuzzily matching filenames enumerated recursively from Vim's base directory
- `:Rg`: fuzzily search the results of a `ripgrep` command (ripgrep is a trendy modern implementation of grep which is very fast)
- `:Helptags`: fuzzily search tags for topics in Vim's help files
- `:Buffers`: swap buffers by fuzzily matching the names of open files

Language Server Protocol

For a long time, there has been a lively debate online regarding whether Vim can really replace an IDE, because of its lack of some language-specific features such as automatic imports, jumping to symbol definitions / references / interface implementations, autocompletion, the ability to view documentation, etc.

As part of the development of Visual Studio Code, Microsoft developed the Language Server Protocol, which was standardized in 2016. This protocol allows text editors to be decoupled from language support. All the editor needs is the ability to act as an LSP client, then it can interact with separate language servers, which provide support for a particular language.

Currently `coc.nvim` is the most popular LSP client implementation for Vim and Neovim. It is unique compared to other LSP client implementations in that it also supports "coc extensions", which are forked directly from VSCode extensions. These extensions can provide more features than the community maintained language servers. If you want IDE-like autocompletion and language-specific features in Vim, `coc.nvim` is currently the easiest way to achieve this.

Useful Topics Not Covered

Unfortunately, there is not enough time in this presentation to cover all of Vim's features, or even all the features I use. The following topics are still very useful to know about, despite not fitting into my talk.

- Marks (`:help 03.10`)
- Vimscript (warning, this one is long: `:help usr_41.txt`)
- Write and execute your own functions (`:help user-functions`, `:help :call`)
- Creating custom commands (`:help user-commands`)
- Autocommmands (`:help 40.3`)

If you want to have your mind blown by a Vim plugin, check out vimspector, which uses multiple Vim windows to create a full IDE-like debugger inside Vim.

Links For Further Learning

I have found the following resources useful while learning to use Vim. Consider taking a look if you feel like learning new Vim features, or gaining a new perspective on features you already know about.

- Max Cantor's talk "How to Do 90% of What Plugins Do (With Just Vim)"
- Erik Falor's talk "From Vim Muggle to Wizard in 10 Easy Steps"
- Chris Toomey's talk "Mastering the Vim Language"
- Drew Neil's book Practical Vim and the content on his website: vimcasts.org
- Romaine Lafourcade's Vim-related GitHub gists, and the articles from his Vim-themed online advent calendar: Vimways

All Praise VI VI VI
Editor of The Beast