THE CS 1001 ENCHIRIDION

Or, How To Use A Computer Real Good

Nathan Jarus and Michael Wisely

Contents

Introduction

Well?

Chapter 1

Exploring Text Editors

Motivation

At this point in your Computer Science career, you've worked with at least one text editor: `jpico`. Love it or hate it, `jpico` is a useful program for reading and writing **plain ASCII text**. C++ programs[1] are written in plain ASCII text. ASCII is a convenient format for humans and programs[2] alike to read and process.

Because of its simple and featureless interface, many people find editors like `jpico` to be frustrating to use. Many users miss the ability to use a mouse or simply copy/paste lines from files without bewildering keyboard shortcuts.

Fortunately, there are myriad text editors available.[3] Many popular options are available to you on campus machines and can be installed on your personal computers as well! These editors offer many features that may (hopefully) already be familiar to you. Such features include:

- Syntax highlighting
- Cut, copy, and paste
- Code completion

Whether you're writing programs, viewing saved program output, or editing Markdown files, you will often find yourself in need of a text editor. Learning the features of a specific text editor will make your life easier when programming. In this lab, you will try using several text editors with the goal of finding one that fits your needs.

[1] And many other programming languages, for that matter.

[2] Including compilers.

[3] If you are reading this, you may ignore the rest of this chapter and instead learn `ed`, the standard editor.

Several of the editors you will see do not have a graphical user interface (GUI). Although the ability to use a mouse is comfortable and familiar, don't discount the console editors! Despite their learning curves, many experienced programmers still prefer console editors due to their speed, stability, and convenience. Knowing a console editor is also handy in situations where you need to edit files on a machine halfway around the globe[4]!

Note: This chapter focuses on text editors; integrated development environments will be discussed later in the semester. Even if you prefer to use an IDE for development, you will still run into situations where a simple text editor is more convenient to use.

Takeaways

- Recognize the value of plain text editors.
- Familiarize yourself with different text editors available on campus machines.
- Choose a favorite editor; master it.

Walkthrough

Note: Because this is your first pre-lab, the walkthrough will be completed in class.

For each:

- Helpful URLs (main website) / tutorial mode
- Special terminology
- Moving around text, cut/copy/paste, nifty editing features
- Multiple files, tabs/splits
- Nifty features (e.g. notepad++ doc map)
- Configuring things; handy config settings
- Plugins

Notepad++

Notepad++[5] is a popular text editor for Windows. It is free, easy to install, and sports a variety of features including syntax highlighting and automatic indentation. Many people choose this editor because it is lightweight and easy to use.

[4] Thanks to cloud computing, this is becoming commonplace, yo.
[5] Website: https://notepad-plus-plus.org/

Keyboard shortcuts

Beyond the standard editing shortcuts that most programs use, Notepad++ has some key shortcuts that come in handy when programming. Word- and line-focused shortcuts are useful when editing variable names or rearranging snippets of code. Other shortcuts indent or outdent[6] blocks of code or insert or remove comments.

In addition to those shortcuts, if your cursor is on a brace, bracket, or parenthesis, you can jump to the matching brace, bracket, or parenthesis with Ctrl+b.

Word-based shortcuts

- Ctrl+← / →: Move cursor forward or backward by one word
- Ctrl+Del.: Delete to start/end of word

Line-based shortcuts

- Ctrl+⇧+⟵: Delete to start/end of line
- Ctrl+l: Delete current line
- Ctrl+t: Transpose (swap) current and previous lines
- Ctrl+⇧+↑ / ↓: Move current line/selection up or down
- Ctrl+d: Duplicate current line
- Ctrl+j: Join selected lines

Indenting and commenting code

- ⭾: Indent current line/block
- ⇧+⭾: Outdent current line/block
- Ctrl+q: Single-line comment/uncomment current line/selection
- Ctrl+⇧+q: Block comment curent line/selection

Column Editing

You can also select text in columns, rather than line by line. To do this, use Alt +⇧+↑ / ↓ / ← / → to perform a column selection, or hold Alt and left-click.

If you have selected a column of text, you can type to insert text on each line in the column or edit as usual (e.g., Del. deletes the selection or one character from each line). Notepad++ also features a column editor that can insert text or a column of increasing numbers. When you have performed a column selection, press Alt+c to open it.

[6] *Outdent* (verb). Latin: To remove a tooth; English: The opposite of indent.

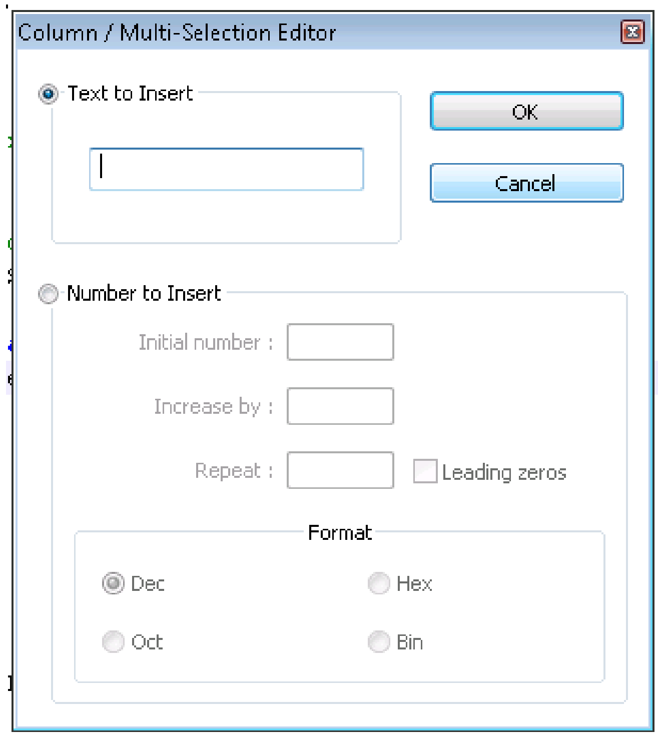


Figure 1.1: Column Editor

Multiple Cursors

Notepad++ supports multiple cursors, allowing you to edit text in multiple locations at once. To place multiple cursors, hold Ctrl and left-click everywhere you want a cursor. Then, you can type as normal and your edits will appear at each cursor location.

For example, suppose we've written the declaration for a class named `road` and that we've copied the member function declarations to an implementation file. We want to scope them (`road::width()` instead of `width()`), but that's tedious to do one function at a time. With multiple cursors, though, you can do that all in one go!

First, place a cursor at the beginning of each function name:

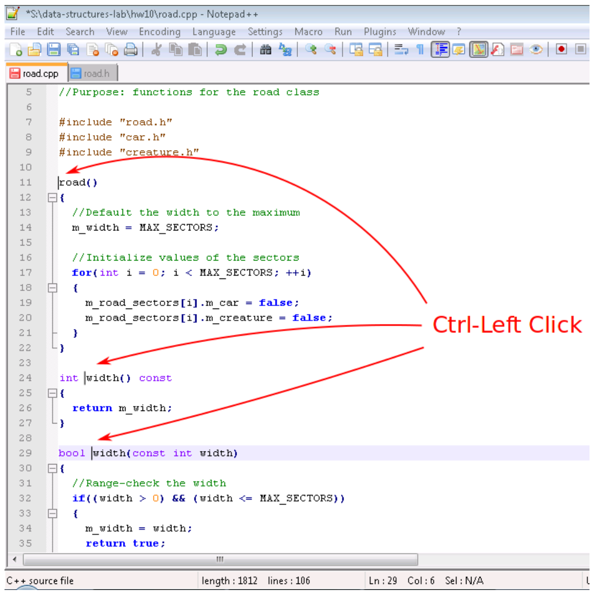


Figure 1.2: Placing multiple cursors with Ctrl + left-click

Then, type `road::`. Like magic, it appears in front of each function:

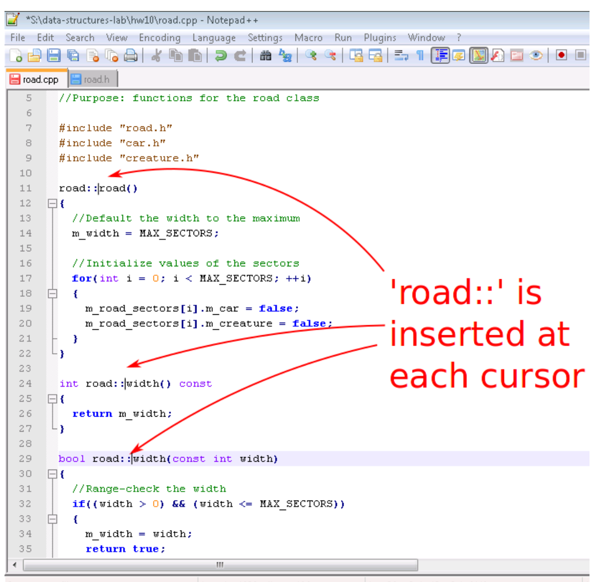


Figure 1.3: Typing `road::` inserts that text at each cursor location

Document Map

A document map can be handy when navigating large files[7]. It shows a bird's-eye view of the document; you can click to jump to particular locations.

The document map can be enabled by clicking View 〉 Document Map

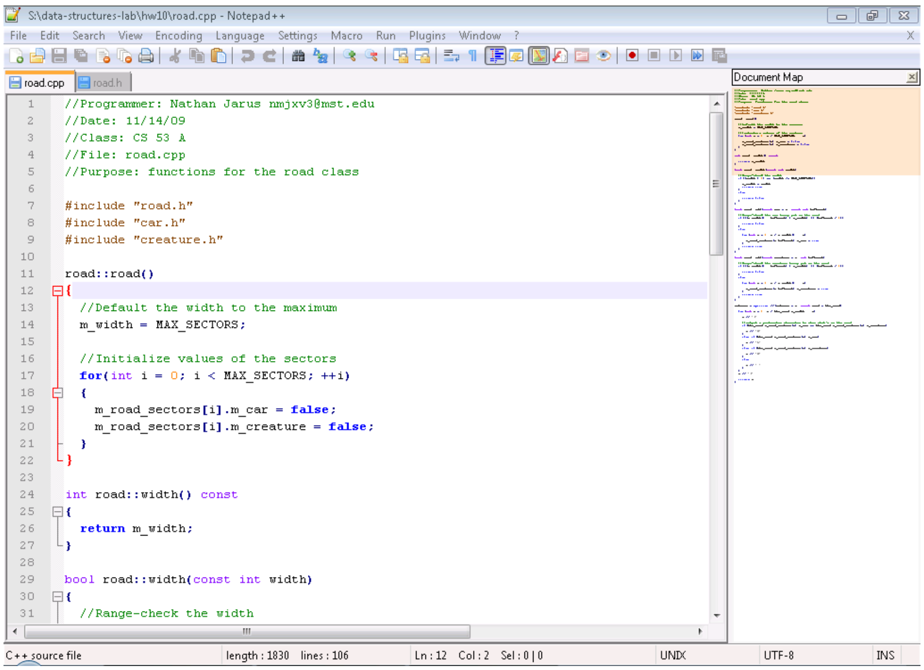


Figure 1.4: The document map

Settings

Notepad++ has a multitude of settings that can configure everything from syntax highlight colors to keyboard shortcuts. You can even customize some settings per programming language, including indentation. One common setting is to switch Notepad++ to use spaces instead of tabs:

Plugins

Notepad++ has support for plugins; you can see a list of them here[8]. Unfortunately, plugins must be installed to the same directory Notepad++ is installed

[7] Of course, this feature might encourage making large files rather than multiple manageable files. . .

[8] http://docs.notepad-plus-plus.org/index.php?title=Plugin_Central

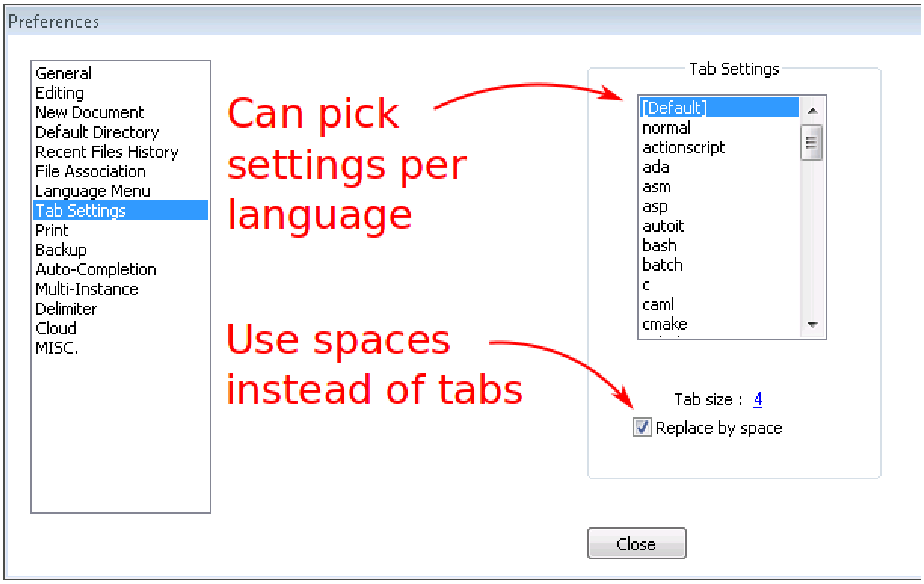


Figure 1.5: Configuring Notepad++ to use spaces rather than tabs

in, so you will need to install Notepad++ yourself to use plugins.

Atom

Programmers like to program. Some programmers like pretty things. Thus there is Atom.

Atom is a featureful text editor that is developed by GitHub. Designed with customization in mind, Atom is built on top of the engine that drives the Google Chrome web browser. Atom allows users to customize just about every feature that it offers. Style can be changed using cascading style sheets[9] and behavior can be changed using JavaScript[10].

Additionally, being a hip-and-trendy™piece of software, you can install community packages written by other developers. In fact, if you find that Atom is missing some particular behavior, you can create a package and make it available to the world, as well[11]!

Atom has a GUI, so it is mouse friendly and human friendly, too.

Figure 1.6: One Atom window with Tree View on the left and an empty pane on the right

Tree View

Using Ctrl+\ (or View › Toggle Tree View), you can toggle Atom's Tree View. The Tree View is a convenient tool for browsing files within a folder or subfolders. By clicking the down arrow to the left of a folder, you can see its contents. Simply double click a file to open it up.

As you double click files, they open up in new **tabs**.

Tabs

To switch between tabs, simply click on them at the top. It works much the same way as browser tabs do.

Keep an eye on your tabs! Atom will indicate when a file has changed and needs to be saved. This can be very helpful when you find yourself asking "why is `g++` *still* complaining?".

Panes

In addition to opening files in several tabs, you can display several files at once in separate **panes**. Each pane has its own collection of tabs.

You can split your window into **left-and-right** panes by right-clicking in the tabs area and choosing Split Right or Split Left. You can split your window

[9] CSS is used to specify the design for websites, and it works in Atom, too.
[10] JavaScript[^java] is the language of the web. It makes web pages interactive!
[11] Just don't expect to get rich.

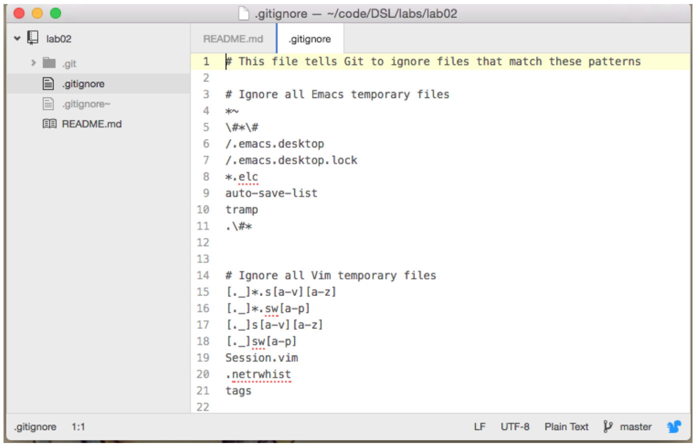


Figure 1.7: Atom with multiple tabs in one pane

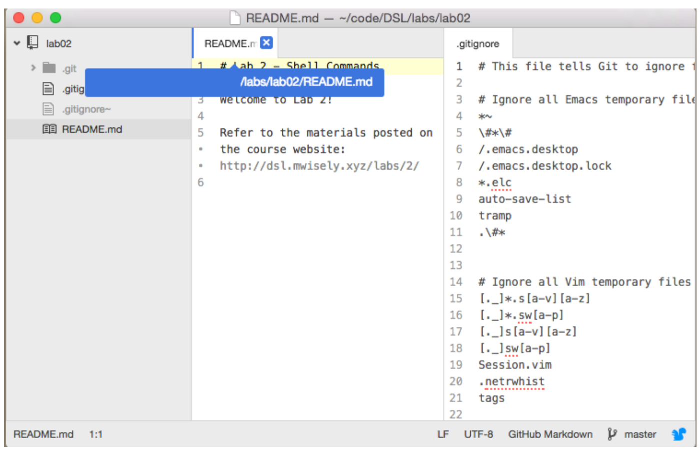


Figure 1.8: One window split into two panes

into **top-and-bottom** panes by right-clicking in the tabs area and choosing Split Down or Split Up. You can also close panes by choosing Close Pane.

The Command Palette

You may notice that Atom's drop-down menu options are sparse. There is not much to choose from. Don't fret[12]!

Most of Atom's functionality is accessible using its **command palette**. To open the command palette simply type Ctrl + ⇧ + p. The command palette is the

[12]Please, please don't fret. It'll be OK. Just keep a-readin', friend.

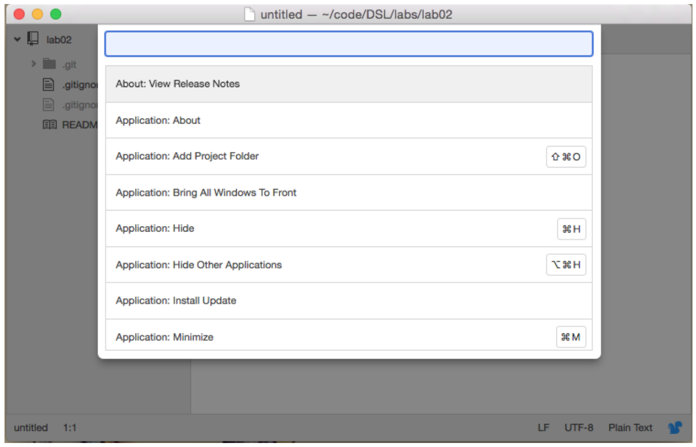


Figure 1.9: The Atom command palette (i.e., your best friend forever).

place to search for any fancy thing you might want to do with Atom.

Any.

Fancy.

Thing.

You can even use it to accomplish a lot of the tasks you would otherwise use your mouse for! For example, you can split your pane using the `pane:split-right` command in the command palette.

Many of the commands have corresponding **keybindings**, as well. These are *very* handy, as they can save you a lot of command typing.

Customization

If you open up Atom's settings (using the menu or command palette), you'll find quite a few bells and whistles that you can customize. As you explore these options, take note that you can search for keybindings here. Atom has a helpful search tool that makes it easy to quickly find the keybinding for a particular command.

If you don't see a keybinding for a command you like, just create your own! You can also choose preset keymaps to make Atom behave like other text editors including (but not limited to) emacs!

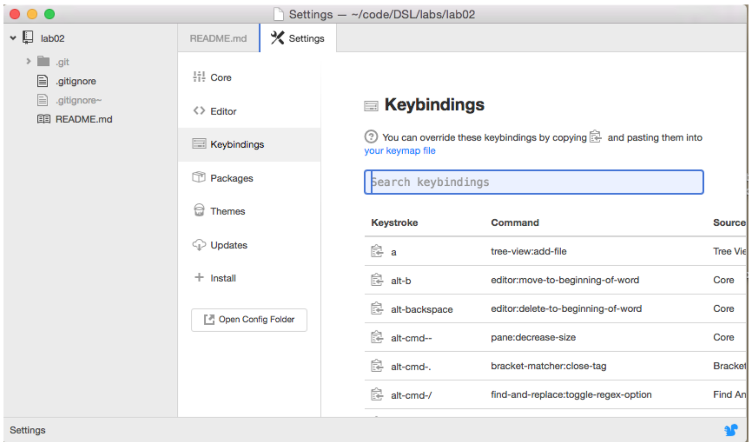


Figure 1.10: Atom's settings open in a new tab. You can search through its keybindings here.

JPico

`jpico` is a command-line text editor for Linux, Windows, and macOS. People choose this editor because it is easy to use (as command-line editors go), has syntax highlighting, and is usually installed on Linux systems. It may seem simple, but it has a surprising number of features that most people are unaware of. Many features draw inspiration from `emacs`, so you may observe some parallels between the two editors.

(Historical note: `jpico` is actually `joe`[13] configured to use commands similar to `pico`[14]. `pico` is a small (eh? eh?) text editor that came with the PINE newsreader[15] and was designed to be easy to use[16].)

How to Get Help

At the top of the `jpico` screen is a window with help information. You can toggle it on and off with Ctrl + g. There are several pages of help information. To scroll forwards through the pages, press Esc and then . ; to scroll backwards, press Esc and then , .

The notation for controls may be unfamiliar to you. In Unix-land, ˆ is shorthand for the Ctrl key. So, for instance, `^X` corresponds to Ctrl + x. For historical

[13] http://joe-editor.sourceforge.net
[14] http://www.guckes.net/pico/
[15] A newsreader is a program for reading Usenet posts. Imagine Reddit, but in the 1980s.
[16] Well, easy to use for people (sometimes known as 'humanitty') who were already used to using Unix terminal programs!

reasons[17], pressing Ctrl+[is the same as pressing Esc, so something like `^[K` corresponds to Esc, then k.

The `joe` website contains more detailed documentation, but the key mappings are different. It is still useful as an explanation behind the rather terse help messages in `jpico`!

Moving Around

If you'd rather exercise your pinky finger (i.e., press Ctrl a lot) than use the arrow keys, you can move your cursor around with some commands:

- Ctrl+f: Forward (right) one character
- Ctrl+b: Back (left) one character
- Ctrl+p: Up one line
- Ctrl+n: Down one line
- Ctrl+a: Beginning of line
- Ctrl+e: End of line

You can also move by word; Ctrl+Space moves forward one word, and Ctrl+z moves back one word.

PgUp and PgDn move up and down one screen at a time. Alternatively, Ctrl+y and Ctrl+v do the same thing.

Analogously, to jump to the beginning of a file, press Ctrl+w Ctrl+y, and to jump to the end, Ctrl+w Ctrl+v.

If there's a particular line number you want to jump to (for instance, if you're fixing a compiler error), press Ctrl+w Ctrl+t, then type the line number to go to and press Enter.

Deleting text is the same as cutting text in `jpico`. See the **Copy and Paste** section for a list of ways to delete things.

Undo and Redo

These are pretty easy. Undo is Esc,-; redo is Esc,=.

---

[17] In the '60s and '70s, Ctrl cleared the top three bits of the ASCII code of whatever key you pressed. The ASCII code for Esc is `0x1B` or `0b00011011` and the ASCII code for [is `0x5b` or `0b01011011`. So, pressing Ctrl+[is the same as pressing Esc. People (and old software) dislike change, so to this day your terminal still pretends that that's what's going on!

Copy and Paste

You: "So, `jpico` is kinda cool. But Ctrl+c and Ctrl+v both mean something other than 'copy' and 'paste'. Can't I just use the normal clipboard?"

Ghost of UNIX past: "It's 1969 and what on earth is a clipboard?"

You: "You know, the thing where you select some text and then copy it and you can paste that text somewhere else."

GOUP: "It's 1969 and what is 'select some text'?????"

You: "Uh, you know, maybe with the mouse, or with the cursor?"

GOUP: "The what now?"

You: "????"

GOUP: "????"

You: "?????????"

GOUP: "?????????!"

Seriously, though, the concept of a system-wide clipboard wasn't invented until the 1980s, when GUIs first became available.[18] Before that, every terminal program had to invent its own copy/paste system! Some programs, including `jpico`, don't just have *a* clipboard–they have a whole buffer (usually called a 'killring', which sounds like a death cult) of everything you've cut that you can cycle through!

There are a number of ways to cut (or delete) things:

- Ctrl+d: Cut a character
- Esc, d: Cut from the cursor to the end of the current word
- Esc, h: Cut from the cursor to the beginning of the current word
- Ctrl+k: Cut a line
- Esc, k: Cut from the cursor to the end of the line

You can repeat a cut command to add more to the last thing cut. For example, to cut several lines at once, just keep pressing Ctrl+k. When you paste, all the lines will get pasted as one piece of text.

If you want to cut a selection of text, press Ctrl+⇧+6 to start selecting text. Move the cursor around like normal; once you have completed selecting, press Ctrl+k to cut the selection.

[18] Command-line programs even predate computer screens! Before then, people used 'teletypes'–electric typewriters that the computer could control. They were incredibly slow to output anything, and there was no way to erase what had already been printed, so having a 'cursor' didn't really make much sense (how would you show it?). Some terminals didn't even have arrow keys as a result!

Cut text goes to the killring. To paste the last thing cut, press Ctrl+u. To paste something else from the killring, press Ctrl+u, then press Esc,u until the desired text appears.

Search and Replace

Ctrl+w lets you search for text. Type the text you want to search for and press Enter. `jpico` displays the following options:

`(I)gnore (R)eplace (B)ackwards Bloc(K) (A)ll files NNN (^C to abort):`

From here, you can:

- Press Enter to search forwards
- Press i to search forward and ignore the case (so searching for "bob" will match "Bob" as well)
- Press b to search backwards
- Press r to replace matches with a new string. `jpico` will prompt whether or not to replace for each match
- Press k to select from the current mark (set with \keys{Ctrl+ ⇧+ 6) to the first match
- Press a to search in all open files
- Enter a number to jump to the N-th next match

Multiple Files

`jpico` can open multiple files and has some support for displaying multiple files on the screen at once.

To open another file, press Esc,e, then enter the name of the file to open. If you press ⭾, `jpico` will show you a listing of files matching what you've typed in so far.

You can split the screen horizontally with Esc, o. Switch between windows with Esc, n and Esc, p. You can adjust the size of the window with Esc, g and Esc, j.

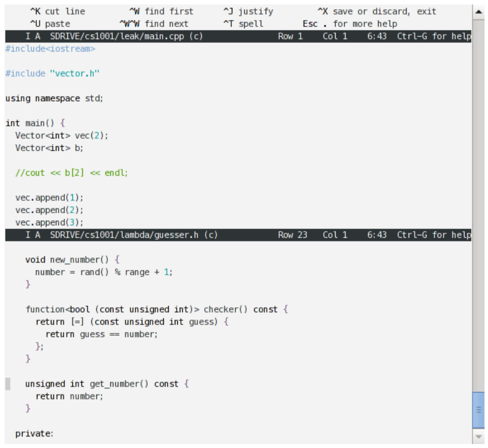


Figure 1.11: `jpico` with two files open on screen

To either show only the current window or to show all windows, press Esc, i.

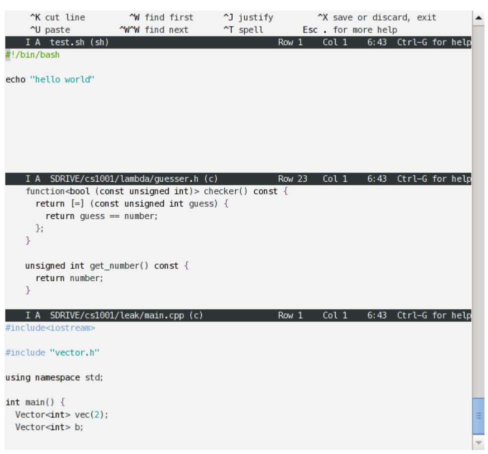


Figure 1.12: `jpico` ‘zoomed out’ to show three open files at once

Configuration

`jpico` looks for a configuration file in `~/.jpicorc`, or, failing that, in `/etc/joe/jpicorc`. To change `jpico`’s configuration, first copy the default config file to your home directory:

```
cp /etc/joe/jpicorc ~/.jpicorc
```

Each setting follows the form `-settingname options`. If there is whitespace between the `-` and the start of the line, the setting is disabled.

Some handy options:

- `-istep 4`: sets indentation width to 4 columns
- `-spaces`: uses spaces for indentation, rather than tabs
- `-mouse`: enables the mouse!

(You can read more about mouse support here[19].)

Emacs

Emacs is a command-line and GUI text editor for Linux, Windows, and macOS. Many joke that Emacs is so featureful that it is the only program you need to have installed on any computer. Some have taken this to an extreme and shown as a proof of concept that you can use Emacs as your operating system. Although that's a fun fact, you shouldn't actually do that.

Emacs was originally developed using a keyboard known as the space-cadet keyboard. Its layout is similar to, though notably different from, today's typical keyboard layout. One such difference is that the space-cadet had a Meta key, which we no longer have today. Another difference is the layout of modifier keys. Many of the Emacs keybindings (keyboard shortcuts, sort of) felt natural for space-cadet users but feel like insane acrobatics today. When starting to use Emacs, many users will find that reaching for Alt, Control, and Escape leaves their pinky fingers feeling tired and swollen. This has known as "Emacs pinky". Prolonged use of Emacs will lead to inhuman pinky strength which can be used with measurable success in combat situations.

Success with Emacs boils down to your development of muscle memory for its vast collection of keybindings. Once you have the basics down, you will find yourself angry about having to ever use a mouse. Emacs provides a tutorial that you can access from any Emacs installation. After launching Emacs, simply type Ctrl+h followed by Ctrl+t to start the tutorial. The tutorial is just like any other editable file, so you can play with it as you please. When you're done, simply exit Emacs with Ctrl+x followed by Ctrl+c. Your changes to the tutorial won't be saved.

[19] https://sourceforge.net/p/joe-editor/mercurial/ci/default/tree/docs/man.md#xterm-mouse-support

Starting Emacs

The command used to start Emacs is simply `emacs`. Just like `jpico`, you can open specific files by listing them as arguments to the command.

```
$ emacs main.cpp
```

When it starts, Emacs will first check to see whether or not it has the ability to open any GUI windows for you[20]. Assuming it can, Emacs will opt to start its GUI interface. The Emacs GUI is no more featureful than the command-line interface. Sure, you have the ability to reach for your mouse and click the Cut button, but that is no faster than simply typing Ctrl+k.

In the name of speed and convenience, many Emacs users choose to skip the GUI. You can start Emacs without a GUI by running `emacs -nw`. The `-nw` flag tells Emacs[21]...

> Dear Emacs,
>
> I know you're very fancy, and you can draw all sorts of cute shapes. That scissor you got there is dandy, and your save button looks like a floppy disk isn't that so great?
>
> Please don't bother with any of that, though. I just want you to open in the command-line like jpico, so that I can get some work done and move on with my life.
>
> With love, Me, the user.

If you choose to use the GUI, you should be aware of the following: **Emacs is still quirky and it is not going to behave like Notepad++ or Atom**. Cut, copy, and paste, for example, are not going to work the way you expect. It is really worth your time to get familiar with Emacs before you jump in blind.

Keybindings

To use Emacs (at all, really) you need to know its keybindings. Keybindings are important enough that this little bit of information deserves its own section.

Keybindings can be thought of as one or more keyboard shortcut. You may have to type a **series** of things in order to get things to work. What's more – if you mess up, you'll likely have to start again from scratch.

Keybindings are read left to right using the following notation:

[20]For example, if you are using X forwarding, Emacs can detect the ability to open a GUI for you.
[21]Well, the `nw` in `-nw` stands for no window, but Emacs takes it much more dramatically.

- The `C-` prefix indicates you need to hold the Control key while you type
- The `M-` prefix indicates you need to hold the Alt key (formerly Meta key) while you type
- Anything by itself you type **without** a modifier key.

Here are a handful of examples:

- `C-f` (Ctrl+f) – Move your cursor forward one character
- `M-w` (Alt+w) – Copy a region
- `C-x C-c` (Ctrl+x followed by Ctrl+c) – Exit Emacs
- `C-u 8 r` (Ctrl+u followed by 8 followed by r) – Type 8 lowercase `r`'s in a row.

You can always ask Emacs what a keybinding does using `C-h k <keybinding>`. For example,

- `C-h k C-f` – What does `C-f` do?
- `C-h k C-x C-c` – What does `C-x C-c` do?

Finally, if you done goofed, you can always tell Emacs to cancel your keybinding-in-progress. Simply type `C-g`. According to the Emacs help page…

> C-g runs the command keyboard-quit… this character quits directly. At a top-level, as an editor command, this simply beeps.

As mentioned, you can also use `C-g` to get your fill of beeps.

Executing Extended Commands

It is worth mention that every keybinding just runs a function in Emacs. For example, `C-f` (which moves your cursor forward) runs a function called `forward-char`. You can run any function by name using `M-x`. `M-x` creates a little command prompt at the very bottom of Emacs. Simply type the name of a command there and press Enter to run it.

For example, if you typed `M-x` and entered `forward-char` in the prompt and pressed Enter, your cursor would move forward one character. Granted, that requires… 13?… More keystrokes than `C-f`, but by golly, you can do it!

`M-x` is *very* useful for invoking commands that don't actually have keybindings.

Figure 1.13: Emacs has commands for all kinds of things!

Moving Around

Although you can use your arrow keys to move your cursor around, you will feel much fancier if you learn the proper keybindings to do so in Emacs.

Moving by character: - `C-f` Move forward a character - `C-b` Move backward a character

Moving by word: - `M-f` Move forward a word - `M-b` Move backward a word

Moving by line: - `C-n` Move to next line - `C-p` Move to previous line

Moving around lines: - `C-a` Move to beginning of line - `C-e` Move to end of line

Moving by sentence: - `M-a` Move back to beginning of sentence - `M-e` Move forward to end of sentence

Scrolling by page: - `C-v` Move forward one screenful (Page Down) - `M-v` Move backward one screenful (Page Up)

Some other useful commands: - `C-l` Emacs will keep your cursor in place and shift the text within your window. Try typing `C-l` a few times in a row to see what it does. - `C-s` starts search. After you type `C-s`, you will see a prompt at the bottom of Emacs. Simply type the string you're searching for and press Enter. Emacs will highlight the matches one at a time. Continue to type `C-s` to scroll through all the matches in the document. `C-g` will quit.

Undo and Redo

Type `C-_` to undo the last operation. If you type `C-_` repeatedly, Emacs will continue to undo actions as far as it can remember.

The way Emacs saves actions takes a little getting used to. Undo actions are, themselves, undo-able. The consequences of this are more obvious when you play around with `C-_` yourself.

To add further quirkiness, Emacs doesn't have redo. So don't mess up, or you're going to have to undo all your undoing.

Saving and Quitting

You can save a document with `C-x C-s`. If necessary, Emacs will prompt you for a file name. Just watch the bottom of Emacs to see if it's asking you any questions.

You can quit Emacs with `C-x C-c`. If you have anything open that has not been saved, Emacs will prompt you to see if you really want to quit.

Kill and Yank

In Emacs, your "copied" and "cut" information is stored in the "kill ring"[22]. The kill ring is... a ring that stores things you've killed (cut), so that you can yank (paste) them later.

Vocabulary:

- **Kill**[23] - Cut
- **Yank** - Paste

In order to kill parts of a file, you'll need to be able to select them. You can select a region by first setting a mark at your current cursor location with `C-space`. Then, simply move your cursor to highlight the stuff you want to select. Use `C-w` to kill the selection and add it to your kill ring. You can also use `M-w` to kill the selection without actually removing it (copy instead of cut).

If you want to get content out of your kill ring, you can "yank" it out with `C-y`. By default, `C-y` will yank whatever you last killed. You can follow `C-y` with `M-y` to circle through other things you've previously killed. That is, Emacs will maintain a history of things you've killed.

That's right! Emacs' kill ring is more sophisticated than a clipboard, because you can store **several** things in there.

To understand why it's called the kill **ring**, consider the following scenario:

[22] Don't ask why Emacs has such violent terms. There's no keyboard-related excuse for that one.

[23] Don't ask why Emacs has such violent terms. There's no keyboard-related excuse for that one.

> First, I kill "Kermit". Then, I kill "Ms. Piggy". Then, I kill "Gonzo".
>
> Next, I yank from my kill ring. Emacs will first yank "Gonzo". If I use `M-y` to circle through my previous kills, Emacs will yank "Ms. Piggy". If I use `M-y` again, Emacs will yank "Kermit".
>
> If I use `M-y` **again**, Emacs will yank "Gonzo" again.

You can circle through your kill ring as necessary to find previously killed content. Emacs will simply replace the yanked text with the next thing from the kill ring.

Multiple Buffers and Windows

You can have several different files open in Emacs at once. Simply use `C-x C-f` to open a new file into a new buffer. By default, you can only see one buffer at a time.

You can switch between the buffers using `C-x b`. Emacs will open a prompt asking for the name of the buffer you want to switch to. You have several options for entering that name:

1. Type it! Tab-completion works, so that's handy.
2. Use your arrow keys to scroll through the names of the buffers.

If you're done with a buffer, you can kill[24] it (close it) using `C-x k`.

You can also see a list of buffers using `C-x C-b`. This will open a new **window** in Emacs.

You can switch between windows using `C-x o`. This is convenient if you want to, say, have a `.h` file and a `.cpp` file open at the same time. `C-x b` works the same for switching buffers, so you can tell Emacs which buffer to show in each window.

You can open windows yourself, too:

- `C-x 2` (runs `split-window-below`) splits the current window in half by drawing a line left-to-right.
- `C-x 3` (runs `split-window-right`) splits the current window in half by drawing a line top-to-bottom.

And, of course, you can close windows, too.

- `C-x 0` closes the current window
- `C-x 1` closes every window **except** the current window. This command is **very** handy if Emacs opens too much junk.

[24] Don't ask why Emacs has such violent terms. There's no keyboard-related excuse for that one.

Configuration and Packages

Emacs stores all of its configuration using a dialect of the Lisp programming language. The default location of its configuration file is in your home directory in `.emacs/init.el`. The `init.el` file contains Lisp code that Emacs runs on start up (**init**ialization). This runs code and sets variables within Emacs to customize how it behaves.

Although you can (and sometimes have to) write your own Lisp code, it's usually easier to let Emacs do it for you. Running the `customize` command (`M-x customize`) will start the customization tool. You can use your normal moving-around keybindings and the Enter key to navigate through the `customize` menus. You can also search for variables to change.

For example:

1. Run the `customize` command (`M-x customize`)
2. In the search bar, type "indent-tabs". Then move your cursor to `[ Search ]` and press Enter.
3. Locate the `Indent Tabs Mode` option and press the `[Toggle]` button by placing your cursor on it and pressing Enter. You'll notice that the State changes from `STANDARD` to `EDITED`.
4. Press the `[ State ]` button and choose option `1` for Save for Future Sessions.

These steps will modify your `init.el` file, so that Emacs will use spaces instead of tab characters whenever you press the tab key. It may seem tedious, but `customize` will always write correct Lisp code to your `init.el` file.

`customize` and other more advanced commands are available by default in Emacs. As further evidence that it is nearly its own operating system, you can install packages in Emacs using its built-in package manager.

If you run the `list-packages` command (`M-x list-packages`), you can see a list of packages available for install. Simply scroll through the list like you would any old buffer. For instructions on installing packages and searching for packages in unofficial software repositories, refer to the Emacs wiki.

Vim

Vim is a command-line and GUI[25] text editor for Linux, Windows, and macOS. It is popular for its power, configurability, and the composability of its commands.

For example, rather than having separate commands for deleting words, lines, paragraphs, and the like, Vim has a single delete command (`d`) that can be

[25] The graphical version is cleverly named `gvim`.

combined with motion commands to delete a word (`w`), line (`d`), paragraph (`{`), etc. In this sense, learning to use Vim is like learning a language: difficult at first, but once you become fluent it's easy to express complex tasks.

Vim offers a tutorial: at a command prompt, run `vimtutor`. You can also access help in Vim by typing `:help <thing you want help with>`. The help search can be tab-completed. To close the help window, type `:q`.

Getting into Insert mode

Vim is what's known as a 'modal editor'; keys have different meanings in different modes. When you start Vim, it is in 'normal' mode; here, your keys will perform different commands – no need to press Ctrl all the time! However, usually when you open a text file, you want to, you know, type some text into it. For this task, you want to enter 'insert' mode. There are a number of ways to put `vim` into insert mode, but the simplest is just to press i.

Some other ways to get into insert mode:

- I: Insert at beginning of line
- a: Insert after cursor (append at cursor)
- A: Insert at end of line (Append to line)
- o: Insert on new line below cursor
- O: Insert on new line above cursor

When in insert mode, you can move around with the arrow keys.

To get back to normal mode, press Esc or Ctrl+c. (Many people who use `vim` swap Caps Lock and Esc to make switching modes easier.)

Moving around in Normal mode

In normal mode, you can move around with the arrow keys, but normal mode also features a number of motion commands for efficiently moving around files. Motion commands can also be combined with other commands, as we will see later on.

Some common motions:

- j/k/h/l: up/down/left/right[26]
- ^/$: Beginning/end of line

[26] Why these letters? Two reasons: first, they're on the home row of a QWERTY keyboard, so they're easy to reach. Second, when Bill Joy wrote `vi` (which inspired `vim`), he was using a Lear Siegler ADM-3A terminal, which didn't have individual arrow keys. Instead, the arrow keys were placed on the h, j, k, and l keys. This keyboard is also the reason for why ~ refers to your home directory in Linux: ~ and Home are on the same key on an ADM-3A terminal.

- w: Next word
- e: End of current word, or end of next word
- b: Back one word
- %: Matching brace, bracket, or parenthesis
- gg/G: Top/bottom of document

Commands can be repeated a number of times; for instance, `3w` moves forward three words.

One very handy application of the motion keys is to change some text with the c command. For example, typing `c$` in normal mode deletes from the cursor to the end of the line and puts you in insert mode so you can type your changes. Repeating a command character twice usually applies it to the whole current line; so `cc` changes the whole current line.

Selecting text in Visual mode

Vim has a visual mode for selecting text; usually this is useful in conjunction with the change, yank, or delete commands. v enters visual mode; motion commands extend the selection. If you want to select whole lines, V selects line-by-line instead.

Vim also has a block select mode: Ctrl+v. In this mode, you can select and modify blocks of text similar to Notepad++'s column selection feature. Pressing I will insert at the beginning of the selection. After returning to normal mode, whatever you insert on the first row is propagated to all other rows. Likewise, c can be used to change the contents of a bunch of rows in one go.

Undo and Redo

To undo a change, type u. U undoes all changes on the current line.

To redo (undo an undo), press Ctrl+r.

Saving and Quitting

In normal mode, you can save a file by typing `:w`. To save and quit, type `:wq` or `ZZ`.

If you've saved your file already and just want to quit, `:q` quits; `:q!` lets you quit without saving changes.

Copy and Paste

Vim has an internal clipboard like `jpico`. The command to copy (yank, in Vim lingo) is `y`. Combine this with a motion command; `yw` yanks one word and `y3j` yanks 4 lines. As with `cc`, `yy` yanks the current line.

In addition to yank there is the `d` command to cut/delete text; it is used in the same way.

Pasting is done with `p` or `P`; the former pastes the clipboard contents after the character the cursor is on, the latter pastes before the cursor.

While Vim lacks a killring, it does allow you to use multiple paste registers with the `"` key. Paste registers are given one-character names; for example, `"ayy` yanks the current line into the `a` register. `"ap` would then paste the current line elsewhere.

If you want to copy to the system clipboard, the paste register name for that is `+`. So `"+p` would paste from the system clipboard. (To read about this register and other special registers, type `:help registers`.)

Indenting

You can indent code one level with `>` and outdent with `<`. Like `c`, these must be combined with a motion or repeated to apply to the current line. For instance, `>%` indents everything up to the matching '}' (or bracket or parenthesis) one level.

Vim also features an auto-indenter: `=`. It is incredibly handy when copying code around. For example, `gg=G` will format an entire file (to break the command down, `gg` moves to the top of the file, then `=G` formats to the bottom).

Multiple files

In Vim terminology, every open file is a 'buffer'. Buffers can be active (visible) or hidden (not on the screen). When you start Vim, it has one window open; each window can show a buffer.

Working with buffers:

- `:e <filename>` opens (edits) a file in a new buffer. You can use tab completion here!
- `:bn` and `:bp` switch the current window to show the next or previous buffer
- `:b <filename>` switches to a buffer matching the given filename (tab completion also works here)
- `:buffers` shows a list of open buffers

Working with windows:

- `:split` splits the current window in half horizontally
- `:vsplit` splits the current window vertically
- Ctrl+w w switches focus to the next window
- Ctrl+w h/j/k/l switches focus to the window above/left/right/below the current window

Vim also has tabs!

- `:tabe` edits a file in a new tab
- `gt` and `gT` switch forward and backward through tabs

To close a window/tab, type `:q`. `:qall` or `:wqall` let you close / save and close all buffers in one go.

Configuration

Vim's user configuration file is located at `~/.vimrc` or `~/.vim/vimrc`. You do not need to copy a default configuration; just create one and add the configuration values you like.

Vim has a mouse mode that can be used to place the cursor or select things in visual mode. In your config file, enter `set mouse=a`.

To use four-space tabs for indentation in Vim, the following two options should be set:

```
set tabstop=4
set expandtab
```

Questions

Note: Because this is your first pre-lab, questions will be answered in class.

For each of the following editors…

- Notepad++
- Atom
- JPico
- Vim
- Emacs

… figure out how to do the following:

- Open a file
- Save a file
- Close a file
- Exit the editor
- Move your cursor around
  - Up/down/right/left
  - Skip words
- Edit the contents of a file
- Undo
- Cut / Copy / Paste ("Yank")
  - Whole lines
  - Select/highlight areas of text
- Open two files at once (tabs/splits/whatever)
  - Change between open files
  - View a list of open files
  - Return to a normal view/frame (just a single file)
- Configure your editor
  - How do you change settings? (Tab width, cleanup trailing whitespace, UI colors, etc.)

Name: ______________________________

For each editor, answer the following questions:

- What do you like about it?

- What do you dislike about it?

Each editor has its own learning curves, but any seasoned programmer will tell you the value of knowing your text editor inside and out. Pick an editor and master it. If you get tired of one, try a different one!

Once you get comfortable with an editor, check out "plugins" for various languages. These can assist you as you code to correct your syntax as you go as well as various other features.

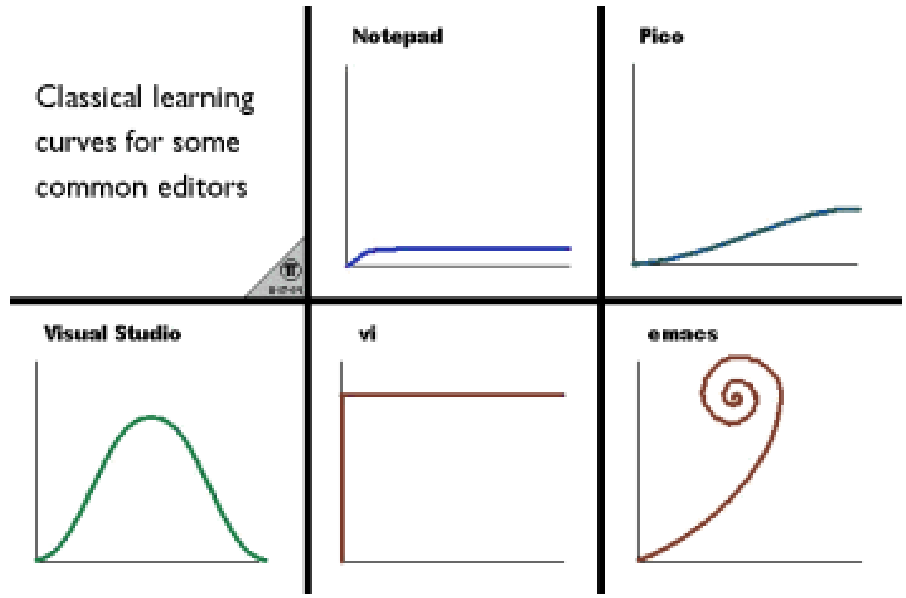


Figure 1.14: Learning editors is easy and fun!

Quick Reference

- How to get out of an editor / help everything is broken
- Doing common stuff: open file, save, motion commands

Further Reading

Notepad++

- The Notepad++ Website
- The Notepad++ Wiki, a handy reference for a lot of Notepad++ features
- Notepad++ Plugin Directory, a list of plugins you might want to install
- Notepad++ Source Code – Notepad++ is free and open source, so you can modify it yourself!

JPico

- The Joe Website
- Joe Manual
- Joe Source Code
- Some Joe History

Vim

- Vim Website
- A Vim Cheat Sheet
- Another Vim Cheat Sheet
- Why do people use Vi?, with handy examples of how to combine Vim features together
- Vim Tips Wiki, full of useful "how do I do X" articles
- Vim Plugins Directory (there are a LOT of plugins...)
- Vim Source Code

Chapter 2

Bash Basics

Motivation

What is a shell? A shell is a hard outer layer of a marine animal, found on beaches.

Figure 2.1: A shell.

Now that that's cleared up, on to some cool shell facts. Did you know that shells play a vital role in the Linux operating system? PuTTY lets you type stuff to your shell and shows you what the shell outputs.

When you log in to one of these machines, a program named `login` asks you for your username and password. After you type in the right username and password, it looks in a particular file, `/etc/passwd`, which lists useful things about you like where your home directory is located. This file also has a program name in it – the name of your shell. `login` runs your shell after it finishes setting up everything for you.

Theoretically, you can use anything for your shell, but you probably want to use a program designed for that purpose. A shell gives you a way to run programs and view their output. Typically they provide some built-in features as well. Shells also keep track of things such as which directory you are currently in.

The standard interactive shell is `bash`[1]. There are others, however! `zsh` and `fish` are both popular.

Takeaways

- Learn what a shell is and how to use common shell commands and features
- Become comfortable with viewing and manipulating files from the command line
- Use I/O redirection to chain programs together and save program output to files
- Consult the manual to determine what various program flags do

Walkthrough

My Dinner with Bash

To use bash, you simply enter commands and press Enter. Bash will run the corresponding program and show you the resulting output.

Some commands are very simple to run. Consider `pwd`:

```
nmjxv3@rc02xcs213:~$ pwd
/usr/local/home/njmxv3
```

When you type `pwd` and press Enter, bash runs `pwd` for you. In turn, `pwd` outputs your present working directory (eh? eh?) and bash shows it to you.

[1] The 'Bourne Again Shell', known for intense action sequences, intrigue, and being derived from the 'Bourne shell'.

Arguments

Some commands are more complex. Consider `g++`:

```
nmjxv3@rc02xcs213:~$ g++ main.cpp
```

`g++` needs more information than `pwd`. After all, it needs *something* to compile.

In this example, we call `main.cpp` a **command line argument**. Many programs require command line arguments in order to work. If a program requires more than one argument, we simply separate them with spaces.

Flags

In addition to command line arguments, we have **flags**. A flag starts with one or more `-` and may be short or long. Consider `g++` again:

```
nmjxv3@rc02xcs213:~$ g++ -Wall main.cpp
```

Here, we pass a command line argument to `g++`, as well as a flag: `-Wall`. `g++` has a set of flags that it knows. Each flag turns features on or off. In this case, `-Wall` asks `g++` to turn on *all warnings*. If *anything* looks fishy in `main.cpp`, we want to see a compiler warning about it.

Reading Commands in this Course

Some flags are optional; some command line arguments are optional. In this course, you will see **many** different commands that take a variety of flags and arguments. We will use the following notation with regard to optional or required flags and arguments:

- If it's got angle brackets (`<>`) around it, it's a placeholder. **You** need to supply a value there.
- If it's got square brackets (`[]`) around it, it's optional.
- If it doesn't have brackets, it's required.

For example:

- `program1 -f <filename>`
  - A `filename` argument is required, but you have to provide it in the specified space
- `program2 [-l]`

  - The `-l` flag is optional. Pass it only if you want/need to.

- `program3 [-l] <filename> [<number of cows>]`
  - The `-l` flag is optional. Pass it only if you want/need to.
  - A `filename` argument is required, but you have to provide it in the specified space
  - The `number of cows` argument is optional. If you want to provide it, it's up to you to decide.

Filesystem Navigation

Close your eyes. It's May 13, 1970. The scent of leaded gasoline exhaust fumes wafts through the open window of your office, across the asbestos tile floors, and over to your Teletype, a Model 33 ASR. You type in a command, then wait as the teletype prints out the output, 10 characters per second. You drag on your cigarette. The sun is setting, and you haven't got time for tomfoolery such as typing in long commands and waiting for the computer to print them to the teletype. Fortunately, the authors of Unix were thoughtful enough to give their programs short names to make your life easier! Before you know it, you're done with your work and are off in your VW Beetle to nab some tickets to the Grateful Dead show this weekend.

Open your eyes. It's today again, and despite being 40 years in the future, all these short command names still persist[2]. Such is life!

Look Around You with `ls`

If you want to see (list) what files exist in a directory, `ls` has got you covered. Just running `ls` shows what's in the current directory, or you can give it a path to list, such as `ls cool_code/sudoku_solver`. Or, let's say you want to list all the `cpp` files in the current directory: `ls *.cpp`[3].

But of course there's more to `ls` than just that. You can give it command options to do fancier tricks.

`ls -l` displays a detailed list of your files, including their permissions, sizes, and modification date. Sizes are listed in terms of bytes; for human readable sizes, use `-h`.

Here's a sample of running `ls -lh`:

```
nmjxv3@rc02xcs213:~/SDRIVE/cs1001/leak$ ls -lh
total 29M
-rwxr-xr-x 1 nmjxv3 mst_users 18K Jan 15  2016 a.out
```

[2] Thanks, old curmudgeons who can't be bothered to learn to type 'list'.
[3] We'll talk more about `*.cpp` later on in this chapter.

```
-rwxr-xr-x 1 nmjxv3 mst_users 454 Jan 15  2016 main.cpp
drwx------ 2 nmjxv3 mst_users   0 Dec 28  2015 oclint-0.10.2
-rwxr-xr-x 1 nmjxv3 mst_users 29M Dec 28  2015 oclint-0.10.2-x86_64.tar.gz
-rwxr-xr-x 1 nmjxv3 mst_users 586 Jan 15  2016 vector.h
-rwxr-xr-x 1 nmjxv3 mst_users 960 Jan 15  2016 vector.hpp
```

The first column shows file permissions; the fifth file size; the sixth the last time the file was modified; and the last the name of the file itself.

Another `ls` option lets you show hidden files. In Linux, every file whose name begins with a `.` is a 'hidden' file[4]. (This is the reason that many configuration files, such as `.vimrc`, are named starting with a `.`.) To include these files in a directory listing, use the `-a` flag. You may be surprised by how many files show up if you run `ls -a` in your home directory!

Change your Location with `cd`

Speaking of directories, if you ever forget which directory you are currently in, `pwd` (short for "print working directory") will remind you.

You can change your directory with `cd`, e.g. `cd mycooldirectory`. `cd` has a couple tricks:

- `cd` with no arguments takes you to your home directory
- `cd -` takes you to the last directory you were in

Shorthand

Linux has some common shorthand for specific directories:

- `.` refers to the current directory
- `..` refers to the parent directory; use `cd ..` to go up a directory
- `~` refers to your home directory, the directory you start in when you log in to a machine
- `/` refers to the root directory – EVERYTHING lives under the root directory somewhere

If you want to refer to a group of files that all follow a pattern (e.g., all files ending in `.cpp`), you can use a "glob" to do that. Linux has two glob patterns:

[4]This convention stems from a "bug" in `ls`. When `.` and `..` were added to filesystems as shorthand for "current directory" and "parent directory", the developers of Unix thought that people wouldn't want to have these files show up in their directory listings. So they added a bit of code to `ls` to skip them: `if(name[0] == '.') continue;`. This had the unintended effect of making every file starting with `.` not appear in the directory listing.

- `*` matches 0 or more characters in a file/directory name
- `?` matches exactly one character in a file/directory name

So, you could do `ls array*` to list all files starting with 'array' in the current directory.

Rearranging Files

If you want to move a file, use the `mv` command. For instance, if you want to rename `bob.txt` to `beth.txt`, you'd type `mv bob.txt beth.txt`. Or, if you wanted to put Bob in your directory of cool people, you'd type `mv bob.txt cool-people/`. You can move directories in a similar fashion.

Note: Be careful with `mv` (and `cp`, `rm`, etc.)! Linux has no trash bin or recycle can, so if you move one file over another, the file you overwrote is gone forever!

If you want to make sure this doesn't happen, `mv -i` interactively prompts you if you're about to overwrite a file, and `mv -n` never overwrites files.

To copy files, use the `cp` command. It is similar to the `mv` command, but it leaves the source file in place. When using `cp` to copy directories, you must specify the 'recursive' flag; for instance: `cp -r cs1001-TAs cool-people`[5].

You can remove (delete) files with `rm`. As with `cp`, you must use `rm -r` to delete directories.

To make a new directory, use `mkdir new_directory_name`. If you have a bunch of nested directories that you want to make, the `-p` flag has got you covered: `mkdir -p path/with/directories/you/want/to/create` creates all the missing directories in the given path. No need to call `mkdir` one directory at a time!

Looking at Files

`cat` prints out file contents. It's name is short for "concatenate", so called because it takes any number of input files and prints all their contents out.

Now, if you `cat` a big file, you'll probably find yourself wanting to scroll through it. The program for this is `less`[6]. You can scroll up and down in `less` with the arrow keys or `j` and `k` (like Vim). Pressing `Space` scrolls one page. Once you're done looking at the file, press `q` to quit.

[5] The reason for this difference between `cp` and `mv` is that moving directories just means some directory names get changed; however, copying a directory requires `cp` to copy every file in the directory and all subdirectories, which is significantly more work (or at least it was in the '70s).

[6] `less` is a successor to `more`, another paging utility, or as the authors would put it, `less` is `more`.

Other times, you just want to see the first or last bits of a file. In these cases, `head` and `tail` have got you covered. By default they print the first or last ten lines of a file, but you can specify how many lines you want with the `-n` flag. So `head -n 5 main.cpp` prints the first five lines of `main.cpp`.

The Manual

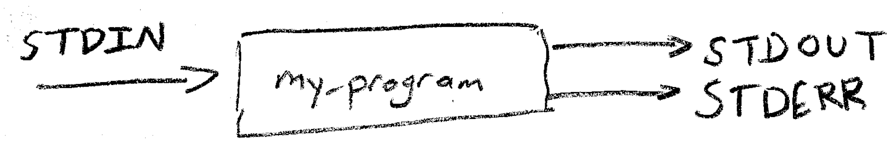
Many programs include help text; typically `--help` or `-h` display this text. It can be a good quick reference of common options.

If you need more detail, Linux includes a manual: `man`. Typically the way you use this is `man program_name` (try out `man ls`). You can scroll like you would with `less`, and `q` quits the manual.

Inside `man`, `/search string` searches for some text in the man page. Press `n` to go to the next match and `N` to go to the previous match.

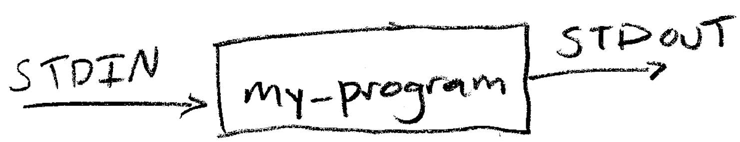
I/O Redirection

When a program runs, it has access to three different 'streams' for IO:



In C++, you read the STDIN stream using `cin`, and you write to STDOUT and STDERR through `cout` and `cerr`, respectively. For now, we'll ignore STDERR (it's typically for printing errors and the like).

Not every program reads input or produces output! For example, `echo` only produces output – it writes whatever arguments you give it back on stdout.



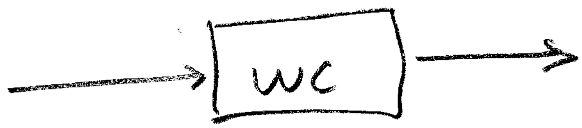
By default, STDOUT gets sent to your shell:

```
nmjxv3@rc02xcs213:~$ echo "hello"
hello
```

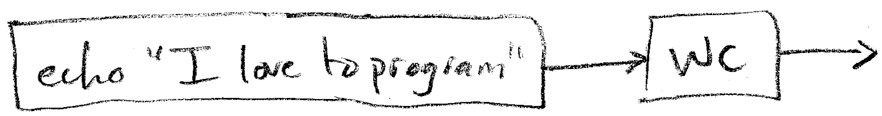
But, we can redirect this output to files or to other programs!

- `|` redirects output to another program. This is called "piping"
- `>` and `>>` redirect program output to files. Quite handy if you have a program that spits out a lot of text that you want to look through later

For example, let's take a look at the `wc` command. It reads input on STDIN, counts the number of characters, words, and lines, and prints those statistics to STDOUT.



If we type `echo "I love to program" | wc`, the `|` will redirect `echo`'s output to `wc`'s input:

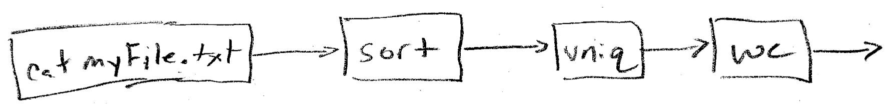



```
nmjxv3@rc02xcs213:~$ echo "I love to program" | wc
      1       4      18
```

Piping lets us compose all the utilities Linux comes with into more complex programs[7]. For a more complex example, let's suppose we want to count the number of unique lines in a file named 'myFile.txt'. We'll need a couple new utilities:

- `sort` sorts lines of input
- `uniq` removes adjacent duplicate lines

So, we can do `cat myFile.txt | sort | uniq | wc` to sort the lines in 'myFile.txt', then remove all the duplicates, then count the number of lines, words, and characters in the deduplicated output!



Another common use for piping is to scroll through the output of a command that prints out a lot of data: `my_very_talkative_program | less`.

[7] And each program in a pipeline can run in parallel with the others, so you can even take advantage of multiple CPU cores!

We can use `>` to write program output to files instead.

For example:

```
nmjxv3@rc02xcs213:~$ echo "hello world" > hello.txt
nmjxv3@rc02xcs213:~$ cat hello.txt
hello world
```

Now for a bit about STDERR. Bash numbers its output streams: STDOUT is `1` and STDERR is `2`. If you want to do pipe STDERR to other programs, you need to redirect it to STDOUT first. This is done like so: `2>&1`.

So, for example, if you have a bunch of compiler errors that you want to look through with `less`, you'd do this:

```
g++ lots_o_errors.cpp 2>&1 | less
```

Questions

Name: ______________________________

1. What does a shell do?

2. What command would you use to print the names of all header (`.h`) files in the `/tmp` directory?

3. How would you move a file named "bob.txt" (in your current directory) to a folder in your home directory named "odd" and rename "bob.txt" to "5.txt"?

4. Suppose you have a file containing a bunch of scores, one score per line (like so: "57 Jenna"). How would you print the top three scores from the file?

Quick Reference

`ls [Directory or Files]`: List the contents of a directory or information about files

- `-l` Detailed listing of file details
- `-h` Show human-readable modification times
- `-a` Show hidden files (files whose name starts with `.`)

`pwd`: Print current working directory

`cd [Directory]`: Change current working directory

- `cd` with no arguments changes to the home directory
- `cd -` switches to the previous working directory

`mv [source] [destination]`: Move or rename a file or directory

- `-i`: Interactively prompt before overwriting files
- `-n`: Never overwrite files

`cp [source] [destination]`: Copy a file or directory

- `-r`: Recursively copy directory (must be used to copy directories)
- `-i`: Interactively prompt before overwriting files
- `-n`: Never overwrite files

`rm [file]`: Removes a file or directory

- `-r`: Recursively remove directory (must be used to remove directories)
- `-i`: Interactively prompt before removing files

`mkdir [directory]`: Make a new directory

- `-p`: Make all directories missing in a given path

`cat [filenames]`: Output contents of files

`less [filename]`: Interactively scroll through long files

`head [filename]`: Display lines from beginning of a file

- `-n num_lines`: Display `num_lines` lines, rather than the default of 10 `tail [filename]`: Display lines from the end of a file

- `-n num_lines`: Display `num_lines` lines, rather than the default of 10

`man [command]`: Display manual page for a command

Special Filenames:

- `.`: Current directory
- `..`: Parent directory
- `~`: Home directory
- `/`: Root directory

Glob patterns:

- `*`: Match 0 or more characters of a file or directory name
- `?`: Match exactly 1 character of a file or directory name

IO Redirection:

- `cmd1 | cmd2`: Redirect output from `cmd1` to the input of `cmd2`
- `cmd > filename`: Redirect output from `cmd` into a file
- `cmd 2>&1`: Redirect the error output from `cmd` into its regular output

Further Reading

List of Bash Commands Bash Reference Manual All About Pipes

Chapter 3

Git Basics

Motivation

Close your eyes[1].

Imagine yourself standing in a wide, open field. In that field stands a desk, and on that desk, a computer. You sit down at the desk ready to code up the Next Big Thing[2].

You start programming and find yourself ready to start writing a cool new feature. "I better back up my code," you think to yourself. "Just in case I really goof it up." You create a new folder, name it "old_version" and continue on your way.

As you work and work[3], you find yourself with quite a few of these backups. You see "old_version" and "old_version2" alongside good old "sorta_works" and "almost_done" "Good thing I made these backups", you say. "Better safe than sorry."

Time passes.

"Wait... this isn't right...," you think. Your code is broken! Boy, it's a good thing you kept those backups. But wait... which of these backups actually worked? What's different in *this* version that's breaking your project?

Open your eyes.

If you haven't already experienced this predicament outside of a daydream, you certainly will. It's a fact that as you work on a programming project, you will add features to your code, change the way it works, and sometimes introduce bugs.

[1] Now open them again, because it's hard to read with your eyes shut.
[2] This is your daydream, friend. I have no idea what this program is or does.
[3] Yes, you're daydreaming about work.

Sure, you can manage your projects by making copy after copy and manually combing through hundreds of lines of. . . .

No, don't do that.

To solve this predicament, some smart people have developed different **version control systems**. A version control system is a program whose job is to help you manage versions of your code. In most cases, they **help you take snapshots of your code**, so that you can see how your code changes over time. As a result, you **develop a timeline** of your code's state.

With a timeline of your code's state, your version control system can:

- help you figure out where bugs were introduced
- make it easier to collaborate with other coders
- keep your experimental code away from your stable, working code.
- do much, much more than three things.

In this course, we will be using **Git** as our version control system. Git is powerful and wildly popular in industry. Your experience with Git will undoubtedly be useful throughout your career in Computer Science.

It's also fun, so that's cool.

Takeaways

- Learn what a version control system is, as well as some common features.
- Gain experience adding files to a Git repository and tracking changes to those files over time.
- Learn how to separate work onto separate Git branches.
- Understand the difference between a local and remote repository.

Walkthrough

Git Repositories

When you using Git, you work within a Git **repository**. A repository is essentially a folder for which Git has been tracking the history. We call that folder containing files and history your **local** copy of a repository. We say it's local because it's stored locally – in a place where you can access its contents just like any other folder.

This is the part where I want to compare Git to Dropbox or Google Drive, but this is a dangerous comparison. Realize[4] that Git will feel similar to these services in some ways, but there are many features that make them *very* different.

[4] Using your mind.

When you work with a local Git repository, you will:

- **ask Git to track** of changes to files. Git *does not* automatically track files. You have to tell it to track stuff.
- **ask Git to take snapshots** of the files in your repository. Essentially, instead of copying your code into a folder to back it up, you'll tell Git to take a snapshot instead. Each snapshot represents the state of your repository *at that moment in time.*

Notice that each of these actions require **you** to ask Git to do stuff. Git does not do these things by itself. Because it's not automatic, you have the ability to take snapshots only when it makes sense. For example, it's common to take snapshots whenever you finish a feature or before you start working on experimental code[5].

Trying out GitLab

To backup[6] work stored in a local repository, people often use an online service to store their repositories remotely. In this course, we will be using a campus-hosted service called **GitLab**.

GitLab, like other git hosting services[7], allows you to log into a website to create a **remote repository**. Once created, you can **clone** (or download) your new repository into a **local copy**, so that you can begin to work. An empty repository will contain no files and an empty timeline (with no snapshots).

Try the following to create your own, empty repository on GitLab:

1. Log in to https://git-classes.mst.edu/ using your Single Sign-on credentials.
2. Click the `+` (New Project) button in the upper right to create a new repository on GitLab.
3. Under Project Name, give your project a good name. Let's call it `my-fancy-project`.
    - You can enter a description if you like, or you can leave it blank.
    - Make sure your repository's visibility is set to Private.
4. Click the Create Project button.
5. Welcome to your repository's home page! Don't close it, yet. We'll need to copy some commands from here.

Now that you've created your repository, it's ready for you to start working. Let's try cloning the remote repository into a local repository.

[5] Maybe you're rewriting a function, and you don't know if it'll work. It's convenient to take a snapshot, so that if things go bad, you can always revert back to a working state.

[6] And other things. Remotes are actually *extremely* useful.

[7] GitLab, GitHub, BitBucket, etc.

1. Look for the "Create a new repository" section and copy the command that starts with `git clone https://...my-fancy-project.git`
2. Connect to a campus Linux machine using PuTTY and paste that command in your bash shell.
3. Press enter, and type in your username and password when prompted.
4. Run `ls`. You should see that a folder called `my-fancy-project` was created in your current working directory.
5. Run `cd` to enter your freshly cloned repository.

Nice work!

Now, it's **very important** that you understand the objective of this exercise. You've now seen what it looks like to create a remote repository on GitLab and clone it down into a local repository. If you were working on a real project, the next step would be to create files in your `my-fancy-project` folder, take snapshots of those files, and upload your snapshots to GitLab.

In this course, **you will not have to create any GitLab repositories yourself**. Instead, your instructor will be creating repositories and sharing them with you. The ability to share repositories on GitLab is one of its more powerful features.

Tracking Files

At this point, you now have a (very fancy) local repository called `my-fancy-project`. Currently, your repository has no timeline, and Git is not watching any of the files in it.

Before we get too involved, let's see what's in our repository so far. Try running `ls -a` within `my-fancy-project`

```
$ ls -a
.    ..   .git
```

See that `.git` directory there? That is a hidden directory that Git uses to store your timeline of snapshots and a bunch of other data about your repository. If you delete it, Git will not know what to do with the files in your directory. In other words, deleting the `.git` directory turns a Git repository into a plain old folder.

So don't do that.

An empty repository isn't much use to us. Let's try asking Git to watch some files for us.

Create a very simple Hello World C++ program and name it `hello.cpp`

```
# Let's see what's in here, first...
$ ls -a
.    ..   .git

# Now let's write that Hello World program
$ emacs hello.cpp

# Cool. There it is.
$ ls -a
.    ..   .git  hello.cpp
```

Now, let's use the `git status` command to ask Git for the **status** of the repository.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    hello.cpp

nothing added to commit but untracked files present (use "git add" to track)
```

Git is telling us that it sees a new file `hello.cpp` that is currently **untracked**. This means that Git has never seen this file before, and that Git has not been told to track the changes made to it. Let's use the `git add` command to ask Git to do just that.

```
$ git add hello.cpp

$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   hello.cpp
```

Now, you can see that `hello.cpp` is listed under "Changes to be committed". In Git terminology, we would say that `hello.cpp` is **staged for commit**. In other words, `hello.cpp` is **ready to be included in the next snapshot**.

Whenever you take a snapshot, Git will only include the changes that are staged. By staging changes for commit, you're essentially picking and choosing what you want to include.

Taking a Snapshot

Although "snapshot" is a convenient term, the real Git term is **commit**. That is, a Git repository timeline is comprised of a series of **commits**.

Now that `hello.cpp` is staged for commit, let's try committing it.

First, let's see what `git status` says

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   hello.cpp
```

OK, that looks good.

Let's also take a look at our current timeline of commits. We'll use `git log` to ask Git to show us our current history.

```
$ git log
fatal: your current branch 'master' does not have any commits yet
```

Remember that, so far, all we've done is clone an empty repository from GitLab and *stage* a new file for commit. It makes sense that we don't see any commits in our history yet.

Before we commit our changes, we need to tell Git who we are. If we don't do this first, Git will refuse to commit anything for us!

```
# Please use your first and last name for the sake of grading.
$ git config --global user.name "<your_name>"

# Please use your university email address, again, for the sake of grading.
$ git config --global user.email "<your_email>"

# Let's also tell Git which text editor you prefer to use.
# You will need to choose a console editor such as jpico, emacs, or vim.
$ git config --global core.editor <editor_command>
```

Now we can finally commit our changes using the `git commit` command.

```
# It's always a good idea to run `git status` before running `git commit`
# just so we can see what we're including in our commit.
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   hello.cpp

# That looks good, so let's commit it!
$ git commit
```

Git will pop open an editor for you. You **must** include a commit message here in order to commit. Simply enter a meaningful message (like `Add hello.cpp`), save the message, and exit the text editor.

Make sure your message is meaningful! If you use garbage commit messages[8], you will only hurt your future self and your grade.

Let's see what our repository status looks like now.

```
$ git status
On branch master
nothing to commit, working tree clean
```

Git is telling us that nothing has changed since the last commit. That makes sense! We added `hello.cpp`, committed it, and we haven't changed anything since that commit.

What about the log?

```
$ git log
commit 648203a12a0b8ab1e0e37336d891b0420994739d (HEAD -> master)
Author: Homer Simpson <simpsonh@lardlad.donuts>
Date:   Mon Jun 12 12:38:01 2017 -0500

    Add hello.cpp
```

That's great! Our timeline now contains one commit: the commit that added `hello.cpp`. Over time, you will commit more and more changes, building up a longer and longer timeline of commits.

[8] Such as "asdf", "stuff", "work", or "finished the lab".

Reading a Status Report

Let's talk in more detail about `git status`.

A file in a Git repository can be in one of **four** states:

- **Unchanged**: Git is tracking this file, but the file looks exactly the same as it did as of the latest commit.
- **Modified**: Git is tracking this file, and the file *has changed* since the last commit.
  - **Not staged**: The changes to this file **will not** be included if you try to commit them with `git commit`.
  - **Staged**: The changes to this file **will** be included if you try to commit them with `git commit`.
- **Untracked**: Git **is not** tracking this file at all. It doesn't know if/how it has changed since the last commit.

So, what's the big deal?

Every time you get ready to run `git commit`, you should make sure you are committing what you want to commit. If you forget to stage changes, Git **will not include them** in your commit!

How do you stage changes to files? Use `git add`. Even if a file is not new, you will need to stage its changes for commit using `git add`.

Uploading to GitLab

Alrighty.

Here you are with your fancy repository. `git status` says that there's nothing new since the last commit. `git log` says that there's one commit in the history.

If you visit the webpage for `my-fancy-project` on GitLab, you'll notice that there's still nothing up there. We need to **push** our new commit to GitLab first.

```
# Enter your Single Sign-on credentials when prompted
$ git push
```

Since we cloned the repository from GitLab earlier, Git assumes that we want to push our changes back to the same place. If you refresh the project page for `my-fancy-project` on GitLab, you should see `hello.cpp` up there!

Take some time to explore your remote repository on GitLab.

Oh, Fork.

Close your eyes again[9].

Here you are working on that Next Big Thing again. As you code, you see a beautiful person emerge[10] from the field's tall grass.

"I am a muse," they say. "Your program is terrible."

You grimace.

Figure 3.1: Grimace

The muse proceeds to explain in great detail how your program can be so much better than it is. You agree. This is a muse after all. Inspiration is their job[11].

Now here's your predicament. The changes proposed by the muse are going to require you to *totally* rework your program. Meanwhile, you need to continue to fix bugs in your existing program to keep your customers happy.

You have two choices:

- Buck up and commit to redoing your entire project, leaving your customers grumpy about the bugs you need to fix.
- Ignore the idea from the muse and fix continue the bugs, throwing the muse's loud, awe-inspiring ideas in the garbage.

[9] Maybe just half-closed this time.
[10] It's as though they were laying there in the grass the whole time. So weird.
[11] You later realize that the muse was just Tony Robbins getting you super amped about everything.

Open your eyes.

Alright, so I lied to you. There are a couple more choices actually.

- You *could* copy your entire local repository into a second one and name it `muse_version`, but that sounds like a bad idea. Soon we'll end up with the same woes we had when we were copying our code into new folders to back it up.
- You *could* let Git manage two parallel lines of development.

That second option sounds **much** better.

One of Git's most powerful features is its ability to **branch** your code's timeline. No, it's not like Primer[12]. You're not going to have separate crazy timelines going back and forth and every which way.

It's more like having parallel universes. Your original universe (branch) is called `master`.

You tell Git to branch at a specific commit, and from there on out, what happens on that branch is separate from other branches. In other words, you enter an alternate universe, and all changes only affect the alternate universe, not the original (`master`) universe.

Now that's dandy, but let's say that we're happy with our experimental branch. How can we integrate that back into the branch of stable code (`master`) again? Well, Git has the ability to **merge** one branch into another. When you merge two branches together, Git will figure out what's different between the two branches, and copy the important stuff from your experimental branch into your stable branch (`master`).

Branching is incredibly useful. Here are just a handful of cases where it comes in handy:

- You want to keep experimental code away from stable, working code.
- You want to keep your work separate from your teammates' code.
- You want to keep your commit history clean by clearly showing where new features were added.

Now, here you are in real life sitting in your leather chair[13], smoking a pipe[14], sipping on bourbon[15], and wondering what commands you use to actually work with branches in Git. It's time to get your hands dirty.

[12] Although you should *absolutely* see it if you haven't. Who doesn't love a good indie time travel movie?

[13] I assume.

[14] I reckon.

[15] That's a given.

Instead of asking you to create a bunch of commits and branches by hands, we're going to use an online tool to work with branches. It's a game, actually.

Pop open a browser and pull up http://learngitbranching.js.org/ Then, work through the following exercises:

- 1.1: Introduction to Git Commits
- 1.2: Branching in Git
- 1.3: Merging in Git

Be sure to read the stuff that pops up! This is a *very* good learning resource.

Your Git Workflow

Your workflow will be something like this:

1. Create/Change files in your repository.
2. Use `git add` to stage changes for commit.
3. Use `git status` to check that the right changes are staged.
4. Use `git commit` to commit your changes.
5. Use `git push` to push your new commits up to GitLab.
6. View your repository on GitLab to ensure that everything looks right.
7. Repeat steps 1 through 6 as necessary.

You don't have to check GitLab every time you push, but it is **highly** recommended that you check your project before it's due. It is easy to forget to push your code before the deadline. Don't lose points for something so simple.

Questions

Name: ______________________________

1. What is the **full** command you ran to clone your `my-fancy-project` repository? (Note, we **don't** want your username/password… we just want the command.)

2. View your `hello.cpp` file on GitLab. Notice that the lines are numbered on the left side of your code. Click on the `3` for line 3.

   a. What happens to that line of code?

   b. Copy the URL for the page and paste it in a new browser tab. What does that link point to?

3. What is the series of commands you used to get through Level 1.3 of http://learngitbranching.js.org/ ? Hint: It's possible to do it in 5 commands.

Quick Reference

`git add`

- Stages new, untracked files for commit
- Stages modified files for commit

`git commit`

- Creates a new commit (snapshot) on your commit timeline
- Opens an editor and requires that you enter a log message

`git status`

- Allows you to check the status of your repository
- Shows which branch you are currently on.
- Files can be **untracked**, **unstaged**, **staged**, or **unchanged**
- It's a good practice to check the status of your repository before you commit.

`git log`

- Shows you a list of commits in your repository

`git push`

- Pushes new **commits** to from a local repository to a remote repository.
- You cannot push files, you can only push commits.

`git checkout`

- Can be used to check out different branches.
- Can be used to *create* a new branch.
- Can be used (with great caution!) to check out specific commits.

`git branch`

- Can be used to create or delete branches.
- Can be used to list all local and remote branches.

`git merge`

- Merges two branches together.

Further Reading

- The Git Book
- Git Branching Tutorial
- GitHub's Git Tutorial

Chapter 4

Bash Scripting

Motivation

In addition to being a fully-functional[1] interactive shell, Bash can also run commands from a text file (known as a 'shell script'). It even includes conditionals and loops! These scripts are the duct tape and bailing wire of computer programming – great for connecting other programs together. Use shell scripts to write one-off tools for odd jobs, to build utilities that make your life easier, and to customize your shell.

Note: There's nothing special about the contents of a shell script – everything you learn in this lab you could type into the Bash prompt itself.

Takeaways

- Learn to glue programs together into shell scripts
- Gain more experience working with output redirection in bash

Walkthrough

Here's a quick example of what a shell script looks like:

```
#!/bin/bash

g++ *.cpp
./a.out
```

[1] Disclaimer: Bash may be neither full nor functional for your use case. Consult your primary care physician to see if Bash is right for you.

This script compiles all the C++ files in the current directory, then runs the resulting executable. To run it, put it in a file named, say, `runit1.sh`, then type `./runit1.sh`[2] at your shell prompt.

Two things to note: 1. The first line, called a "shebang"[3], tells Bash what program to run the script through. In this case, it's a Bash script. 2. The rest of the file is a sequence of commands, one per line, just as you would type them into the shell.

Variables

Declaring and Using

There's no special keyword for declaring variables; you just define what you want them to be. When you use them, you must prefix the variable name with a `$`:

```
#!/bin/bash

COW="big"

echo "$COW"
```

Note: It is *very* important that you not put any spaces around the `=` when assigning to variables in Bash. Otherwise, Bash gets very confused and scared, as we all do when encountering something unfamiliar. If this happens, gently pet its nose until it calms down, then take the spaces out and try again.

Variables can hold strings or numbers. Bash is dynamically typed, so there's no need to specify `int` or `string`; Bash just works out what you (probably) want on its own.

It is traditional to name variables in uppercase, but by no means required. Judicious use of caps lock can help keep the attention of a distractible Bash instance.

Special Variables

Bash provides numerous special variables that come in handy when working with programs.

To determine whether a command succeeded or failed, you can check the `$?` variable, which contains the return value[4] of the last command run. Traditionally,

[2] `.` is shorthand for the current directory, so this tells bash to look in the current directory for a file named `runit1.sh` and execute that file. We'll talk more about why you have to write this later on.

[3] A combination of "sharp" (#) and "bang" (!).

[4] This is the very same value as what you return from `int main()` in a C++ program!

a value of `0` indicates success, and a non-zero value indicates failure. Some programs may use different return values to indicate different types of failures; consult the man page for a program to see how it behaves.

For example, if you run `g++` on a file that doesn't exist, g++ returns `1`:

```
nmjxv3@rc02xcs213:~$ g++ no-such-file.cpp
g++: error: no-such-file.cpp: No such file or directory
g++: fatal error: no input files
compilation terminated.
nmjxv3@rc02xcs213:~$ echo $?
1
```

Bash also provides variables holding the command-line arguments passed to the script. A command-line argument is something that you type after the command; for instance, in the command `ls /tmp`, `/tmp` is the first argument passed to `ls`. The name of the command that started the script is stored in `$0`. This is almost always just the name of the script[5]. The variables `$1` through `$9` contain the first through ninth command line arguments, respectively. To get the 10th argument, you have to write `${10}`, and likewise for higher argument numbers.

The array `$@` contains all the arguments except `$0`; this is commonly used for looping over all arguments passed to a command. The number of arguments is stored in `$#`.

Whitespace Gotchas

Bash is very eager to split up input on spaces. Normally this is what you want – `cat foo bar` should print out the contents of two files named "foo" and "bar", rather than trying to find one file named "foo bar". But sometimes, like when your cat catches that mouse in your basement but then brings it to you rather than tossing it over the neighbor's fence like a good pal, Bash goes a little too far with the space splitting.

If you wanted to make a file named "cool program.cpp" and compile it with `g++`, you'd need to put double quotes around the name: `g++ "cool program.cpp"`. Likewise, when scripting, if you don't want a variable to be space split, surround it with double quotes. So as a rule, rather than `$1`, use `"$1"`, and iterate over `"$@"` rather than `$@`.

Example

We can spiff up our `runit1.sh` example to allow the user to set the name of the executable to be produced:

[5] If you must know, the other possibility is that it is started through a link (either a hard link or a symbolic link) to the script. In this case, `$0` is the name of the link instead. Any way you slice it, `$0` contains what the user typed in order to execute your script.

```
#!/bin/bash

g++ *.cpp -o "$1"
./"$1"
```

You'd run this one something like `./runit2.sh program_name`.

Conditionals

If statements

The `if` statement in Bash runs a program[6] and checks the return value. If the command succeeds (i.e., returns 0), the body of the if statement is executed.

Bash provides some handy commands for writing common conditional epxressions: `[ ]` is shorthand for the `test` command, and `[[ ]]` is a Bash builtin. `[ ]` works on shells other than Bash, but `[[ ]]` is far less confusing[7].

Here's an example of how to write `if` statements in Bash:

```
#!/bin/bash

# Emit the appropriate greeting for various people

if [[ $1 = "Jeff" ]]; then
    echo "Hi, Jeff"
elif [[ $1 == "Maggie" ]]; then
    echo "Hello, Maggie"
elif [[ $1 == *.txt ]]; then
    echo "You're a text file, $1"
elif [ "$1" = "Stallman" ]; then
    echo "FREEDOM!"
else
    echo "Who in blazes are you?"
fi
```

Be careful not to forget the semicolon after the condition or the `fi` at the end of the if statement.

[6] Or a builtin shell command (see `man bash` for details).

[7] If you're writing scripts for yourself and your friends, using `[[ ]]`is a-ok; the only case you'd care about using `[ ]` is if you're writing scripts that have to run on a lot of different machines. In this book, we'll use `[[ ]]` because it has fewer gotchas.

Writing conditionals with `[[ ]]`

Since Bash is dynamically typed, `[[ ]]` has one set of operators for comparing strings and another set for comparing numbers. That way, you can specify which type of comparison to use, rather than hoping that Bash guesses right[8].

Comparing Strings:

- `=`,`==`: Either
  - String equality, if both operands are strings, or
  - Pattern (glob) matching, if the RHS is a glob.
- `!=`: Either
  - String ineqaulity, if both operands are strings, or
  - Glob fails to match, if the RHS is a glob.
- `<`: The LHS sorts before the RHS.
- `>`: The LHS sorts after the RHS.
- `-z`: The string is empty (length is zero).
- `-n`: The string is not empty (e.g., `[[ -n "$var" ]]`).

Comparing Numbers:

(These are all meant to be used infix, like `[[ $num -eq 5 ]]`.)

- `-eq`: Numeric equality
- `-ne`: Numeric inequality.
- `-lt`: Less than
- `-gt`: Greater than
- `-le`: Less than or equal to
- `-ge`: Greater than or equal to

Checking Attributes of Files:

(Use these like `[[ -e story.txt ]]`.)

- `-e`: True if the file exists
- `-f`: True if the file is a regular file
- `-d`: True if the file is a directory

There are a number of other file checks that you can perform; they are listed in the Bash manual.

Boolean Logic:

[8] If you know some JavaScript you might be familiar with the problem of too-permissive operators: in JS, `"4" + 1 == "41"`, but `"4" - 1 == 3`.

- `&&`: Logical AND
- `||`: Logical OR
- `!`: Logical NOT

You can also group statements using parentheses:

```
#!/bin/bash

num=5

if [[ ($num -lt 3) && ("story.txt" == *.txt) ]]; then
  echo "Hello, text file!"
fi
```

Writing conditionals with `(( ))`

`(( ))` is used for arithmetic, but it can also be used to do numeric comparisons in the more familiar C style.

- `>`/`>=`: Greater than/Greater than or equal
- `<`/`<=`: Less than/Less than or equal
- `==`/`!=`: Equality/inequality

When working with `(( ))`, you do not need to prefix variable names with `$`:

```
#!/bin/bash

x=5
y=7

if (( x < y )); then
  echo "Hello there"
fi
```

Case statements

Case statements in Bash work similar to the `==` operator for `[[ ]]`; you can make cases for strings and globs.

Here is an example case statement:

```
#!/bin/bash
```

```
case $1 in
    a)
        echo "a, literally"
        ;;
    b*)
        echo "Something that starts with b"
        ;;
    *c)
        echo "Something that ends with c"
        ;;
    "*d")
        echo "*d, literally"
        ;;
    *)
        echo "Anything"
        ;;
esac
```

Do not forget the double semicolon at the end of each case – `;;` is *required* to end a case. And, as with `if`, `case` statements end with `esac`.

Example

We can use conditional statements to spiff up our previous `runit2.sh` script. This example demonstrates numeric comparison using both `(( ))` and `[[ ]]`.

```
#!/bin/bash

if (( $# > 0 )); then
    g++ *.cpp -o "$1"
    exe="$1"
else
    g++ *.cpp
    exe=a.out
fi

if [[ $? -eq 0 ]]; then
    ./"$exe"
fi
```

Can you make this example even spiffier using file attribute checks?

Arithmetic

`(( ))` performs arithmetic; the syntax is pretty much borrowed from C. Inside `(( ))`, you do not need to prefix variable names with `$`!

For example,

```
#!/bin/bash

x=5
y=7
(( sum = x +  y ))
echo $sum
```

Operator names follow those in C; `(( ))` supports arithmetic, bitwise, and logical operators. One difference is that `**` does exponentiation. See the Bash manual for an exhaustive list of operators.

Looping

For Loops

Bash for loops typically follow a pattern of looping over the contents of an array (or array-ish thing).

For (heh) example, you can print out the names of all `.sh` files in the current directory like so:

```
#!/bin/bash

for file in *.sh; do
    echo $file
done
```

Or sum all command-line arguments:

```
#!/bin/bash

sum=0

for arg in "$@"; do
  (( sum += arg ))
done

echo $sum
```

If you need a counting for loop (C-style loop), you can get one of those with `((
))`:

```
#!/bin/bash

for (( i=1; i < 9; i++ )); do
    echo $i;
done
```

With for loops, do not forget the semicolon after the condition. The body of the loop is enclosed beween the `do` and `done` keywords (sorry, no `rof` for you!).

While Loops

Bash also has while loops, but no do-while loops. As with for loops, the loop body is enclosed between `do` and `done`. Any conditional you'd use with an if statement should also work with a while loop.

For example,

```
#!/bin/bash

input=""
while [[ $input != "4" ]]; do
    echo "Please guess the random number: "
    read input
done
```

This example uses the `read` command, which is built in to Bash, to read a line of input from the user (i.e., STDIN). `read` takes one argument: the name of a variable to read the line into.

"Functions"

Bash functions are better thought of as small programs, rather than functions in the typical programming sense. They are called the same way as commands, and inside a function, its arguments are available in `$1`, `$2`, etc. Furthermore, they can only return an error code; "returning" other values requires some level of trickery.

Here's a simple function example:

```
#!/bin/bash
parrot() {
```

```
    while (( $# > 0 )); do
        echo "$1"
        shift
    done
}

parrot These are "several arguments"
```

(Note that `shift` throws away the first argument and shifts all the remaining arguments down one.)

To return something, the easiest solution is to `echo` it and have the caller catch the value:

```
#!/bin/bash

average() {
    sum=0
    for num in "$@"; do
        (( sum += num ))
    done

    (( avg = sum / $# ))
    echo $avg
}

my_average=$(average 1 2 3 4)

echo $my_average
```

Here, `my_average=$(average 1 2 3 4)` calls `average` with the arguments `1 2 3 4` and stores the STDOUT of `average` in the variable `my_average`.

Tips

To write a literal `\`, `‘`, `$`, `"`, `’`, `#`, escape it with `\`; for instance, `"\$"` gives a literal `$`.

When writing scripts, sometimes you will want to change directories, for instance if you want to write some temporary files in `/tmp`. Rather than using `cd` and keeping track of where you were so you can `cd` back later, use `pushd` and `popd`. `pushd` pushes a new directory onto the directories stack and `popd` removes a directory from this stack. Use `dirs` to print out the stack.

For instance, suppose you start in `~/cool_code`. `pushd /tmp` changes the current directory to `/tmp`. Calling `popd` then removes `/tmp` from the stack and changes to the next directory in the stack, which is `~/cool_code`.

Putting `set -u` at the top of your script will give you an error if you try to use a variable without setting it first. This is particularly handy if you make a typo; for example, `rm -r $delete_mee/*` will call `rm -r /*` if you haven't set `$delete_mee`!

Bash contains a help system for its built-in commands: `help pushd` tells you information about the `pushd` command.

Questions

Name: ______________________________

1. What does the `let` builtin do?

2. Write a script that prints "fizz" if the first argument is divisible by 3, "buzz" if it is divisible by 5, and "fizzbuzz" if it is divisible by both 3 and 5.[9]

3. Write a script that prints "directory" if the first argument is a directory and "file" if the first argument is a file.

[9] Also, why do so many people ask this as an interview question!?

Quick Reference

Further Reading

- Bash Manual
- Bash Guide
- Bash Tutorial

Chapter 5

Regular Expressions

Motivation

Regular expressions describe patterns in strings. They're incredibly useful for parsing input to programs. Need to pull the digits out of a phone number? Find a particular entry in a several-megabyte log file? Regex has got you covered! You can even use regular expressions to transform one string to another.

In your theory of computer science class, you will learn about what makes a regular expression regular.[1] Because nobody pays attention in theory classes, most regular expression libraries are not actually 'regular' in the theoretical sense. That's fine, though; irregular expressions can describe patterns that strictly regular expressions cannot.[2] These regular expressions are usually called 'extended regular expressions'.

When most developers say 'regex', they're thinking of Perl Compatible Regular Expressions (PCRE), but there are several other flavors of regular expressions.[3] In this chapter we will cover the flavors used by common Linux utilities; they are nearly the same as PCRE but have some minor differences. In addition to the utilities we will discuss in this chapter, nearly every programming language (even C++) has a regular expressions library.

Takeaways

- Learn the syntax for writing regular expressions

[1] If it weren't for Noam Chomsky, we'd only have irregular expressions like "every boat is a bob".

[2] With one caveat: irregular expressions can be very slow to check; regular regular expressions can always be checked quickly. (Whether your regex library actually checks quickly is another story for another time, because I can see you nodding off right now.)

[3] The umami flavors are my favorite.

Figure 5.1: https://xkcd.com/208/

- Use `grep` to search files using regular expressions
- Use `sed` to search and edit files using regular expressions

Walkthrough

The general idea of writing a regular expression is that you're writing a *search string*. They may look complicated, but break them down piece by piece and you should be able to puzzle out what is going on.

There are several websites that will visually show you what a regular expression matches. We recommend you try out examples from this chapter in one of these websites; try https://regex101.com/.

Syntax

Letters, numbers, and spaces match themselves: the regex `abc` matches the string "abc". In addition to literal character matches, there are several single-character-long patterns:

- `.`: Matches one of any character.
- `\w`: Matches a word character (letters, numbers, and _).
- `\W`: Matches everything `\w` doesn't.
- `\d`: Matches a digit.
- `\D`: Matches anything that isn't a digit.
- `\s`: Matches whitespace (space, tab, newline, carriage return, etc.).
- `\S`: Matches non-whitespace (everything `\s` doesn't match).

So `a\wb` matches "aab", "a2b", and so on.

`\` is also the escape character, so `\\` matches "\".

If these character patterns don't quite meet your needs, you can make your own by listing the possible matches between `[]`s. So if we wanted to match "abc" and "adc" and nothing else, we could write `a[bd]c`.

Custom character classes can include other character classes, and you can use `-` to indicate a range of characters. For instance, if you wanted to match a hexadecimal digit, you could write the following: `[\da-fA-F]` to match a digit (`\d`) or a hex letter, either uppercase or lowercase. You can also negate character classes by including a `^` at the beginning. `[^a-z]` matches everything except lowercase letters.

Now, if you want to match names, you can use `\w\w\w\w` to match "Finn" or "Jake", but that won't work to match "Bob" or "Summer". What you really need is a variable-length match. Fortunately there are several of these!

- `{n}`: matches n of the previous character class.
- `{n,m}`: matches between n and m of the previous character class (inclusive).
- `{n,}`: matches at least n of the previous character.

So you could write `/\w{4}/` to match four-letter words, or `\w{1,}` to match one or more word characters.

Because some of these patterns are so common, there's shorthand for them:

- `*`: matches 0 or more of the previous character; short for `{0,}`.
- `+`: matches 1 or more of the previous character; short for `{1,}`.
- `?`: matches 0 or 1 of the previous character; short for `{0,1}`.

So we could write our name regex as `/\w+/`.

More examples:

- `0x[a-fA-F\d]+`: Matches a hexadecimal number (`0xdeadbeef`, `0x1337c0de`, etc.).
- `a+b+` : Matches any string containing one or more `a`s, followed by one or more `b`s.
- `\d{5}` : Matches any string containing five digits (a regular ZIP code).
- `\d{5}-\d{4}` : Matches any string containing 5 digits followed by a dash and 4 more digits (a ZIP+4 code).

What if you wanted to match a ZIP code either with or without the extension? It's tempting to write `\d{5}-?\d{0,4}`, but this would also match "12345-", "12345-6", and so on, which are not valid ZIP+4 codes.

What we really need is a way to group parts of the match together. Fortunately, you can do this with `()`s! `\d{5}(-\d{4})?` matches any ZIP code with an optional +4 extension.

A group can match one of several options, denoted by `|`. For example, `[ac][bd]` matches "ab", "cd", "ad", and "cb". To match "ab" or "cd" but not "ad" or "cb", use `(ab|cd)`.

The real power of groups is in backreferences, which come in handy both when matching expressions and doing string transformations. You can refer to the substring matched by the first group with `\1`, the second group with `\2`, etc. We can match "abab" or "cdcd" but not "abcd" or "cdab" with `(ab|cd)\1`.

If you have a pattern where you need to refer to both a backreference and a digit immediately afterward, use an empty group to separate the backreference and digit. For example, let's say you want to match "110", "220", . . . , "990". If you wrote `(\d)\10`, your regex engine would be confused because `\10` looks like a backreference to the 10th group. Instead, write `(\d)\1()0` – the `()` matches an empty string (i.e. nothing), so it's as if it wasn't there.

By default, regular expressions match a substring anywhere in the string. So if you have the regex `a+b+` and the string "cccaabbddddd", that will count as a match because `a+b+` matches "aabb". To specify that a match must start at the beginning of a line, use `^`, and to specify that the match ends at the end of a line, use `$`. So, `a+b+$` matches "cccaabb" but not "aabbcc", and `^a+b+$` matches only lines containing some "a"s followed by some "b"s.

Now, it's the nature of regular expressions to be greedy and gobble up (match) as much as they can. Usually this sort of self-interested behavior is fine, but sometimes it goes too far.[4] You can use `?` on part of a regular expression to make that part polite (i.e. non-greedy), in which case it matches only as much as it needs for the whole regex to match.

One example of this is if you are trying to match complete sentences from lines of text. Using `(.+\.)` (i.e. match one or more things, followed by a period) is fine, as long as there is just one sentence per line. But if there's more than one sentence on a line, this regex will match all of them, because `.` matches "."! If you want it to match one and only one sentence, you have to tell the `.+` to match only as much as needed, so `(.+?\.)`.

Alternatively, you could rewrite it using a custom character class: `([^\.]+\.)` – match one or more things that aren't a period, followed by a period.

`grep`

Imagine that you have sat yourself down at your computer. It's 1984 and you dial in to your local BBS on your brand new 9.6 kbps modem. Your stomach growls. As your modem begins its handshake, you stand up, suddenly aware that you must have nachos. You fetch the tortilla chips and cheese and heat them in the microwave next to the phone jack. You sit back down at your machine. Your terminal is filling with lines of junk, `!#@$!%^IA(jfio2q4Tj1$T(!34u17f143$#` over and over and over. Dang it, the microwave is interfering with the phone line. You lean back, close your eyes, and listen to the cheese sizzling.

Your reverie is cut short when you suddenly remember that you have a big file that you really need to find some stuff in. *GREP!* If only there was some program that could use that line noise from your nachos to help...

Okay, enough imagining. There *is* a command to use that line noise to look through files: `grep`. This interjection of a command name is short for "global regular expression print", and it does exactly just that. In this case, "just that" means it prints strings from files (or standard in) that match a given regular expression. If you want to look for "bob" in "cool_people.txt", you could do it with `grep` like so: `grep bob cool\_people.txt`. If you don't specify a filename, `grep` reads from standard input, so you can pipe stuff into it as well.

`grep` has a few handy options:

[4] POLITICS!

- `-C LINES`: Give `LINES` lines of context around the match.
- `-v`: Print every line that doesn't match (it inverts the match).
- `-i`: Ignore case when matching.
- `-P`: Use Perl-style regular expressions.
- `-o`: Only print the part of the line the regex matches. Handy for figuring out what a regex is matching.

Without Perl-style regexes, `grep` requires you to escape special characters to get the special meaning.[5] In other words, `a+` matches "a+", whereas `a\+` matches one or more "a"s.

For these examples, we'll use STDIN as our search text. That is, grep will use the pattern (passed as an argument) to search the input received over STDIN.

```
$ echo "bananas" | grep 'b\(an\)\+as'
bananas
$ echo "bananananananananas" | grep 'b\(an\)\+as'
bananananananananas
$ echo "banas" | grep 'b\(an\)\+as'
banas
$ echo "banana" | grep 'b\(an\)\+as'
```

`sed`

`grep` is great and all but it's really only for printing out matches of regular expressions. We can do so much more with regular expressions, though! `sed` is a 'stream editor': it reads in a file (or standard in), makes edits, and prints the edited stream to standard out. `sed` is noninteractive; while you *can* use it to perform any old edit, it's best for situations where you want to automate editing.

Some handy `sed` flags:

- `-r`: Use extended regular expressions. **NOTE**: even with extended regexes, `sed` is missing some character classes, such as `\d`.
- `-n`: Only print lines that match (handy for debugging).

`sed` has several commands that you can use in conjunction with reglar expressions to perform edits. One such command is the print command, `p`. It prints every line that a particular regex matches. `sed -n '/REGEX/ p'` works almost exactly like `grep REGEX` does. Use this command to make sure your regexes match what you think they should.

[5] You may think this actually makes some sense and that PCRE is needlessly confusing. You may even feel slightly despondent as you realize that a piece of software being popular doesn't mean that it's good. That's what you get for thinking.

The substitute command, `s`, substitutes the string matched by a regular expression with another string. `sed 's/REGEX/REPLACEMENT/'` replaces the match for `REGEX` with 'REPLACEMENT. This lets you perform string transformations, or edits.

For example,

```
$ echo "bananas" | sed -r 's/(an)+/uen/'
buenas
```

You can use backreferences in your replacement strings!

```
$ echo "ab" | sed -r 's/(ab|cd)/First group matched \1/'
First group matched ab
```

The substitute command has a few options. The global option, `g`, applies the command to every match on a line, rather than just the first:

```
$ echo "ab ab" | sed 's/ab/bob/'
bob ab
$ echo "ab ab" | sed 's/ab/bob/g'
bob bob
```

The `i` option makes the match case insensitive, like `grep`'s `-i` flag.

```
$ echo "APPLES ARE GOOD" | sed 's/apple/banana/i'
bananaS ARE GOOD
```

Finally, you can combine the substitute and print commands:

```
$ echo -e "apple\nbanana\napple pie" | sed -n 's/apple/grape/ p'
grape
grape pie
```

There are even more `sed` commands, and more ways to combine them together. Fortunately for you, though, this is not a book on `sed`, so we'll leave it at that. It's definitely worthwhile to spend a bit of time looking through the `sed` manual if you find yourself needing to do something it's good for.

Questions

Name: ______________________________

1. Suppose, for the sake of simplicity[6], that we want to match email addresses whose addresses and domains are letters and numbers, like "abc123@xyz.wibble". Write a regular expression to match an email address.

2. Write a command to check if Clayton Price is in "cool_nerds.txt", a list of cool nerds.

3. Imagine that you are the owner of Pat's Pizza Pie Pizzaria, a pizza joint that's fallen on tough times. You're trying to reinvent the business as a hip, fancy eatery, "The Garden of Olives (And Also Peperoncinis)". As part of this reinvention, you need to jazz up that menu by replacing "pizza" with "foccacia and fresh tomato sauce". Suppose your menu is stored in "menu.txt". Write a command to update every instance of "pizza" and place the new, hip menu in "carte-du-jour.txt".

[6] In practice, email addresses can have all sorts of things in them! Like spaces! Or quotes!

Quick Reference

Regex:

- `.`: Matches one of any character.
- `\w`: Matches a word character (letters, numbers, and _).
- `\W`: Matches everything `\w` doesn't.
- `\d`: Matches a digit.
- `\D`: Matches anything that isn't a digit.
- `\s`: Matches whitespace (space, tab, newline, carriage return, etc.).
- `\S`: Matches non-whitespace (everything `\s` doesn't match).

- `{n}`: matches *n* of the previous character class.
- `{n,m}`: matches between *n* and *m* of the previous character class (inclusive).
- `{n,}`: matches at least *n* of the previous character.
- `\*`: matches 0 or more of the previous character; short for `{0,}`.
- `+`: matches 1 or more of the previous character; short for `{1,}`.
- `?`: matches 0 or 1 of the previous character; short for `{0,1}`.

`grep REGEX [FILE]`: Search for `REGEX` in `FILE`, or standard input if no file is specified

- `-C LINES`: Give `LINES` lines of context around the match.
- `-v`: Print every line that doesn't match (it inverts the match).
- `-i`: Ignore case when matching.
- `-P`: Use Perl-style regular expressions.
- `-o`: Only print the part of the line the regex matches.

`sed COMMANDS [FILE]`: Perform `COMMANDS` to the contents of `FILE`, or standard input if no file is specified, and print the results to standard output

- `-r`: Use extended regular expressions.
- `-n`: Only print lines that match.

- `/REGEX/ p`: Print lines that match `REGEX`
- `s/REGEX/REPLACEMENT/`: Replace strings that match `REGEX` with `REPLACEMENT` -g`: Replace every match on each line, rather than just the first match `-i‘: Make matches case insensitive

Further Reading

- Regex reference

- Regex Crossword Puzzles
- `grep` manual
- `sed` manual
- `sed` tutorial
- C++ regex library reference

Chapter 6

Integrated Development Environments

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 7

Building with Make

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 8

Debugging with GDB

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 9

Locating memory leaks with Memcheck

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 10

Profiling

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 11

Unit testing with Boost Unit Test Framework

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 12

Using C++11 and the Standard Template Library

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 13

Graphical User Interfaces with Qt

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading

Chapter 14

Typesetting with LaTeX

Motivation

Takeaways

Walkthrough

Questions

Quick Reference

Further Reading