

Prelab Thinger

Homer Simpson



# Contents

<b>1</b>	<b>Exploring Text Editors</b>	<b>7</b>
	Motivation . . . . .	7
	Takeaways . . . . .	8
	Walkthrough . . . . .	8
	Notepad++ . . . . .	8
	Atom . . . . .	13
	JPico . . . . .	17
	Emacs . . . . .	22
	Vim . . . . .	28
	Questions . . . . .	33
	Quick Reference . . . . .	35
	Further Reading . . . . .	35
	Notepad++ . . . . .	35
	JPico . . . . .	36
	Vim . . . . .	36
<b>2</b>	<b>Bash Basics</b>	<b>37</b>
	Motivation . . . . .	37
	Takeaways . . . . .	38
	Walkthrough . . . . .	38
	My Dinner with Bash . . . . .	38
	Filesystem Navigation . . . . .	40
	Shorthand . . . . .	41

Rearranging Files . . . . .	42
Looking at Files . . . . .	42
The Manual . . . . .	43
I/O Redirection . . . . .	43
Questions . . . . .	46
Quick Reference . . . . .	47
Further Reading . . . . .	48
<b>3 Git Basics</b>	<b>49</b>
<b>4 Bash Scripting</b>	<b>51</b>
<b>5 Regular Expressions</b>	<b>53</b>
<b>6 Integrated Development Environments</b>	<b>55</b>
<b>7 Building with Make</b>	<b>57</b>
<b>8 Debugging with GDB</b>	<b>59</b>
<b>9 Locating memory leaks with Memcheck</b>	<b>61</b>
<b>10 Profiling</b>	<b>63</b>
<b>11 Unit testing with Boost Unit Test Framework</b>	<b>65</b>
<b>12 Using C++11 and the Standard Template Library</b>	<b>67</b>
<b>13 Graphical User Interfaces with Qt</b>	<b>69</b>
<b>14 Typesetting with LaTeX</b>	<b>71</b>

# 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<sup>1</sup> are written in plain ASCII text. ASCII is a convenient format for humans and programs<sup>2</sup> 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. 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.

---

<sup>1</sup>And many other programming languages, for that matter.

<sup>2</sup>Including compilers.

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<sup>3</sup>!

**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++**<sup>4</sup> 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.

---

<sup>3</sup>Thanks to cloud computing, this is becoming commonplace, yo.

<sup>4</sup>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<sup>5</sup> 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** + **I**: 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 current 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.

---

<sup>5</sup> *Outdent* (verb). Latin: To remove a tooth; English: The opposite of indent.

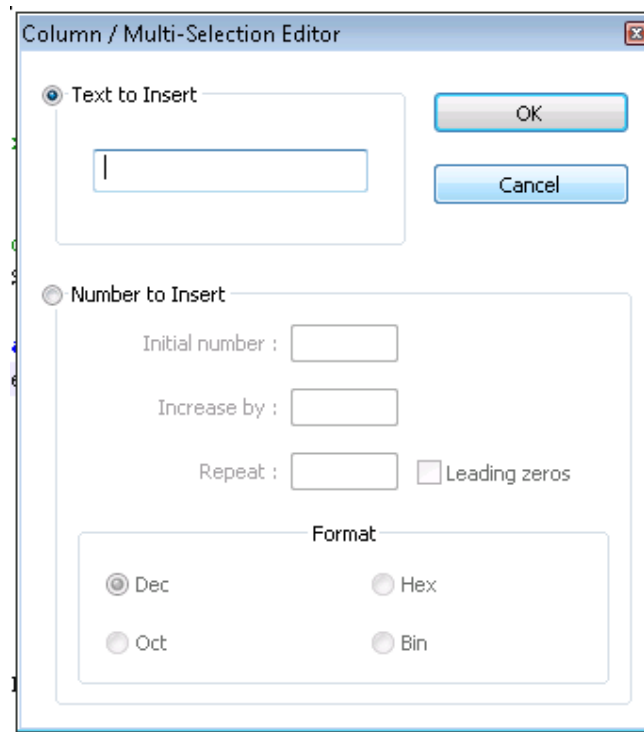


Figure 1.1: Column Editor

### 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:

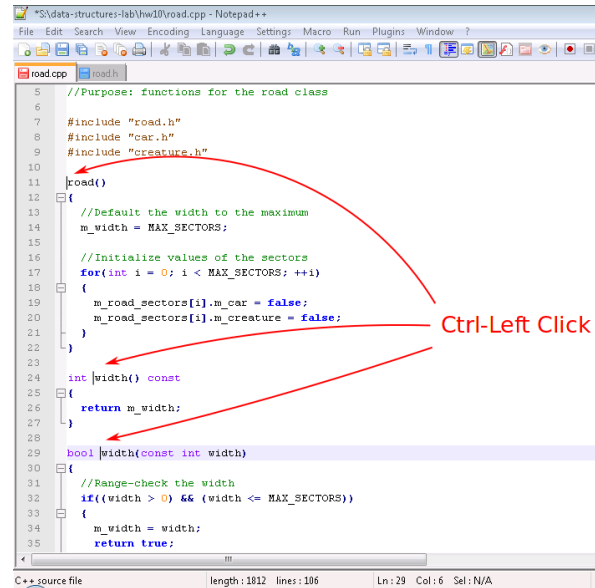


Figure 1.2: Placing multiple cursors with Ctrl + left-click

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

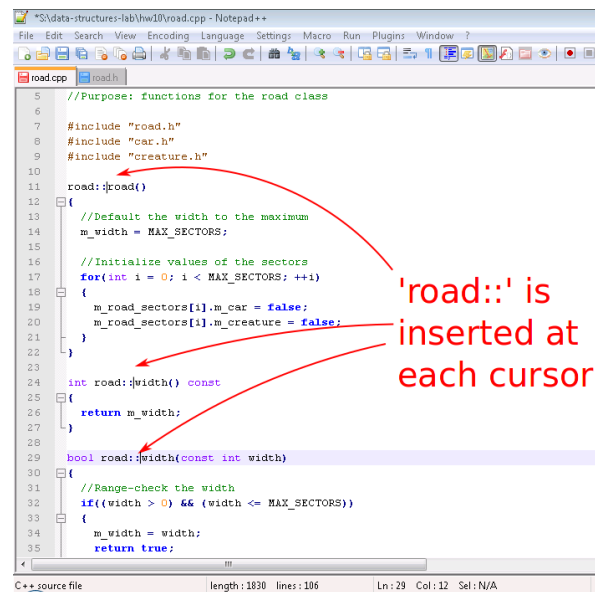


Figure 1.3: Typing `road::` inserts that text at each cursor location

## Document Map

A document map can be handy when navigating large files<sup>6</sup>. 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**

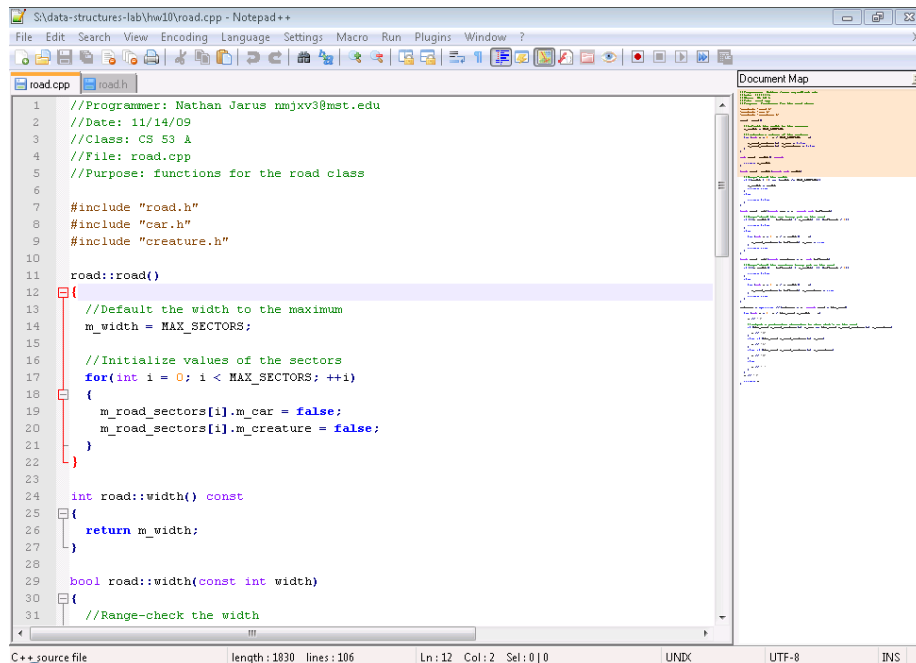


Figure 1.4: The document map

## 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](#)<sup>7</sup>. Unfortunately, plugins must be installed to the same directory Notepad++ is installed

<sup>6</sup>Of course, this feature might encourage making large files rather than multiple manageable files...

<sup>7</sup>[http://docs.notepad-plus-plus.org/index.php?title=Plugin\\_Central](http://docs.notepad-plus-plus.org/index.php?title=Plugin_Central)

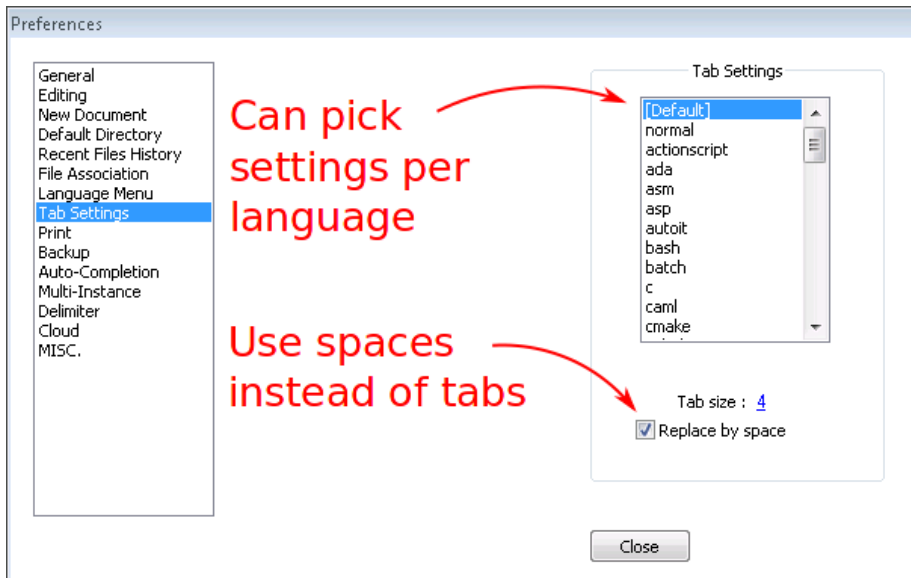


Figure 1.5: Configuring Notepad++ to use spaces rather than tabs

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<sup>8</sup> and behavior can be changed using JavaScript<sup>9</sup>.

Additionally, being a hip-and-trendy <sup>TM</sup>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<sup>10</sup>!

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

---

<sup>8</sup>CSS is used to specify the design for websites, and it works in Atom, too.

<sup>9</sup>JavaScript[<sup>^</sup>java] is the language of the web. It makes web pages interactive!

<sup>10</sup>Just don’t expect to get rich.

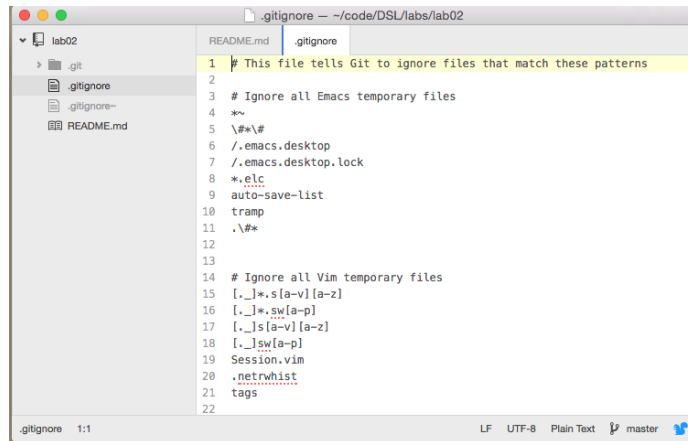


Figure 1.7: Atom with multiple tabs in one pane

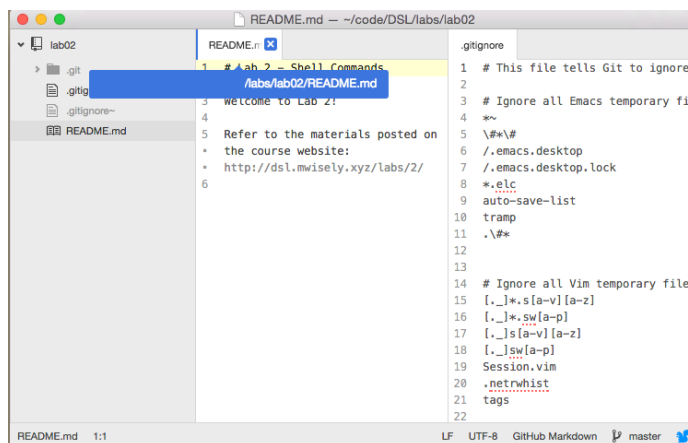


Figure 1.8: One window split into two panes

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<sup>11</sup>!

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

<sup>11</sup>Please, please don't fret. It'll be OK. Just keep a-readin', friend.

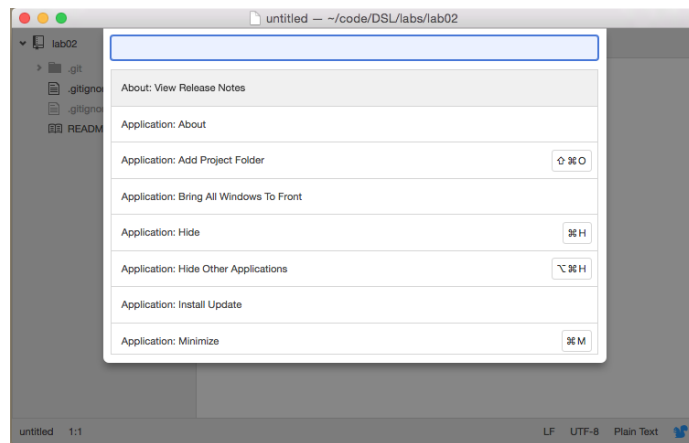


Figure 1.9: The Atom command palette (i.e., your best friend forever).

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 a 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!



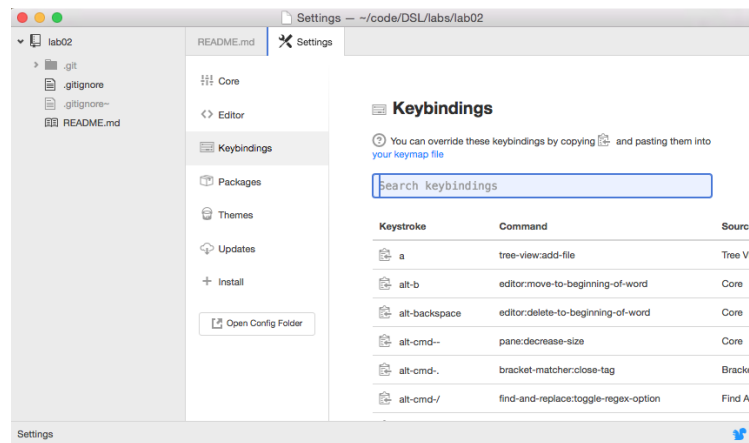


Figure 1.10: Atom’s settings open in a new tab. You can search through its keybindings here.

## 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`<sup>12</sup> configured to use commands similar to `pico`<sup>13</sup>. `pico` is a small (eh? eh?) text editor that came with the PINE newsreader<sup>14</sup> and was designed to be easy to use<sup>15</sup>.)

## 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, `^` is shorthand for the `Ctrl` key. So, for instance, `^X` corresponds to `Ctrl+X`. For historical

<sup>12</sup><http://joe-editor.sourceforge.net>

<sup>13</sup><http://www.guckes.net/pico/>

<sup>14</sup>A newsreader is a program for reading Usenet posts. Imagine Reddit, but in the 1980s.

<sup>15</sup>Well, easy to use for people (sometimes known as ‘humanity’) who were already used to using Unix terminal programs!

reasons<sup>16</sup>, 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**,**=**.

---

<sup>16</sup>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.<sup>17</sup> 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.

---

<sup>17</sup>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:

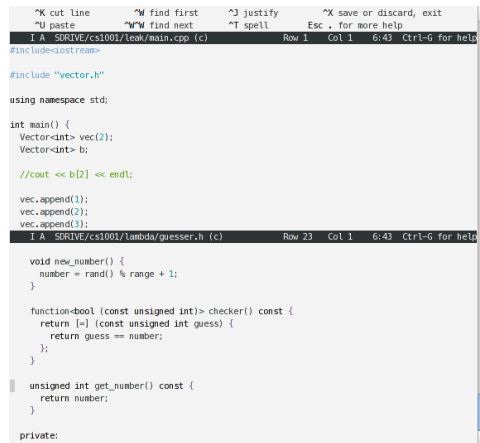
- Press `Enter` to search forwards
- Press `i` to search forward and ignore the case (so searching for “bob” will match “Bob” as well)
- Press `b` to search backwards
- Press `r` to replace matches with a new string. `jpico` will prompt whether or not to replace for each match
- Press `k` to select from 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`.



```

^K cut line      ^W find first    ^J justify      ^X save or discard, exit
^U paste         ^WW find next   ^T spell        Esc. for more help
I A SDRIVE/cs1001/leak/main.cpp (c) Row 1 Col 1 6:43 Ctrl-G for help
#include<iostream>

#include "vector.h"

using namespace std;

int main() {
    Vector<int> vec(2);
    Vector<int> b;

    //cout << b[2] << endl;

    vec.append(1);
    vec.append(2);
    vec.append(3);

I A SDRIVE/cs1001/lambda/guesser.h (c) Row 23 Col 1 6:43 Ctrl-G for help
void new_number() {
    number = rand() % range + 1;
}

function<bool> (const unsigned int)> checker() const {
    return [=] (const unsigned int guess) {
        return guess == number;
    };
}

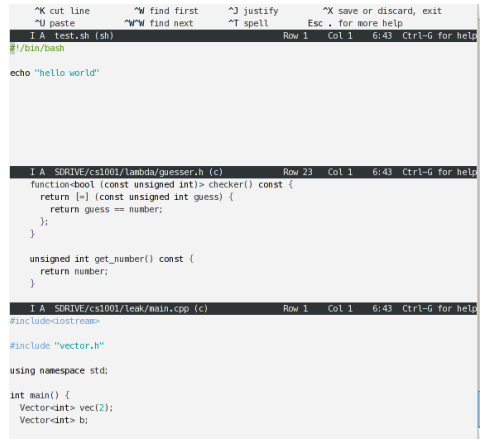
unsigned int get_number() const {
    return number;
}

private:

```

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`.



```

^K cut line      ^W find first    ^J justify      ^X save or discard, exit
^U paste         ^WW find next   ^T spell        Esc. for more help
I A test.sh (sh) Row 1 Col 1 6:43 Ctrl-G for help
#!/bin/bash

echo "hello world"

I A SDRIVE/cs1001/lambda/guesser.h (c) Row 23 Col 1 6:43 Ctrl-G for help
function<bool> (const unsigned int)> checker() const {
    return [=] (const unsigned int guess) {
        return guess == number;
    };
}

unsigned int get_number() const {
    return number;
}

I A SDRIVE/cs1001/leak/main.cpp (c) Row 1 Col 1 6:43 Ctrl-G for help
#include<iostream>

#include "vector.h"

using namespace std;

int main() {
    Vector<int> vec(2);
    Vector<int> b;

```

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](#)<sup>18</sup>.)

## 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.

---

<sup>18</sup><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<sup>19</sup>. 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<sup>20</sup>...

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 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 right to left using the following notation:

---

<sup>19</sup>For example, if you are using X forwarding, Emacs can detect the ability to open a GUI for you.

<sup>20</sup>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.



## 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”<sup>21</sup>. The kill ring is... a ring that stores things you’ve killed (cut), so that you can yank (paste) them later.

Vocabulary:

- **Kill**<sup>22</sup> - 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:

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.

---

<sup>21</sup>Don’t ask why Emacs has such violent terms. There’s no keyboard-related excuse for that one.

<sup>22</sup>Don’t ask why Emacs has such violent terms. There’s no keyboard-related excuse for that one.

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<sup>23</sup> 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.

## 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 (**initialization**). 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:

---

<sup>23</sup>Don't ask why Emacs has such violent terms. There's no keyboard-related excuse for that one.

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<sup>24</sup> 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 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`.

<sup>24</sup>The graphical version is cleverly named `gvim`.

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<sup>25</sup>
- `^`/`$`: Beginning/end of line
- `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.

---

<sup>25</sup>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.

### 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, `"a` 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
- JPic
- 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 1.13: 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`<sup>1</sup>. 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/nmjxv3
```

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.

---

<sup>1</sup>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<sup>2</sup>. 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`<sup>3</sup>.

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
```

<sup>2</sup>Thanks, old curmudgeons who can't be bothered to learn to type 'list'.

<sup>3</sup>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<sup>4</sup>. (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:

---

<sup>4</sup>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`<sup>5</sup>.

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`<sup>6</sup>. 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.

---

<sup>5</sup>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).

<sup>6</sup>`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

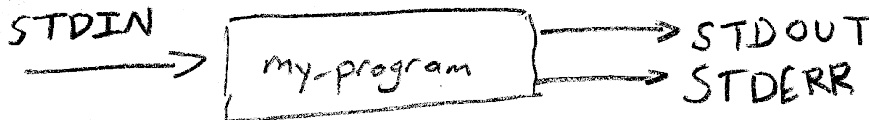
Many programs include help text; typically `--help` or `-h` display this text. It can be a good quick reference of common options.

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.

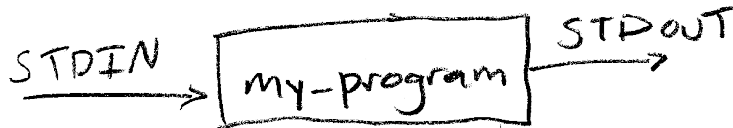
## I/O Redirection

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:

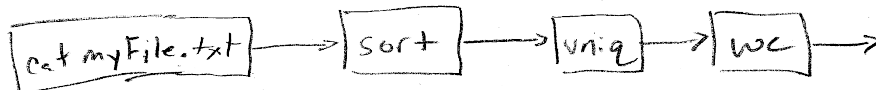


```
nmjxv3@rc02xcs213:~$ echo "I love to program" | wc
      1      4     18
```

Piping lets us compose all the utilities Linux comes with into more complex programs<sup>7</sup>. 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`.

<sup>7</sup>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

Blap blap blap



## Chapter 4

# Bash Scripting

Blap blap blap



## Chapter 5

# Regular Expressions

Blap blap blap



## Chapter 6

# Integrated Development Environments

Blap blap blap





## Chapter 7

# Building with Make

Blap blap blap



## Chapter 8

# Debugging with GDB

Blap blap blap



## Chapter 9

# Locating memory leaks with Memcheck

Blap blap blap



## Chapter 10

# Profiling

Blap blap blap





## Chapter 11

# Unit testing with Boost Unit Test Framework

Blap blap blap



## Chapter 12

# Using C++11 and the Standard Template Library

Blap blap blap



## Chapter 13

# Graphical User Interfaces with Qt

Blap blap blap



## Chapter 14

# Typesetting with LaTeX

Blap blap blap