

# THE CS 1001 ENCHIRIDION<sup>1</sup>

Or, How To Use A Computer Real Good

Nathan Jarus and Michael Wisely

<sup>1</sup>Thanks to those who put up with our nonsense and those who paid us to write a book that is full of it.



# Contents

1	Exploring Text Editors	7
2	Bash Basics	37
3	Git Basics	49
4	Bash Scripting	63
5	Regular Expressions	75
6	Integrated Development Environments	85
7	Building with Make	93
8	Debugging with GDB	105
9	Finding Memory Safety Bugs	115
10	Profiling	131
11	Unit testing with Boost Unit Test Framework	143
12	Using C++11 and the Standard Template Library	145
13	Graphical User Interfaces with Qt	147
14	Typesetting with LaTeX	149
A	General PuTTY usage	151
B	X-forwarding	157
C	Markdown	161
D	Parsing command-line arguments in C++	163



# 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.<sup>3</sup> 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.

<sup>3</sup>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<sup>4</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>5</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>4</sup>Thanks to cloud computing, this is becoming commonplace, yo.

<sup>5</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>6</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>6</sup> *Outdent* (verb). Latin: To remove a tooth; English: The opposite of indent.

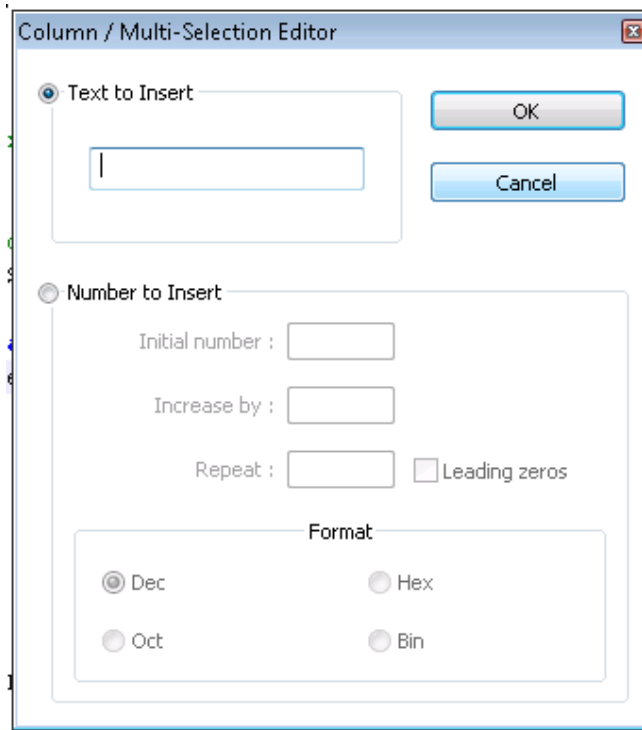


Figure 1.1: Column Editor

### 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:

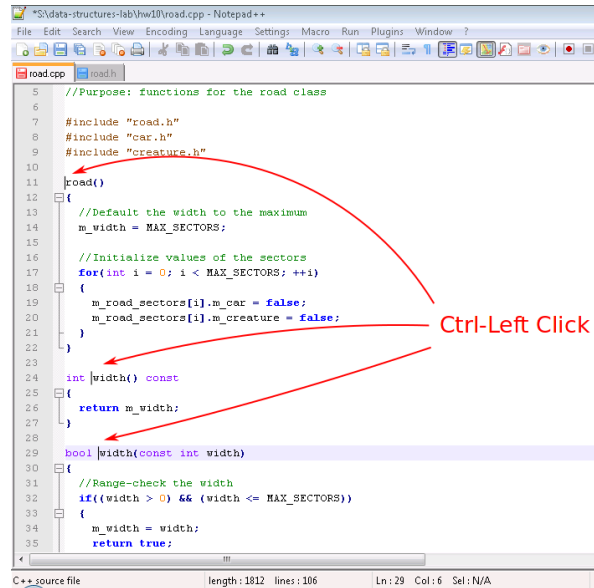


Figure 1.2: Placing multiple cursors with Ctrl + left-click

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

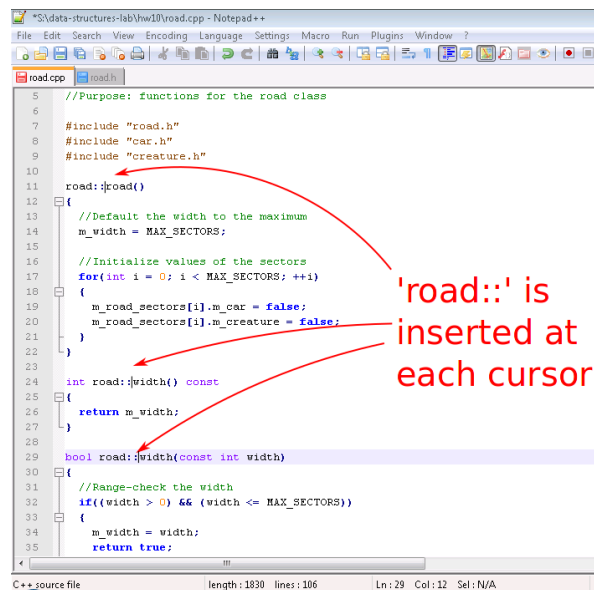


Figure 1.3: Typing `road::` inserts that text at each cursor location

## Document Map

A document map can be handy when navigating large files<sup>7</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**

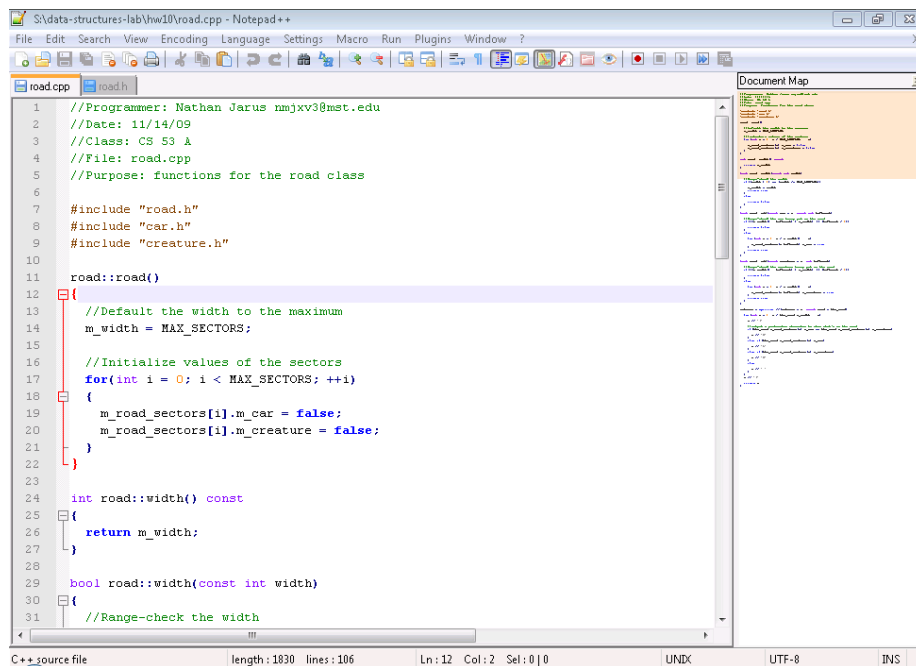


Figure 1.4: The document map

## 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:

<sup>7</sup>Of course, this feature might encourage making large files rather than multiple manageable files...

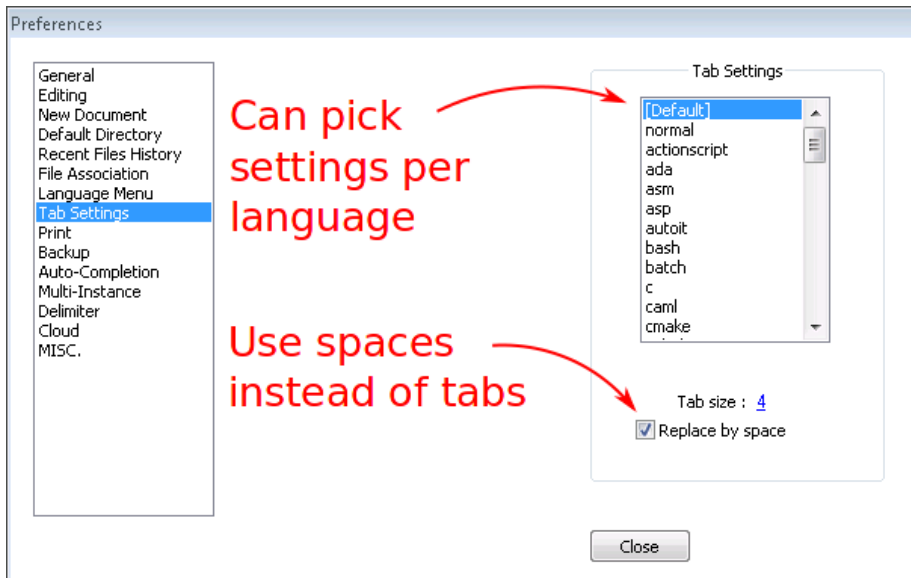


Figure 1.5: Configuring Notepad++ to use spaces rather than tabs

## Plugins

Notepad++ has support for plugins; you can see a list of them [here](#)<sup>8</sup>. Unfortunately, plugins must be installed to the same directory Notepad++ is installed 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>9</sup> and behavior can be changed using JavaScript<sup>10</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

<sup>8</sup>[http://docs.notepad-plus-plus.org/index.php?title=Plugin\\_Central](http://docs.notepad-plus-plus.org/index.php?title=Plugin_Central)

<sup>9</sup>CSS is used to specify the design for websites, and it works in Atom, too.

<sup>10</sup>JavaScript[<sup>^</sup>java] is the language of the web. It makes web pages interactive!

missing some particular behavior, you can create a package and make it available to the world, as well<sup>11</sup>!

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

## Tree View



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

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?”.

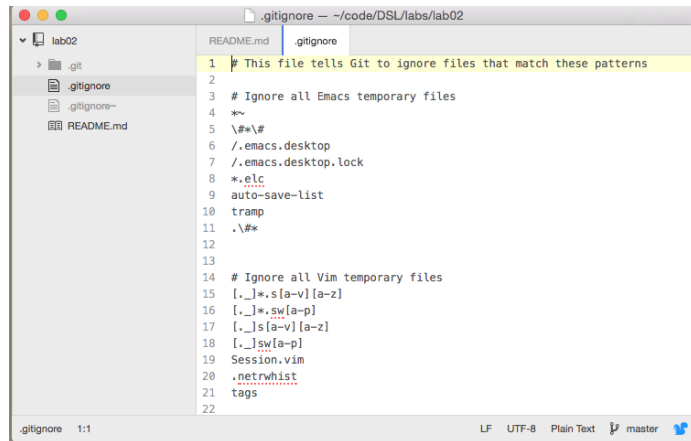


Figure 1.7: Atom with multiple tabs in one pane

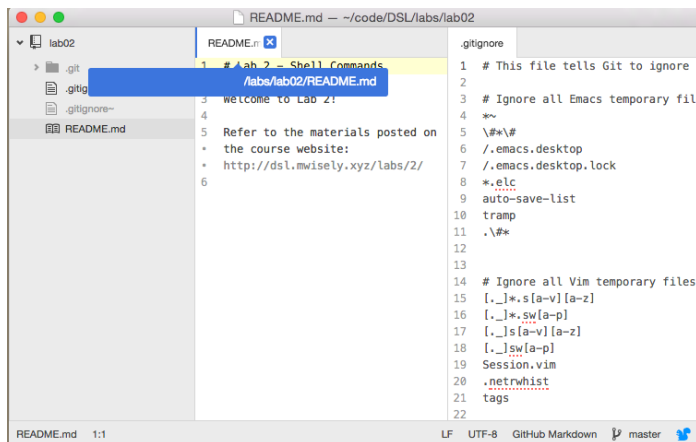


Figure 1.8: One window split into two panes

## 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 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`.

---

<sup>11</sup>Just don't expect to get rich.

## The Command Palette

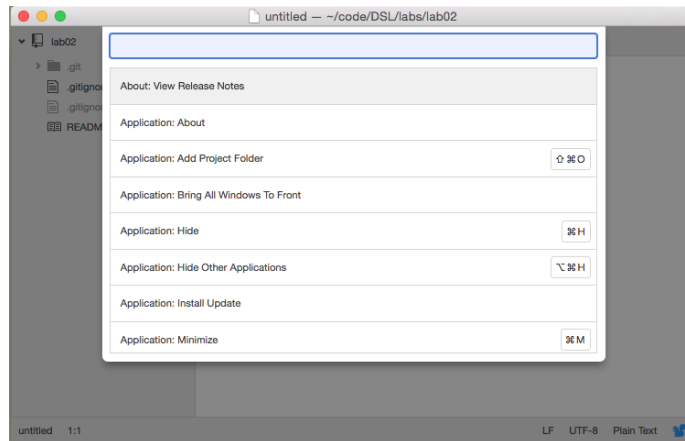


Figure 1.9: The Atom command palette (i.e., your best friend forever).

You may notice that Atom’s drop-down menu options are sparse. There is not much to choose from. Don’t fret<sup>12</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 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.

<sup>12</sup>Please, please don’t fret. It’ll be OK. Just keep a-readin’, friend.



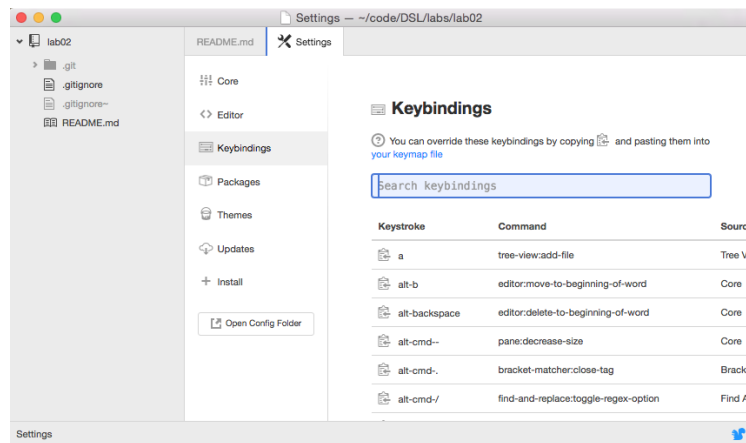


Figure 1.10: Atom's settings open in a new tab. You can search through its keybindings here.

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!

## 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>13</sup> configured to use commands similar to **pico**<sup>14</sup>. **pico** is a small (eh? eh?) text editor that came with the PINE newsreader<sup>15</sup> and was designed to be easy to use<sup>16</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

<sup>13</sup><http://joe-editor.sourceforge.net>

<sup>14</sup><http://www.guckes.net/pico/>

<sup>15</sup>A newsreader is a program for reading Usenet posts. Imagine Reddit, but in the 1980s.

<sup>16</sup>Well, easy to use for people (sometimes known as 'humanity') who were already used to using Unix terminal programs!

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 reasons<sup>17</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>17</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>18</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>18</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:

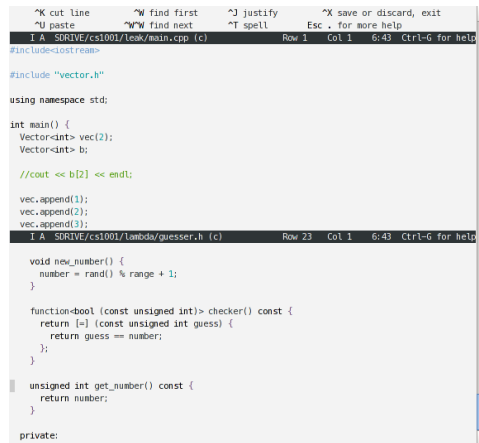
- Press `Enter` to search forwards
- Press `i` to search forward and ignore the case (so searching for “bob” will match “Bob” as well)
- Press `b` to search backwards
- Press `r` to replace matches with a new string. `jpico` will prompt whether or not to replace for each match
- Press `k` to select from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

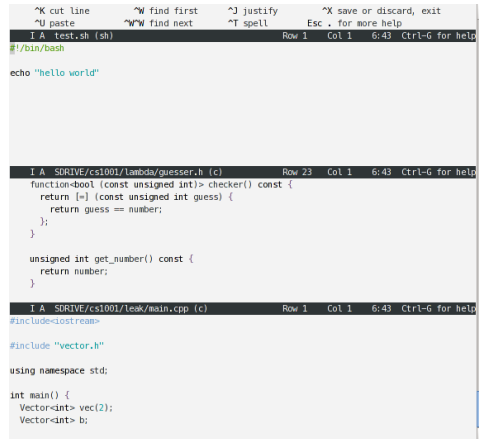
unsigned int get_number() const {
    return number;
}

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 SORIVE/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 SORIVE/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>19</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.

## 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.

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

```
$ 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>20</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>21</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 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:

- 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

<sup>20</sup>For example, if you are using X forwarding, Emacs can detect the ability to open a GUI for you.

<sup>21</sup>Well, the `nw` in `-nw` stands for no window, but Emacs takes it much more dramatically.

- **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”<sup>22</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>23</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.

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:

---

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

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

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>24</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:

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**.

---

<sup>24</sup>Don't ask why Emacs has such violent terms. There's no keyboard-related excuse for that one.

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>25</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 (`f`), 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

---

<sup>25</sup>The graphical version is cleverly named `gvim`.

- **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>26</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.

## 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,

---

<sup>26</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.

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` 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.)



## QUESTIONS

33

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.

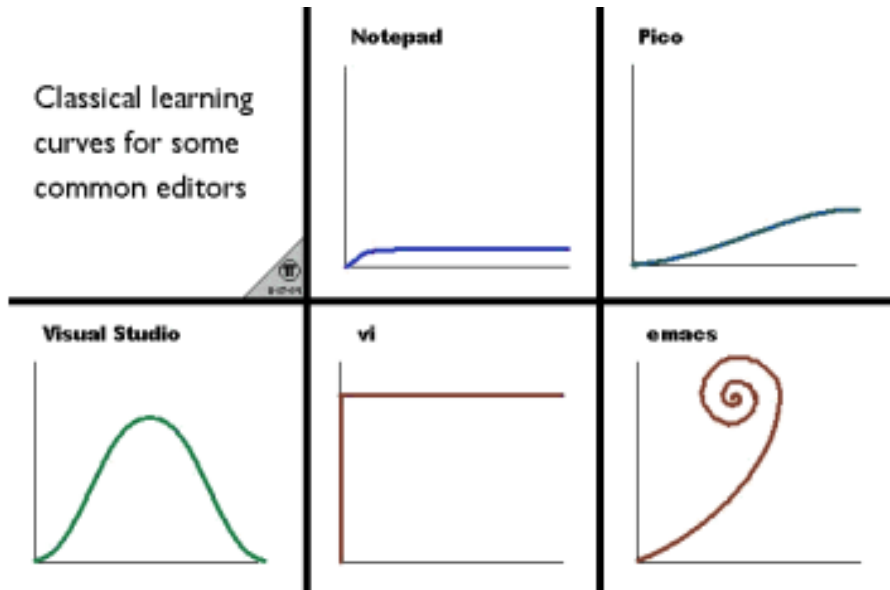


Figure 1.14: Learning editors is easy and fun!

## 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`<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
-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
```

<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.



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:

- `*` 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.

---

<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.

## 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.

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`.

---

<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`.

## The Manual

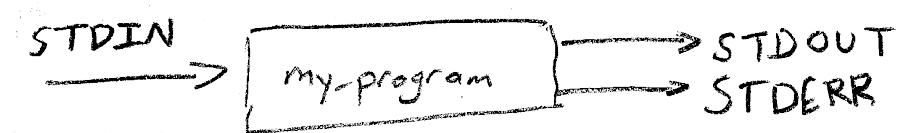
Many programs include help text; typically `--help` or `-h` display this text. It can be a good quick reference of common options.

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
```

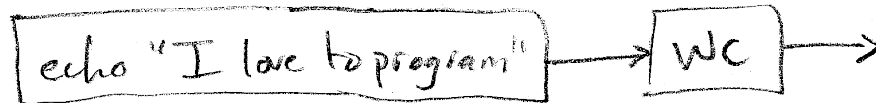
But, we can redirect this output to files or to other programs!

- `|` 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:

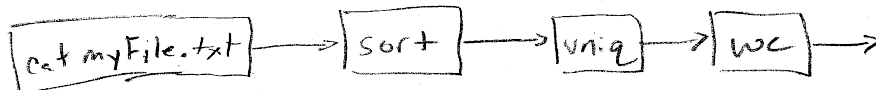


```
nmjxv3@rc02xcs213:~$ echo "I love to program" | wc
      1      4     18
```

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`.

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`.

<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!

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<sup>1</sup>.

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<sup>2</sup>.

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<sup>3</sup>, 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.

---

<sup>1</sup>Now open them again, because it’s hard to read with your eyes shut.

<sup>2</sup>This is your daydream, friend. I have no idea what this program is or does.

<sup>3</sup>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<sup>4</sup> that Git will feel similar to these services in some ways, but there are many features that make them *very* different.

---

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

## Trying out GitLab

To backup<sup>6</sup> 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<sup>7</sup>, 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.

1. Look for the "Create a new repository" section and copy the command that starts with `git clone https://...my-fancy-project.git`

---

<sup>5</sup>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.

<sup>6</sup>And other things. Remotes are actually *extremely* useful.

<sup>7</sup>GitLab, GitHub, BitBucket, etc.

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<sup>8</sup>, 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.

## 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.

---

<sup>8</sup>Such as "asdf", "stuff", "work", or "finished the lab".

- **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<sup>9</sup>.

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

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

You grimace.

---

<sup>9</sup>Maybe just half-closed this time.

<sup>10</sup>It's as though they were laying there in the grass the whole time. So weird.





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<sup>11</sup>.

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.

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.

---

<sup>11</sup>You later realize that the muse was just Tony Robbins getting you super amped about everything.

One of Git's most powerful features is its ability to **branch** your code's timeline. No, it's not like Primer<sup>12</sup>. 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<sup>13</sup>, smoking a pipe<sup>14</sup>, sipping on bourbon<sup>15</sup>, and wondering what commands you use to actually work with branches in Git. It's time to get your hands dirty.

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.

---

<sup>12</sup>Although you should *absolutely* see it if you haven't. Who doesn't love a good indie time travel movie?

<sup>13</sup>I assume.

<sup>14</sup>I reckon.

<sup>15</sup>That's a given.

## 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<sup>1</sup> 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
```

---

<sup>1</sup>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`<sup>2</sup> at your shell prompt.

Two things to note: 1. The first line, called a “shebang”<sup>3</sup>, 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<sup>4</sup> of the last command run. Traditionally,

---

<sup>2</sup>. 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.

<sup>3</sup>A combination of “sharp” (`#`) and “bang” (`!`).

<sup>4</sup>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<sup>5</sup>. 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:

---

<sup>5</sup>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<sup>6</sup> 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 expressions: `[ ]` is shorthand for the `test` command, and `[[ ]]` is a Bash builtin. `[ ]` works on shells other than Bash, but `[[ ]]` is far less confusing<sup>7</sup>.

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.

### 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

<sup>6</sup>Or a builtin shell command (see `man bash` for details).

<sup>7</sup>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.

type of comparison to use, rather than hoping that Bash guesses right<sup>8</sup>.

Comparing Strings:

- `=,==`: Either
  - String equality, if both operands are strings, or
  - Pattern (glob) matching, if the RHS is a glob.
- `!=`: Either
  - String inequality, 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:

- `&&`: 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
```

---

<sup>8</sup>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`.

```
    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 between 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.<sup>9</sup>
3. Write a script that prints “directory” if the first argument is a directory and “file” if the first argument is a file.

---

<sup>9</sup>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



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

## 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.<sup>1</sup> 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.<sup>2</sup> 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.<sup>3</sup> 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
- 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/>.

---

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

<sup>2</sup>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.)

<sup>3</sup>The umami flavors are my favorite.

## 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 `as`, followed by one or more `bs`.
- `\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 `()`! `\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 "cccaabbbddddd", 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.<sup>4</sup> 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 "!"

---

<sup>4</sup>POLITICS!

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, `!#0$!%^IA(jfio2q4Tj1$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:

- `-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.<sup>5</sup> 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.

---

<sup>5</sup>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.



```
$ echo "bananas" | grep 'b\(an\)\+as'
bananas
$ echo "bananananananas" | grep 'b\(an\)\+as'
bananananananas
$ 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 regular 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.

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<sup>6</sup>, 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 Pizzeria, 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”.

---

<sup>6</sup>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

Despite their modern appearance and ergonomic appeal, some people don't much care for console applications. Truly, it is a mind-bending reality that some people would prefer to use a mouse to click *buttons* rather than typing commands in a shell. What a world!



Figure 6.1: Ain't that just the way?

In our first lab, we tried a number of different text editors. A couple of those (including Atom and Notepad++) included **graphical user interfaces** (GUIs). This means you're free to use your mouse to select text, move text, scroll through text, choose menu action, click buttons, et cetera. You don't have to do *everything* with keyboard shortcuts.

GUI text editors usually come with a lot of “batteries included”. However, there are still many more batteries available.

Consider how your programming process would change if you could:

- See syntax errors in your text editor.
- Compile and run your program with a single button click.
- Step through the execution of your program, line by line, to hunt down bugs.

These features are often present in **integrated development environments** (IDEs). An IDE is GUI<sup>1</sup> essentially a text-editor with a compiler and other development tools built in. Everything you need is in one place. There’s no need to open a terminal to run `g++` or anything like that unless you really want to.

We know, we know. You’re *really* going to miss `g++`.

Just bear with us for this lab. Then you can travel back to 1980 when things were actually good<sup>2</sup>.

## Takeaways

- Gain experience with Integrated Development Environments (IDEs)
  - Explore editor features
  - Try build (compilation) features
  - Experiment with built-in debugging tools

## Walkthrough

We’re going to try out a couple of open source<sup>3</sup> IDEs that run on Linux. Usually when we connect to a remote Linux machine, all we need is a shell, and PuTTY does everything we need. For this lab, we’re going to need to setup X-forwarding as well. If you haven’t already, be sure to reference the appendix on [X-forwarding](#).

Before lab, make sure you’re able to start the two IDEs we will be using:

- Geany (`geany`) should look like Figure [6.2](#)
- Code::Blocks (`codeblocks`) should look like Figure [6.3](#)

---

<sup>1</sup>Depending who you ask, IDEs are not necessarily GUI programs. If you install enough plugins in vim or Emacs, some people would call those IDEs, as well.

<sup>2</sup>Something something Ronald Reagan something something Breakfast Club something something.

<sup>3</sup>Open source software (OSS) that is free as in freedom and free as in beer (or waffles if you prefer).

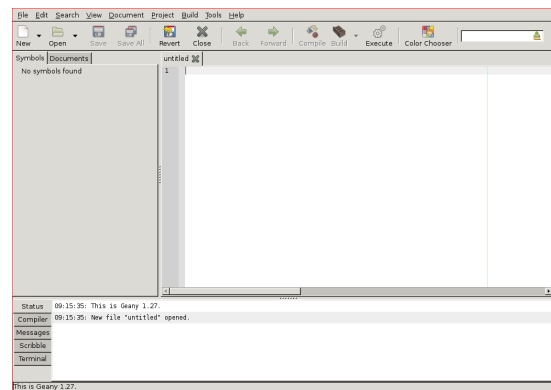


Figure 6.2: Geany

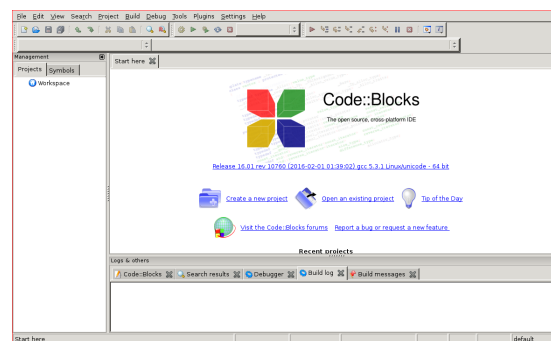


Figure 6.3: Code::Blocks

During your lab session, you'll be exploring some of the features of each of these IDEs.

## Questions

1. Click the “Build” drop-down menu in Geany. What are the sub-menu options?

Note that we want to include the grayed-out options, too. Hint: there are seven of them.

2. Click the “Help” drop-down menu in Code::Blocks. What are the sub-menu options?

Don’t worry about sub-menus. Hint: there are three of them.



## Quick Reference

### General Tips

- Make sure you start Xming before you try to forward any X11 windows!
- You won't be able to use bash within the shell that's running **geany** or **codeblocks**. You may find it useful to keep a couple of PuTTY windows open while you work.

### Geany

#### Troubleshooting

- If geany complains that it's unable to open its shell...
  1. Go to **Edit** » **Preferences** » **Tools**
  2. Change Terminal from `x-terminal-emulator -e "/bin/sh %c"` to `xterm -e "/bin/sh %c"`
  3. Save your changes

#### Geany Features

- **F9** Builds the project
- **F5** Runs the project

#### Building with Geany

It doesn't work so well for multi-file projects.

After you create a project...

- Go to **Project** » **Properties** » **Build**
- Find the "Build" command (`g++ -Wall -o "%e" "%f"`)
- Change it to `g++ -Wall -o "%e" *.cpp`

#### Writing code with Geany

- **Ctrl** + **t** – Go to function implementation
- **Ctrl** + **space** – Show completions.
- Set space preference to "Spaces" in **Edit** » **Preferences** » **Editor** » **Indentation**
- You can auto-close brackets and parentheses as well.

## Code::Blocks

### Troubleshooting

- Don't start Code::Blocks if your `pwd` is in your SDRIVE. Try changing to your linuxhome directory (i.e., `cd ~`). Otherwise you'll be waiting *all day* for Code::Blocks to open up.
- If Code::Blocks doesn't let you navigate to your cloned repository, you can find it by going the long way. You can find your SDRIVE by going to its absolute path: `/nethome/users/<username>/cs1001/lab06/` (or whatever you call your repository.)

### Building with Code::Blocks

- **F9** – Build and run
- **Ctrl** + **F9** – Build
- **Ctrl** + **F10** – Run
- Enable `-Wall` in **Project** > **Build Options**

### Writing code with Code::Blocks

- **Ctrl** + **.** – Go to function implementation.
- **Ctrl** + **↑** + **.** – Go to function declaration.
- **Ctrl** + **Space** – Show completions.
- Right-click on a file and choose 'Format this file' to autoformat.

## Further Reading

### Geany

- Geany Project Homepage: <http://www.geany.org/>
- Geany Plugins: <http://plugins.geany.org/downloads.html>

### Code::Blocks

- Code::Blocks Project Homepage: <http://www.codeblocks.org>
- Tutorial from cplusplus.com: <http://www.cplusplus.com/doc/tutorial/introduction/codeblocks/>

## Other IDEs

There are a bunch of other IDEs out there. Some are free; some are not. If you like the IDEa of an IDE, but you don't like Geany or Code::Blocks, you may give one of these guys a try.

- Visual Studio Code: <https://code.visualstudio.com/>
- CLion: <https://www.jetbrains.com/clion/>



## Chapter 7

# Building with Make

### Motivation

Wow. You’ve made it six chapters through this book. And probably some appendices too. And yet you have made not one sandwich. Not one!

Let’s fix that. Time for a classic pastrami on rye. You go to fetch ingredients from the refrigerator, but alas! It is empty. Someone else has been eating all your sandwiches while you were engrossed in regular expressions.

You hop on your velocipede<sup>1</sup> and pedal down to the local bodega only to discover that they, too, are out of sandwich fixin’s. Just as you feared – you are left with no choice other than to derive a sandwich from first principles.

A day of cycling rewards you with the necessities: brisket, salt, vinegar, cucumbers, yeast, rye, wheat, caraway seeds, sugar, mustard seed, garlic cloves, red bell peppers, dill, and peppercorns. Fortunately you didn’t have to tow a cow home! You set to work, pickling the beef and the cucumbers and setting the bell peppers out to dry. Once the meat has cured, you crush the peppers, garlic, mustard seed, and peppercorns into a delicious dry rub and fire up the smoker. Eight hours later and your hunk of pastrami is ready to be steamed until it’s tender.

Meanwhile, you make a dough of the rye and wheat flours, caraway seeds, yeast, and a little sugar for the yeast to eat. It rises by the smoker until it’s ready to bake. You bake it with a shallow pan of water underneath so it forms a crisp outer crust.

Finally, you crush some mustard seed and mix in vinegar. At last, your sandwich is ready. You spread your mustard on a slice of bread, heap on the pastrami, and garnish it with a fresh pickle. Bon appétit!

---

<sup>1</sup>Velocipede (n): A bicycle for authors with access to a thesaurus.

In between bites of your sandwich, you wonder: “Wow, that was a lot of work for a sandwich. And whenever I eat my way through what I’ve prepared, I’ll have to do it all over again. Isn’t there a Better Way?”

A bite of crunchy pickle is accompanied by a revelation. If the human brain is a computer, then this sandwich is code<sup>2</sup>: without it, your brain could compute nothing!<sup>3</sup> “Wow!” you exclaim through a mouthful of pickle, “This is yet another problem solved by GNU Make!”

Using the powers of `git`, you travel into the future, read the rest of this chapter, then head off your past self before they pedal headfirst down the road of wasted time.<sup>4</sup> Instead of this hippie artisanal handcrafted sandwich garbage, you sit your past self down at their terminal and whisper savory nothings<sup>5</sup> in their ear. They – you – crack open a fresh editor and pen the pastrami of your dreams:

```
pickles: cucumbers vinegar dill
        brine --with=dill cucumbers

cured-brisket: brisket vinegar
              brine brisket

paprika: red-peppers
         sun-dry red-peppers | grind > paprika

rub: paprika garlic mustard-seed peppercorns
     grind paprika garlic mustard-seed peppercorns > rub

smoked-brisket: cured-brisket rub
               season --with=rub cured-brisket -o seasoned-brisket
               smoke --time=8 hours seasoned-brisket

pastrami: smoked-brisket
         steam --until=tender smoked-brisket

dough: rye wheat coriander yeast sugar water
       mix --dry-first --yeast-separately rye wheat coriander yeast \
         sugar water --output=dough

rye-loaf: dough
         rise --location=beside-smoker dough && bake -t 20m
```

---

<sup>2</sup>And your code is a sandwich: oodles of savory instructions sandwiched between the ELF header and your static variables.

<sup>3</sup>The only thing more tortured than this analogy is the psychology 101 professors reading another term paper on how humans are just meat computers.

<sup>4</sup>Boy howdy do I wish this happened to me more often.

<sup>5</sup>Chaste tales of future sandwiches.

```
mustard: mustard-seed vinegar
        grind mustard-seed | mix vinegar > mustard

sandwich: pastrami pickles rye-loaf
          slice rye-loaf pastrami
          stack bread-slice --spread=mustard pastrami bread-slice
          present sandwich --garnish=pickle
```

Et voilà! You type `make sandwich`. Your computer’s fans spin up. Text flies past on the screen. A bird flies past the window. Distracted momentarily, you look away to contemplate the beauty of nature. When you turn back, there on your keyboard is a delicious sandwich, accompanied by a pickle. You quickly get a paper towel to mop up the pickle brine before it drips into your computer. Should have used `plate`!

Since you’ve already read this chapter in the future, I should not need to mention the myriad non-culinary uses of `make`. However, for the benefit of those who skipped the time-travel portion of the `git` chapter, I will anyway. `make` is a program for making files from other files. Perhaps its most common application is compiling large programming projects: rather than compiling every file each time you change something, `make` can compile each file separately, and only recompile the files that have changed. Overall, your compile times are shorter, and typing `make` is much easier than typing `g++ *.cpp -o neat-program`. It has other uses, too: this book is built with `make`!

## Takeaways

- Learn to make a decent pastrami on rye
- Learn how to compile and link your C++ code
- Understand `make`’s syntax for describing how files are built
- Use variables and patterns to shorten complex makefiles

## Walkthrough

### A bit about compiling and linking

Before we can set up a makefile for a C++ project, we need to talk about compiling and linking code. “Compiling” refers to the process of turning C++ code into machine instructions. Each `.cpp` file gets compiled separately, so if you use something defined in another file or library – for example, `cin` – the compiler leaves itself a note saying “later on when you figure out where `cin` is, put its address here”. Once your code is compiled, the linker then goes through all your compiled code and any libraries you have used and fills in all the notes with the appropriate addresses.

You can separate these steps: `g++ -c file.cpp` just does the compilation step to `file.cpp` and produces a file named `file.o`. This is a so-called *object file*; it consists of assembly code and cookies left out for the linker.<sup>6</sup>

To link a bunch of object files together, you call `g++` again<sup>7</sup> like so: `g++ file1.o file2.o -o myprogram`. `g++` notices that you have given it a bunch of object files, so instead of going through the compilation process, it prepares a detailed list<sup>8</sup> explaining to the linker which files and libraries you used and how to combine them together. It sets the list next to your object files<sup>9</sup> and waits for the linker. When the linker arrives, it paws through your object files, eats all the cookies, and then through a terrifying process not entirely understood by humans<sup>10</sup>, leaves you a beautiful executable wrapped up under your tree<sup>11</sup>.

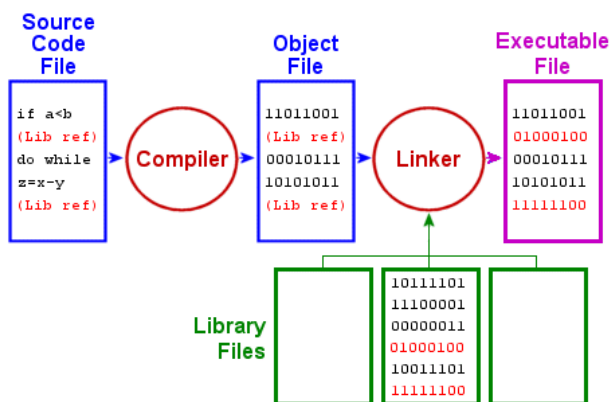


Figure 7.1: How Executables are Made

So, what's the big deal? Well, if you compile your code to object files and then change some of your code, you only need to rebuild the object files associated with the code you changed! All the other object files will stay the same. If you have a big project with a lot of files, this can make recompiling your code significantly faster! Of course, doing all this by hand would be awful... which is why we have `make`!

<sup>6</sup>Good programmers always set a glass of milk out for the linker when compiling large programs.

<sup>7</sup>I agree that this is somewhat confusing, but trust me it is much much much less confusing than figuring out how to call `ld`, the actual linker tool, on your own.

<sup>8</sup>And checks it twice.

<sup>9</sup>And the glass of milk, if you set one out for this purpose.

<sup>10</sup>I am not exaggerating here; linking executables is surprisingly full of arcane, system-dependent edge cases. Fortunately some other poor soul (i.e., your compiler maintainer) has figured this out already and you should never need to worry about it.

<sup>11</sup>This Christmas chapter brought to you by H. P. Lovecraft.



## Making Files

When you run `make`, it looks for a file named `Makefile` or `makefile` in the current directory for a recipe for building your code. The contents of your `makefile` determine what gets made and how.

Most of what goes in a `makefile` are *targets*: the names of files you want to create. Along with each target goes one or more commands that, when run, create the target file.

For example, let's say you want to build an executable named `program` by compiling all the C++ files in the current directory. You could do the following:

```
program:
    g++ *.cpp -o program
```

Here, `program` is the target name. In a `makefile`, every target name is followed by a colon. On the next line, indented one tab, is the command that, when run, produces a file named `program`.

**NOTE:** Unlike most programs, `make` *requires* that you use tabs, not spaces, to indent.<sup>12</sup> If you use spaces, you'll get a very strange error message. So make sure to set your editor to put actual tabs in your `makefiles`.

Once you've put this rule in your `makefile`, you can tell `make` to build `program` by running `make program`.<sup>13</sup> You can also just run `make`; if you don't specify a target name, `make` will build the first target in your `makefile`.

Now, if you edit your code and run `make program` again, you'll notice a problem:

```
$ make program
make: 'program' is up to date.
```

Well, that's no good. `make` determines if it needs to re-build a target by looking at when the associated file was last modified and comparing that with the modification times of the files the target depends on. In this case, our `program` has no dependencies, so `make` doesn't think anything needs to happen!

Let's fix that. Since `make` is not omniscient<sup>14</sup>, we need to explicitly state what each file depends on. For our example, let's suppose we have a classic CS 1570

---

<sup>12</sup>Stuart Feldman, the author of `make`, explains:

Why the tab in column 1? Yacc was new, Lex was brand new. I hadn't tried either, so I figured this would be a good excuse to learn. After getting myself snarled up with my first stab at Lex, I just did something simple with the pattern newline-tab. It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history.

(From "Chapter 15. Tools: make: Automating Your Recipes", *The Art of Unix Programming*, Eric S. Raymond, 2003)

<sup>13</sup>Don't try to execute the `makefile` itself. `bash` is confused enough without trying to interpret `make`'s syntax!

<sup>14</sup>Not yet, at least.

assignment with a `main.cpp` file, a `funcs.cpp` file, and an associated `funcs.h` header. Whenever any one of these files change, we want to recompile `program`. We specify these dependencies after the colon following the target name:

```
program: main.cpp funcs.h funcs.cpp
    g++ *.cpp -o program
```

Now when we change those files and run `make`, it will re-build `program`!

This is all well and good, you say, but what about all those promises of sweet, sweet incremental compilation the previous section suggested? Not to worry! You can tell `make` about each one of your targets and dependencies and it will do the work of running each compilation and linking step as needed. Continuing our example, we add two new targets for our object files, `main.o` and `funcs.o`:

```
program: main.o funcs.o
    g++ main.o funcs.o -o program

main.o: main.cpp funcs.h
    g++ -c main.cpp

funcs.o: funcs.cpp funcs.h
    g++ -c funcs.cpp
```

Some things to notice in this example:

1. Our `program` target now depends not on our source files, but on `main.o` and `funcs.o`, which are themselves targets. This is fine with `make`; when building `program` it will first look to see if `main.o` or `funcs.o` need to be built and build them if so, then build `program`.
2. Each of our object file targets depends on `funcs.h`. This is because both `main.cpp` and `funcs.cpp` include `funcs.h`, so if the header changes, both object files may need to be rebuilt.

File targets are the meat and potatoes<sup>15</sup> of a healthy `make` breakfast. Most of your makefiles will consist of describing the different files you want to build, which files those files are built from, and what commands need to be run to build those files. Later on in this chapter we'll discuss how to automate common patterns, like for the object files in the above example. But now you know everything you need to get started using `make` to *make life-easier*!<sup>16</sup>

## Phony Targets

Building files is great and all, but sometimes there are commands that you want to run often that you'd really rather not type out each time. You *could* write a shell script, but you've already got a makefile; why not use that? The most

<sup>15</sup>Or tofu and potatoes, if that's your thing.

<sup>16</sup>At this point, have your past self pat you on the back. Good work!

common example of this is commands to clean up all the files **make** generates in your current directory. In this case, you don't want to generate any new files, and you don't want to lie to **make** because you're an honest upstanding citizen.

Fortunately, **make** supports this through something called *phony targets*. You can tell **make**, "Hey, this target isn't actually a file; just run the commands listed here whenever I ask you to build this target," and **make** will be like, "Sure thing, boss! Look at me, not being confused at all about why there's no file named 'clean'!"

Let's make a **clean** target for our example from the last section. Having a target named **clean** that gets rid of all the compiled files in your current directory is good **make** etiquette.

```
program: main.o funcs.o
    g++ main.o funcs.o -o program

main.o: main.cpp funcs.h
    g++ -c main.cpp

funcs.o: funcs.cpp funcs.h
    g++ -c funcs.cpp

.PHONY: clean

clean:
    -@rm -f program
    -@rm -f *.o
```

Now when you run **make clean**, it will delete any object files, as well as your compiled program.<sup>17</sup> There are a few pieces to this target:

1. There is a special target named **.PHONY**. Every target **.PHONY** depends on is a phony target that gets run every time you ask **make** to build it.
2. The **-** in front of a command tells **make** to ignore errors<sup>18</sup> from that command. Normally when a program exits with an error, **make** bails out under the assumption that if that command failed, anything that depended on it probably won't work either. Rather than trying anyway, it stops and lets you know what command failed so you can fix it. In this case, we don't care if there isn't a file named **program** to delete; we just want to delete them if they exist.
3. The **@** in front of a command tells **make** to not print the command to the screen when **make** executes it. We use it here so the output of **make clean** doesn't clutter up the screen.

---

<sup>17</sup>The **clean** target is commonly used in anger when the dang compiler isn't working right and you're not sure why. Maybe re-doing the whole process from scratch will fix things. (It probably won't, but hey, it's worth a shot!)

<sup>18</sup>In other words, a non-zero return value from that command.

## Variables

When writing more complex makefiles, you will want to use variables so that you can easily change targets in the future. It is common to use variables to hold lists of files, compiler flags, and even programs!

Here’s the syntax for variables:

- `var=value` sets `var` to ‘value’.
- `${var}` or `$(var)` accesses the value of `var`.<sup>19</sup>
- The line `target: var=thing` sets the value of `var` to `thing` when building `target` and its dependencies.

For example, let’s suppose we want to add some flags to `g++` in our example. Furthermore, we want to have two sets of flags: one for a “release” build, and one for a “debug” build. Using the “release” build will result in a fast and lean program, but compiling might take a while. The “debug” build will compile faster and include debug information in our program.

We’ll make a `CFLAGS` variable to hold the “release” flags and a `DEBUGFLAGS` variable to hold our “debug” flags. That way, if we want to change our flags later on, we only need to look in one spot. We’ll also add a phony `debug` target so that running `make debug` builds our program in debug mode.

```
CFLAGS = -O2
DEBUGFLAGS = -g -Wall -Wextra

program: main.o funcs.o
    g++ ${CFLAGS} main.o funcs.o -o program

.PHONY: debug clean
debug: CFLAGS=${DEBUGFLAGS}
debug: program

main.o: main.cpp funcs.h
    g++ ${CFLAGS} -c main.cpp

funcs.o: funcs.cpp funcs.h
    g++ ${CFLAGS} -c funcs.cpp

clean:
    -@ rm -f program *.o
```

Note that the `debug` target doesn’t actually have any commands to run; it just changes the `CFLAGS` variable and then builds the `program` target. (We’ll talk about the `-g` flag in the next chapter. It’s quite cool.)

---

<sup>19</sup>Be careful not to confuse this with `bash`’s `$()`, which executes whatever is between the parentheses.

As an aside, since **make** only tracks the modification times of files, when switching from doing a debug build to a release build or vice-versa, you will want to run **make clean** first so that **make** will rebuild all your code with the appropriate compiler flags.

## Pattern Targets

You've probably noticed that our example so far has had an individual target for each object file. If you had more files, adding all those targets would be a lot of work. Instead of writing each of these out manually, **make** supports pattern targets that can save you a lot of work.

Before we set up a pattern target, we need some way to identify the files we want to compile. **make** supports a wide variety of fancy variable assignment statements that are incredibly handy in combination with pattern targets.

You can store the files that match a glob expression using the **wildcard** function: **cppfiles=\$(wildcard \*.cpp)** stores the name of every file ending in **.cpp** in a variable named **cppfiles**.

Once you have your list of C++ files, you may want a list of their associated object files. You can do pattern substitution on a list of filenames like so: **objects=\$(cppfiles:%.cpp=%.o)**. This will turn your list of files ending in **.cpp** into a list of files ending in **.o** instead!

When writing a pattern rule, as with substitution, you use **%** for the variable part of the target name. For example: **%.o: %.cpp** creates a target that builds a **.o** file from a matching **.cpp** file.

**make** has several special variables that you can use in any target, but are especially handy for pattern targets:

- **\$@**: The name of the target.
- **\$<**: The name of the first dependency.
- **\$^**: The names of all the dependencies.

Let's rewrite our example using a pattern target to make the object files:

```
SOURCES=$(wildcard *.cpp)
OBJECTS=$(SOURCES:%.cpp=%.o)
HEADERS=$(wildcard *.h)
```

```
program: ${OBJECTS}
    g++ $^ -o program
```

```
%.o: %.cpp ${HEADERS}
    g++ -c $<
```

Now our `program` target depends on all the object files which are calculated from the names of all the `.cpp` files in the current directory. The target for each object file depends on the associated `.cpp` file as well as all the headers.<sup>20</sup> With this pattern rule, you won't need to update your makefile if you add more files to your program later!

## Useful make Flags

`make` has a few flags that come in handy from time to time:

- `-j3` runs up to 3 jobs in parallel.<sup>21</sup>
- `-B` makes targets even if they seem up-to-date. Useful if you forget a dependency or don't want to run `make clean`!
- `-f <filename>` uses a different makefile other than one named `makefile` or `Makefile`.

---

<sup>20</sup>This is because header files might be included in several places. If you're interested in more accurately calculating dependencies, check out the `makedepend` program.

<sup>21</sup>The general rule of thumb for the fastest builds is to use one more job than you have CPU cores. This makes sure there's always a job ready to run even if some of them need to load files off the hard drive.

## Questions

Name: \_\_\_\_\_

Consider the following makefile:

```
default: triangles
```

```
triangles: main.o TrianglePrinter.o funcs.o  
    g++ $^ -o triangles
```

```
main.o: main.cpp TrianglePrinter.h funcs.h  
    g++ -c main.cpp
```

```
TrianglePrinter.o: TrianglePrinter.cpp TrianglePrinter.h funcs.h  
    g++ -c TrianglePrinter.cpp
```

```
funcs.o: funcs.cpp funcs.h  
    g++ -c funcs.cpp
```

1. If you run `make`, what files get built?

2. If you change `TrianglePrinter.h`, what targets will need to be rebuilt?

## Quick Reference

## Further Reading

- [The GNU Make Manual](#)
- [Special Variables](#)

There are a couple of programs that can help generate makefiles for you:

- [makedepend](#) computes dependencies in C and C++ code
- [CMake](#) generates makefiles and various IDE project files



## Chapter 8

# Debugging with GDB

### Motivation

You thought you were getting a bargain when you bought that pocket watch from that scary old lady at the flea market in the French Quarter. Little did you realize that pocket watch did more than just tell time<sup>1</sup>. In fact, holding the button on the side *slows down* time.

At first it's a novelty. You prank your roommate. Maybe take some extra time on a test. Steal a quick bagel for breakfast. Rob a bank<sup>2</sup>...



Figure 8.1: When you can slow down time, taking candy from a baby is like taking candy from a baby.

---

<sup>1</sup>Actually it doesn't tell time at all. The hands didn't move when you bought it, and they still don't. It sure looks cool, though.

<sup>2</sup>That escalated pretty quickly.

Desperate for a way out, you start mashing and twisting the other buttons and dials on your magic pocket watch.

With a deafening *bwee*, you're hurled back in time. Once again, you're crouched in front of the control panel for the Big Bank's vault. You stare, bewildered, at the room around you.

Blue wire.

The array of lasers begins slowly cutting across the doorway. You're trapped again. With your last few seconds of freedom, you look in the panel and find *another blue wire*. You cut the wrong blue wire!

You reach into your pocket to grab your watch and reset time. Instead, you pull out that pastrami sandwich from last lab. Also you've turned into a raccoon somehow.

In real life, you may not be able to slow down or reset time analyze disasters in detail<sup>5</sup>. You can, however, slow down disasters as they happen in your programs.

Using a **debugger** you can slow down the execution of your programs to help you figure out why it's not working. A debugger can't automatically tell you what's broken, but it can...

- step through your code line by line
- show you the state of variables
- show you the contents of memory

...so that you can figure out what's wrong on your own.

In your bank robber dream, the watch allowed you to slow down time to see what set off the laser defense system. In real life, a debugger allows you to slow down the execution of your program, so that you can see where and when it breaks. It's still up to you to figure out why it's breaking and how to fix your

<sup>4</sup>It was a dream that whole time. What a slap in the face!

<sup>5</sup>In real life, hopefully you're not a bank robber either.

bugs, but a debugger can definitely give you some clues to make it easier to find them.

In this lab, we'll be using the GNU Debugger (**gdb**) to debug a couple of C++ programs. **gdb**, like **g++**, is open source software. There are GUI frontends available for **gdb**, but in this class, we'll be using the command line interface.

## Takeaways

- Learn to debug C++ projects with **gdb**
  - Step through the execution of a compiled C++ program
  - Inspect the contents of program variables

## Walkthrough

## Compilin' for Debuggin'

When you compile a program, you lose a lot of information. All of the C++ code that you write gets translated into machine-executable code. As far as your CPU is concerned, there's really no need for function names, curly braces, or comments.

As a human person, you will need those things.

If we plan to debug our code with `gdb`, we need to ask `g++` to keep the details of our source code when we compile it. Whenever you compile your code, simply add the `-g` flag to your `g++` command. For example:

```
$ g++ -g main.cpp
```

If you forget the `-g`, `gdb` will have a lot less information to work with. As a result, debugging will be much less useful.

## Starting GNU Debugger

Alrighty, friend. Let's get our hands dirty.

Pop open an editor and drop in the following C++ program. Save it as `main.cpp`. **Make sure** you keep all the whitespace the same. We're going to need to refer to line numbers when we use `gdb`.

```
#include <iostream> // 1
// 2
using namespace std; // 3
// 4
int main() // 5
```

```

{                                                    // 6
    int x = 7;                                       // 7
    if (x < 10)                                       // 8
        cout << "Less than 10!" << endl;           // 9
                                                    // 10
    if (x >= 10)                                       // 11
        cout << "Greater than or equal to 10!" << endl; // 12
                                                    // 13
    return 0;                                         // 14
}                                                    // 15

```

Now let's compile it and run it in `gdb`.

```

$ g++ -g -o main main.cpp
$ gdb main
GNU gdb ...
Reading symbols from main...done.
(gdb)

```

`gdb` will show you a **bunch** of license information and other junk before it give you its command prompt. You can ignore that stuff.

You're now in `gdb`! Here, you can issue commands to `gdb`, and it will do your bidding. You can step through your program, run parts of it at full speed, and check the values of different variables.

## GNU Debugger Commands

### Running programs

Assuming you followed the directions in the previous section, you should be sitting at the `(gdb)` prompt ready to run your first command.

Try issuing the `run` command. Just type `run` and press enter. Be sure to make a note of the output for the questions section.

The `run` command will start your program and execute it at full speed until it reaches a stopping condition. Stopping conditions include:

1. reaching the end of a program.
2. reaching a breakpoint.
3. encountering a fatal error (like a segmentation fault).

In order to debug your program, it has to be running. The `run` command is likely one of the first commands you will run whenever you debug a program.

If you need to pass any command line arguments to the program, you'll pass them to the `run` command. For example, if we wanted to debug `g++`<sup>6</sup>...

---

<sup>6</sup>Don't actually do this, though. We're just demonstrating how you'd pass command line

```
$ gdb g++
(gdb) run funcs.cpp main.cpp
```

... would be similar to running `g++ funcs.cpp main.cpp` outside of `gdb`.

### Stopping at the right time

If you don't want your program to pause at a specific line, you can place a **breakpoint**.

Assuming your program is not currently running, let's place a breakpoint at line 8 of `main.cpp`. Then, we'll run our program.

```
(gdb) break main.cpp:8
(gdb) run
Starting program: main
```

```
Breakpoint 1, main () at main.cpp:8
if (x < 10)
(gdb)
```

Our program **is running**, but `gdb` has paused its execution at line 8.

You can set as many breakpoints as you like. The **info breakpoints** command will show you a list of all the breakpoints you've set.

If you find that you have too many breakpoints, or if they're getting in the way, you can delete them using the **delete** command. By itself, **delete** will delete all breakpoints. If you want to delete a specific breakpoint, you can refer to it by its ID number. For example:

```
(gdb) info breakpoints
1          breakpoint      keep y   0x000000000040086c in main() at main.cpp:8
2          breakpoint      keep y   0x000000000040088e in main() at main.cpp:10
3          breakpoint      keep y   0x0000000000400894 in main() at main.cpp:12
(gdb) delete 2
(gdb) info breakpoints
1          breakpoint      keep y   0x000000000040086c in main() at main.cpp:8
3          breakpoint      keep y   0x0000000000400894 in main() at main.cpp:12
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) info breakpoints
No breakpoints or watchpoints.
```

---

arguments.

### Where are we?

You can use the `backtrace` command to ask `gdb` where we currently are.

```
(gdb) backtrace
#0  main () at main.cpp:8
```

In the output, `gdb` will show you the current backtrace. Depending how many functions you've called, you'll see deeper and deeper traces. In this case, we've only called `main()`, so that's the only line we see. You can also see the file name and line number.

You can also use `list` to ask `gdb` to show you some source code to give you context. Sometimes a line number isn't enough if you're too lazy to tab over to your text editor.

```
(gdb) backtrace
#0  main () at main.cpp:8
(gdb) list
3  using namespace std;
4
5  int main()
6  {
7      int x = 7;
8      if (x < 10)
9          cout << "Less than 10!" << endl;
10
11     if (x >= 10)
12         cout << "Greater than or equal to 10!" << endl;
```

An interesting quirk about `list` is that it will continue to show more lines if you run it again. If it runs out of lines to show you, it becomes grumpy.

```
(gdb) list
13
14     return 0;
15 }
(gdb) list
Line number 16 out of range; main.cpp has 15 lines.
```

### Stepping through the code

Now that we've used a breakpoint to pause our code, we can step through the execution. There are a handful of commands that will help us do this:

- **continue:** Resume running the program at full speed. We'll only stop/pause execution if we reach a stopping condition as described for `run`.

- **step**: Runs **one** line of code, stepping **into** function calls. If you reach a function call, **step** will enter that function.
- **next**: Runs **one** line of code, stepping **over** function calls. If you reach a function call, **next** will run the entire function until it returns.
- **finish**: Runs code until the current function returns.

Assuming you're following along, we should be paused at line 8 of `main.cpp`. You can verify this with **backtrace**. If necessary, run **kill** to stop debugging followed by **run** again.

Try running **step** three times. Make note of the line you end up on.

### Looking at contents of variables

`gdb` also allows you to look at what's stored in different variables. This can be a *very* handy alternative to placing `cout` statements all over the place.

Let's make sure we're on the same metaphorical page here:

1. Run **kill** to stop your program.
2. Run **delete** to delete all of your breakpoints.
3. Run **break main.cpp:7** to create a new breakpoint at line 7.
4. Run **run**

`gdb` should pause execution at line 7 (`int x = 7;`). Now let's use the **print** command to have a look at the contents of `x`.

```
(gdb) print x
$1 = 1231764817
```

Well that's interesting, isn't it? 1231764817 is not 7 at all! That's because at this point, `gdb` hasn't actually run line 7 yet. Let's step forward and check it again.

```
(gdb) step
8   if (x < 10)
(gdb) print x
$2 = 7
```

That makes a lot more sense!

You can use **print** to print out any variable that's in scope. You can also use **info locals** to check on the value of local variables. Try running it and make a note of the output.

## Questions

Name: \_\_\_\_\_

1. What was the output from using the `run` command?
2. How would you set a breakpoint for line 10 of file `my_funcs.cpp`?
3. What line number did you end up on after you ran `step` three times?
4. What was the output from running `info locals`?



## Quick Reference

### Using gdb

- `gdb PROGRAM` launches the debugger for debugging `PROGRAM`
  - **Note:** You will want to pass `g++` the `-g` flag when you compile!
- `run arg1 arg2` runs the command with command line arguments `arg1` and `arg2`
- `backtrace` or `bt` shows the call stack

### Setting breakpoints with gdb

- `break main.cpp:10` will stop execution whenever line 10 in `main.cpp` is reached.
- `continue` resumes running as normal.
- `step` runs one more line of code; steps **into** functions as necessary.
- `next` runs until execution is on the next line; steps **over** functions.
- `finish` runs until the current function returns.
- `delete` removes all breakpoints.

### Looking at variables with gdb

- `print VARIABLE` prints the contents of variable with name `VARIABLE`
  - `p VARIABLE` also works
  - `p` also works with expressions of just about any sort
- `x address` examines one word memory at a given address
- `x/2 address` examines two words of memory
- `info registers` lists all CPU register values
- `p $REGNAME` prints the value of a CPU register

### Miscellaneous

- Conditional Breakpoints: `condition BPNUMBER EXPRESSION`
- Editing variables at runtime with gdb:
  - `set var VARIABLE = VALUE` assigns `VALUE` to `VARIABLE`
  - `set {int}0x1234 = 4` writes 4 (as an integer) to the memory address 0x1234
- Disassembling code: `disassemble FUNCTION` prints the assembly for a function named `FUNCTION`

## Further Reading

- [More on breakpoints](#)

## Chapter 9

# Finding Memory Safety Bugs

### Motivation

It's the night before the Data Structures linked list assignment is due, and you are so ready. Not only do you *understand* linked lists, you *are* a linked list. You sit down at your terminal<sup>1</sup>, crack your knuckles, and (digitally) pen a masterpiece. Never before in the history of computer science has there been such a succinct, elegant linked list implementation.

You compile it and run the test suite, expecting a beautiful **All Tests Passed!**. Instead, you are greeted with a far less congratulatory **Segmentation Fault (core dumped)**. I guess that's what you get for expecting a machine to appreciate beauty!

A more pragmatic reason for that segmentation fault is that somewhere your program has accessed memory it didn't have permission to access. Segmentation faults are but one kind of memory safety bug. Other memory bugs tend to be less immediately obvious, but they can introduce hard-to-find bugs. In this chapter we will explore the different types of memory safety bugs you may encounter as well as tools for detecting and analyzing them.

There is another incentive for ruthlessly excising memory safety bugs: every kind of memory safety bug allows an attacker to use it to exploit your program. Many of these bugs allow for arbitrary code execution, where the attacker injects their own code into your program to be executed.

The first widespread internet worm, the [Morris worm](#), used a memory safety bug to infect other machines in 1988. Unfortunately, even after almost *thirty*

---

<sup>1</sup>And the computer it is running on, being that it's the 21st century and all.

years, memory safety bugs are incredibly common in popular software and many viruses still use memory safety bugs. For one example, the [WannaCry](#) and [Petya](#) viruses use a memory safety exploit called [EternalBlue](#) “allegedly” developed by the NSA and released by “Russian” hackers early in 2017.



Figure 9.1: Smokey Says: “Only You Can Prevent Ransomware”

## Takeaways

- Learn about how memory is managed in programs
- Learn four common memory-related bugs
- Use Valgrind, Address Sanitizer, and GCC to help find and diagnose these bugs

## Walkthrough

### The Stack and The Heap

Your operating system provides two areas where memory can be allocated: the stack and the heap. Memory on the stack is managed automatically<sup>2</sup>, but any allocation only lives as long as the function that makes it. When a function is called, a *stack frame* is pushed onto the stack. The stack frame holds variables as well as some bookkeeping information. Crucially, this information includes the memory address of the code to return to once the function completes. When that function **returns**, its stack frame is popped off the stack and the associated memory is used for the stack frame of the next called function.

Memory allocated on the heap lives as long as you like it to; however, you have to manually allocate and free that memory using **new** and **delete**.<sup>3</sup> While the automatic management of the stack is nice, the freedom of being able to

<sup>2</sup>It's a joint effort between how the compiler compiles your code and the operating system.

<sup>3</sup>Or if you're writing C, with the **malloc** and **free** functions.

make memory allocations that live longer than the function that created them is essential, especially in large programs.

On modern Intel CPUs, the stack starts at a high memory address and grows downward, while the heap starts at a low memory address and grows upward. The addresses they start at vary from system to system, and are often randomized to make writing exploits more difficult.

## Uninitialized Values

When you allocate memory, either on the stack or on the heap, the contents of that memory is undefined. Maybe it's a bunch of zeros; maybe it's some weird looking garbage; maybe it's a password because the last time that memory was used, a program stored a password there. You don't know what's there, and you'd be foolhardy to use that value for anything.<sup>4</sup> In particular, you *really* don't want to use such a value in a condition for an `if` statement or a loop! Doing so gives control over your program to anyone can control the values of your uninitialized variables.

Even worse, in C++, accessing uninitialized memory is *undefined behavior* and thus if the compiler catches you doing it,<sup>5</sup> it can do whatever it wants, including `rm -rf /`, playing mean taunts over your speakers to you, or just removing any code that depends on the value of an uninitialized variable from your executable.

So, using uninitialized values in your program can introduce weird seemingly-random bugs or result in certain features disappearing from your code that you know for sure are there. In other words: initialize your dang variables!

Here's an example of uninitialized values, one on the stack and one on the heap:

```
#include<iostream>
using namespace std;

int main()
{
    cout << "Stack uninitialized value" << endl;
    int x;
    if(x > 5)
    {
        cout << "> 5" << endl;
    }

    cout << "Heap uninitialized value" << endl;
```

---

<sup>4</sup>It's not even a good source of random numbers, unless you like random numbers that aren't very random.

<sup>5</sup>It doesn't always do this because statically analyzing software (i.e., at compile time) is Really Hard, but it still catches some stuff.

```

int* y = new int;
if(*y < 5)
{
    cout << "< 5" << endl;
}

delete y;

return 0;
}

```

Your first hint that this isn't right is from the compiler itself if you use the `-Wall` flag:<sup>6</sup>

```

$ g++ -Wall -g uninitialized-value.cpp -o uninitialized-value
uninitialized-value.cpp: In function 'int main()':
uninitialized-value.cpp:8:3: warning: 'x' is used uninitialized ↵
    in this function [-Wuninitialized]
    if(x > 5)
    ^

```

Here GCC is smart enough to catch the uninitialized use of our stack-allocated variable. (This warning doubles as a subtle reminder that if you start asking GCC to optimize this code, that `if` statement is probably going to be removed!)

Valgrind's Memcheck tool<sup>7</sup> can detect when your program uses a value uninitialized. Memcheck can also track where the uninitialized value is created with the `--track-origins=yes` option. If we run the above program (named `uninitialized-values`) through Valgrind (`valgrind --track-origins=yes uninitialized-values`), we get two messages.

The stack-allocated uninitialized value was accessed on line 8 and created on line 5:

```

==19296== Conditional jump or move depends on uninitialised value(s)
==19296==    at 0x4008FE: main (uninitialized-value.cpp:8)
==19296==    Uninitialised value was created by a stack allocation
==19296==    at 0x4008D6: main (uninitialized-value.cpp:5)

```

Line 8 is in fact `if(x > 5)`. All stack-allocated variables appear to be allocated at the start of the function call, so `x` is listed as being created on line 5 rather than line 7.

The heap-allocated uninitialized value was accessed on line 15 and created by a call to `new` on line 14:

---

<sup>6</sup>Lines that are too long to fit on the page have been split over multiple lines; this is indicated by the ↵ symbol.

<sup>7</sup>Valgrind has a whole bunch of tools included, but it runs the Memcheck tool by default. We'll see some other Valgrind tools in future chapters of this book.

```

==19296== Conditional jump or move depends on uninitialised value(s)
==19296==    at 0x40094F: main (uninitialized-value.cpp:15)
==19296== Uninitialised value was created by a heap allocation
==19296==    at 0x4C2E0EF: operator new(unsigned long)
==19296==    by 0x400941: main (uninitialized-value.cpp:14)

```

Heap-allocated uninitialized values<sup>8</sup> cannot be caught by the compiler – you must use a tool like Valgrind to find them.

Using `--track-origins=yes` is particularly handy when debugging heap uninitialized values as it is possible for something to be `new`'d in one function and then not used until much later on.

## Unallocated or Out-of-Bounds Reads and Writes

Perhaps the most common memory bug is reading or writing to memory you ought not to. This type of bug comes in a few flavors: you could use a pointer with an uninitialized value, or you could access outside of an array's bounds, or you could use a pointer after deleting the thing it points at.

Sometimes this kind of error causes a segmentation fault, but sometimes the memory being accessed happens to be something else your program is allowed to access. In this case, these memory bugs can go about their business, silently mangling your data and making your program do very bizarre things.

Furthermore, these types of bugs can easily turn into exploits. Remember that at the top of each stack frame is the address of the code to jump to when the associated function returns. An out-of-bounds write lets an attacker overwrite this address with their own! Once they have this, they can have the computer start executing whatever code they desire as long as they know where it is in memory. This kind of exploit is known as a buffer overflow exploit.

You can detect these kinds of bugs using either Valgrind or Address Sanitizer (a.k.a. `asan`). `asan` is part library, part compiler feature that instruments your code at compile time. Then when you run your program, the instrumentation tracks memory information much in the way Valgrind does. `asan` is much faster than Valgrind, but requires special compiler flags to work.

To enable address sanitizer, you must use the following flags to `g++`: `-g -fsanitize=address -fno-omit-frame-pointer`.<sup>9</sup> Furthermore, you need to set two environment variables<sup>10</sup> if you want function names and line numbers to appear in `asan`'s output:

---

<sup>8</sup>And more generally, any uninitialized value being accessed through a pointer.

<sup>9</sup>Note: don't try to use `asan` and Valgrind at the same time. Your output will be `craaaaaazy`. It's best to make a special `asan` makefile target that turns on the relevant compiler flags.

<sup>10</sup>This requires `llvm` to be installed. Also, depending on the system you are running, you may need to append a version number, e.g., `export ASAN_SYMBOLIZER_PATH=`which llvm-symbolizer-3.9``

```
export ASAN_SYMBOLIZER_PATH=`which llvm-symbolizer`
export ASAN_OPTIONS=symbolize=1
```

Let’s look at some examples of this class of bugs and the relevant Valgrind and `asan` output. First up, out-of-bounds accesses on a stack-allocated array:

```
#include<iostream>

int main()
{
    int array[5];
    array[5] = 5; // Out-of-bounds write
    std::cout << array[5] << std::endl; // Out-of-bounds read

    return 0;
}
```

If you run this program normally, it probably won’t crash, and in fact it will probably behave how you expect. This is a mirage. It only works because whatever is one `int` after `array` in `main()`’s stack frame happens to not be used again. This illustrates how important it is to check that you do not have these bugs! Even worse, Valgrind does not detect this out-of-bounds access!

However, `asan` does. Its output is somewhat terrifying to see, but the relevant parts look like this:<sup>11</sup>

```
==29210==ERROR: AddressSanitizer: stack-buffer-overflow on ↵
    address 0x7fff2f9d9654 at pc 0x000000400ce4 bp 0x7fff2f9d9610 ↵
    sp 0x7fff2f9d9600
WRITE of size 4 at 0x7fff2f9d9654 thread T0
    #0 0x400ce3 in main invalid-stack.cpp:6
    #1 0x7fa90759b82f in __libc_start_main
    #2 0x400b58 in _start (invalid-stack+0x400b58)

Address 0x7fff2f9d9654 is located in stack of thread T0 ↵
    at offset 52 in frame
    #0 0x400c35 in main invalid-stack.cpp:4

This frame has 1 object(s):
    [32, 52) 'array' <== Memory access at offset 52 ↵
                                overflows this variable
```

In particular, we have a write of “size 4”, that is, 4 bytes, on line 6 of our program. The stack trace showing how execution reached line 6 is shown as well. Furthermore, the stack variable we wrote out-of-bounds to was allocated on line 4 as part of `main()`’s stack frame, which contains one object: `array`.

<sup>11</sup>The file paths shown have been cut down to fit on the page, and some have been removed entirely to clean up the output. Don’t be concerned if the output you see is slightly different from what is printed here.



Address sanitizer halts the program after the first error, so we don't see output for the invalid read. The reason for this behavior is that generally one bug causes others, and it's easier to just see the first problem and fix it rather than wade through screenfuls of errors trying to figure out which one unleashed the torrent of access violations.

Pretty handy, eh? What more could you ask for!

Both Valgrind and `asan` can detect heap out-of-bounds accesses. Here is a small sample program that demonstrates an out-of-bounds write:

```
#include<iostream>

int main()
{
    std::cout << "Invalid write" << std::endl;
    int *arr = new int[5];
    for(int i = 0; i <= 5; i++)
    {
        arr[i] = i;
    }

    delete[] arr;
    return 0;
}
```

Here is Valgrind's output:

```
==2894== Invalid write of size 4
==2894==    at 0x40097C: main (invalid.cpp:9)
==2894== Address 0x5ac00d4 is 0 bytes after a block of size 20 alloc'd
==2894==    at 0x4C2E80F: operator new[](unsigned long)
==2894==    by 0x400953: main (invalid.cpp:6)
```

And here is `asan`'s output:

```
==3334==ERROR: AddressSanitizer: heap-buffer-overflow on ↵
    address 0x60300000eff4 at pc 0x000000400ce1 bp 0x7ffee305a690 ↵
    sp 0x7ffee305a680
WRITE of size 4 at 0x60300000eff4 thread T0
    #0 0x400ce0 in main invalid.cpp:9
    #1 0x7fc6258f282f in __libc_start_main
    #2 0x400b88 in _start (invalid+0x400b88)

0x60300000eff4 is located 0 bytes to the right of 20-byte region ↵
    [0x60300000efe0,0x60300000eff4) allocated by thread T0 here:
    #0 0x7fc6260bf6b2 in operator new[](unsigned long)
    #1 0x400c83 in main invalid.cpp:6
    #2 0x7fc6258f282f in __libc_start_main
```

Since the memory we are accessing out-of-bounds is heap allocated, Valgrind and `asan` can track the exact line of your code where it was allocated (in this case, line 6). The write itself occurred on line 9. Furthermore, they show that the write happened 0 bytes to the right<sup>12</sup> (in other words, after the end of) our allocated chunk, indicating that we are writing one index past the end of the array.

Finally, let's see an example of a use-after-free. This type of bug is exploitable by means similar to using an uninitialized value, but it is usually far easier to control the contents of memory for a use-after-free bug.<sup>13</sup> Like out-of-bounds accesses, this type of bug can go undetected; the below example appears to work, even though it is incorrect!

```
#include<iostream>

int main()
{
    int* x = new int[5]; // Declare and initialize an array
    for(int i = 0; i < 5; i++)
    {
        x[i] = i;
    }

    int* y = &x[1]; // "Accidentally" hold a pointer to an array element

    delete [] x; // Delete the array

    std::cout << *y << std::endl; // Uh-oh!

    return 0;
}
```

Valgrind shows where the invalid read occurs, where the block was deallocated, and where it was allocated:

```
==10658== Invalid read of size 4
==10658==    at 0x40090B: main (use-after-free.cpp:15)
==10658== Address 0x5abfc84 is 4 bytes inside a block of size 20 free'd
==10658==    at 0x4C2F74B: operator delete[](void*)
==10658==    by 0x400906: main (use-after-free.cpp:13)
==10658== Block was alloc'd at
==10658==    at 0x4C2E80F: operator new[](unsigned long)
==10658==    by 0x4008B7: main (use-after-free.cpp:5)
```

<sup>12</sup>Did you know that chickens also visually organize smaller quantities on the left and larger quantities on the right?

<sup>13</sup>Since this is not a book on exploiting software, we won't go into further detail; writing exploits is its own universe of rabbit holes.

Address Sanitizer's output is similar:

```

==10827==ERROR: AddressSanitizer: heap-use-after-free on ↵
    address 0x60300000efe4 at pc 0x000000400ca6 bp 0x7ffce3f2c0e0 ↵
    sp 0x7ffce3f2c0d0
READ of size 4 at 0x60300000efe4 thread T0
    #0 0x400ca5 in main use-after-free.cpp:15
    #1 0x7f7c327c682f in __libc_start_main
    #2 0x400b08 in _start (use-after-free+0x400b08)

0x60300000efe4 is located 4 bytes inside of 20-byte region ↵
[0x60300000efe0,0x60300000eff4)
freed by thread T0 here:
    #0 0x7f7c32f93caa in operator delete[](void*)
    #1 0x400c6e in main use-after-free.cpp:13
    #2 0x7f7c327c682f in __libc_start_main

previously allocated by thread T0 here:
    #0 0x7f7c32f936b2 in operator new[](unsigned long)
    #1 0x400be7 in main use-after-free.cpp:5
    #2 0x7f7c327c682f in __libc_start_main

```

Both outputs show that a 4-byte (i.e., `int`) read happened 4 bytes (i.e., at index 1) inside our block of 20 bytes that is our array of 5 ints.

## Mismatched and Double Deletes

Mismatched deletes occur when you use `delete` to delete an array or `delete []` to delete a non-array. In the former case, not all allocated memory may be marked as free, resulting in a memory leak. In the latter case, too much memory may be freed!<sup>14</sup>

A simple example calls `delete` on something allocated with `new[]`:

```

int main()
{
    int* arr3 = new int[5];
    delete arr3;

    return 0;
}

```

Valgrind reports the error like so:

---

<sup>14</sup>The implementation of `delete []` isn't specified, but the size of the allocation is stored somewhere; depending on where it is stored, various Bad Things can happen if you try to `delete []` something that wasn't intended to be.

```

==16964== Mismatched free() / delete / delete []
==16964==    at 0x4C2F24B: operator delete(void*)
==16964==    by 0x400667: main (mismatched.cpp:4)
==16964== Address 0x5abfc80 is 0 bytes inside a block of size 20 alloc'd
==16964==    at 0x4C2E80F: operator new[](unsigned long)
==16964==    by 0x400657: main (mismatched.cpp:3)

```

And here is Address Sanitizer's output:

```

==17080==ERROR: AddressSanitizer: alloc-dealloc-mismatch ↵
(operator new [] vs operator delete) on 0x60300000efe0
#0 0x7f8cde981b2a in operator delete(void*)
#1 0x400747 in main mismatched.cpp:4
#2 0x7f8cde53e82f in __libc_start_main
#3 0x400658 in _start (mismatched+0x400658)

0x60300000efe0 is located 0 bytes inside of 20-byte region ↵
[0x60300000efe0,0x60300000eff4)
allocated by thread T0 here:
#0 0x7f8cde9816b2 in operator new[](unsigned long)
#1 0x400737 in main mismatched.cpp:3
#2 0x7f8cde53e82f in __libc_start_main

```

Both identify where the delete and matching allocation occurred (here, on lines 4 and 3 respectively). You can tell what the exact mismatch is by looking at the operators called by the deletion and allocation lines. In this example, **operator delete** is called to delete the allocation, but **operator new[]** is called to allocate it.

Double deletes may seem innocuous, but they can be easily turned into a use-after-free bug. This is because freed memory is usually re-used in future allocations. So deleting something, then allocating a second thing, then deleting the first thing again results in the second thing being deleted! Any future uses of the second thing then become a use-after-free problem, and attempting to properly clean up that second allocation brings on a double delete. For example,

```

#include<iostream>

int main()
{
    int* x = new int(5);
    delete x;

    int* y = new int(7); // Probably allocated where x was
    // If we were to print out *x and *y, we would likely
    // see "7" for both values!

    delete x; // Either deletes y or explodes

```

```

    delete y; // If the last line didn't explode, BOOM!

    return 0;
}

```

(We don't print the values out because Valgrind and Address Sanitizer would discover a use-after-free on `x`, which is not what we're trying to demonstrate here.)

Valgrind properly attributes the double free to line 12 (the second `delete x`):

```

==20974== Invalid free() / delete / delete[] / realloc()
==20974==    at 0x4C2F24B: operator delete(void*)
==20974==    by 0x40078D: main (double.cpp:12)
==20974== Address 0x5abfc80 is 0 bytes inside a block of size 4 free'd
==20974==    at 0x4C2F24B: operator delete(void*)
==20974==    by 0x40076D: main (double.cpp:6)
==20974== Block was alloc'd at
==20974==    at 0x4C2E0EF: operator new(unsigned long)
==20974==    by 0x400757: main (double.cpp:5)

```

As does asan:

```

==20761==ERROR: AddressSanitizer: attempting double-free ↵
    on 0x60200000eff0 in thread T0:
      #0 0x7f0c58dbdb2a in operator delete(void*)
      #1 0x400adb in main double.cpp:12
      #2 0x7f0c585f082f in __libc_start_main
      #3 0x400958 in _start (double+0x400958)

0x60200000eff0 is located 0 bytes inside of 4-byte region ↵
[0x60200000eff0,0x60200000eff4)
freed by thread T0 here:
      #0 0x7f0c58dbdb2a in operator delete(void*)
      #1 0x400a84 in main double.cpp:6
      #2 0x7f0c585f082f in __libc_start_main

previously allocated by thread T0 here:
      #0 0x7f0c58dbd532 in operator new(unsigned long)
      #1 0x400a37 in main double.cpp:5
      #2 0x7f0c585f082f in __libc_start_main

```

Both show the location of the allocation and the first delete. Typically, this kind of bug arises when you don't properly keep track of whether a pointer has been deleted yet.

## Memory Leaks

Last, but certainly not least, are memory leaks. While these pose less potential for security flaws, nobody likes programs that hog memory. Just like it's good practice to close files when you're done accessing them, it's good practice to deallocate memory when you're done using it.

There are two kinds of memory leaks: direct and indirect. A direct memory leak occurs when you have allocated a block of memory but no longer have a pointer pointing to it. An indirect memory leak occurs when the only pointers to an allocated block of memory are in a block of memory that has been leaked. The distinction is drawn because typically indirect memory leaks occur due to not running a destructor on some directly leaked object.

Both Valgrind and Address Sanitizer can detect memory leaks. Let's look at a simple example that has one directly leaked block and one indirectly leaked block:

```
struct List
{
    int value;
    List* next;
};

int main()
{
    List l;
    l.value = 5;
    l.next = new List;

    l.next->value = 6;
    l.next->next = new List;

    //These two lines would fix the memory leaks
    //delete l.next->next;
    //delete l.next;

    return 0;
}
```

When using Valgrind to debug memory leaks, the `--leak-check=full` option shows where each leaked block was allocated.

Valgrind's output, with a full leak check, is shown below:

```
==649== HEAP SUMMARY:
==649==      in use at exit: 72,736 bytes in 3 blocks
==649==    total heap usage: 3 allocs, 0 frees, 72,736 bytes allocated
```

```

==649==
==649== 32 (16 direct, 16 indirect) bytes in 1 blocks are definitely ↵
        lost in loss record 2 of 3
==649==    at 0x4C2E0EF: operator new(unsigned long)
==649==    by 0x40060F: main (leak.cpp:11)
==649==
==649== LEAK SUMMARY:
==649==    definitely lost: 16 bytes in 1 blocks
==649==    indirectly lost: 16 bytes in 1 blocks
==649==    possibly lost: 0 bytes in 0 blocks
==649==    still reachable: 72,704 bytes in 1 blocks
==649==    suppressed: 0 bytes in 0 blocks

```

On some systems, including this one, the system runtime library allocates some memory and does not deallocate it.<sup>15</sup> This memory will appear in the “still reachable” section of Valgrind’s output. Do not worry yourself too much about it.

The big thing we’re disturbed to see is that we have leaked 32 bytes: 16 directly and 16 indirectly. The directly-leaked block that leaks our indirectly-leaked block was allocated on line 11 (`l.next = new List`). Valgrind does not show the location that indirectly-leaked blocks are allocated.

Address Sanitizer also has a leak checker; it does not track “still reachable” memory, so there are no false positives here:

```

==733==ERROR: LeakSanitizer: detected memory leaks

```

```

Direct leak of 16 byte(s) in 1 object(s) allocated from:

```

```

    #0 0x7f4e3b152532 in operator new(unsigned long)
    #1 0x4008cf in main leak.cpp:11
    #2 0x7f4e3ad0f82f in __libc_start_main

```

```

Indirect leak of 16 byte(s) in 1 object(s) allocated from:

```

```

    #0 0x7f4e3b152532 in operator new(unsigned long)
    #1 0x40091c in main leak.cpp:14
    #2 0x7f4e3ad0f82f in __libc_start_main

```

```

SUMMARY: AddressSanitizer: 32 byte(s) leaked in 2 allocation(s).

```

As opposed to Valgrind, Address Sanitizer shows where both directly and indirectly leaked blocks are allocated.

---

<sup>15</sup>It’s not fair to say that the runtime developers are lazy, though. There are some technical difficulties with freeing this memory, and since it is in use up until your program exits anyway, there is little benefit to going to the effort of freeing it since the operating system deallocates it once your program exits anyway.

## Questions

Name: \_\_\_\_\_

1. In your own words, what is a use-after-free error?
2. What does `--track-origins=yes` do when used with Valgrind?
3. What bug does Address Sanitizer catch that Valgrind does not?



## Quick Reference

Using Valgrind: `valgrind [valgrind flags] program-to-run`

- `--track-origins=yes`: Show where undefined variables are declared.
- `--leak-check=full`: Show where directly leaked blocks are allocated.

Using Address Sanitizer:

- Compile your code with the `-g -fsanitize=address -fno-omit-frame-pointer` flags.
- Set the following environment variables:<sup>16</sup>

```
export ASAN_SYMBOLIZER_PATH=`which llvm-symbolizer`  
export ASAN_OPTIONS=symbolize=1
```
- Run your code as you normally would.

## Further Reading

- [Valgrind Memcheck Manual](#)
- [Address Sanitizer Wiki](#)
- [GCC `-fsanitize=address` documentation](#)
- [Paper on Address Sanitizer](#)

---

<sup>16</sup>This requires `llvm` to be installed. Also, depending on the system you are running, you may need to append a version number, e.g., `export ASAN_SYMBOLIZER_PATH=`which llvm-symbolizer-3.9``



## Chapter 10

# Profiling

### Motivation

“IT’S NOT FAIR!”

You sit at your desk, bewildered. If you’d known you’d have to deal with actors, you would have never taken this video editing job in 1960s Hollywood.

“I’m tellin’ ya. *She* gets TWICE as much screen time as ME!”

Mr. Grampton St. Rumpsterfrabble. You know he’s famous and you know people like him, but you never understood why. Especially now.

“I really don’t think that’s true,” you calmly reply as you push up your rose-colored glasses<sup>1</sup>. “I’ve already spliced the film together, and I don’t think Ms. Stembleburgiss is seen anymore than you are.”

“Yeah? Well prove it, wise guy.”

You begrudgingly roll your chair over to your latest film invention: the Timeinator. Carefully, you load the master reel for “The Duchess Approves II: The Duchess Doesn’t Approve” into the input slot. The machine whirs and clicks – clacks and bonks. Finally the 8-segment display at the bottom shows its output

```
idiots:      30m
film:        1h30m
projector:   5s
```

“Look,” you motion to the display. “This machine tells me how much of the movie features... actors..., how long the movie is, and how much time the projector spends warming up and stuff. If you’re going to be angry, be angry about the fact that you and Ms. Stembleburgiss are only on screen for 30 minutes. As I

---

<sup>1</sup>They’re literally classes with rose-colored lenses. It’s not a metaphor or any such nonsense.

recall, the film is mostly footage of Puddles the Amazing Schnauzer balancing on a beach ball.”

“How *dare* you” St. Rumpsterfrabble whispers with rage. “Puddles earned every frame she is in.”

A tear runs down his cheek.

“I cannot compete with that level of talent. I *can* compete with HER, though.”

“Ms. Stumbleburgiss”

“Yes, Ms. Stumbleburger... Ms. Stepplemurder... Ms... YOU KNOW WHO I MEAN. I want you to go figure out **exactly** how much screen time *she* has and how much I have.”

Realizing he’s not going to leave until you do, you come up with a shortcut. You load the master reel into your preview machine and check every 50th frame to see who’s in the shot. The film was shot at 24 frames per second, so you’ll check every 2 seconds of film. It’s not exact, but it’s good enough. Besides, the film is mostly Puddles anyway.

“You both share the same amount of screen time. Down to a granularity of 2-seconds,” you blandly state.

“Two seconds?! That’s enough to make or break a career!” St. Rumpsterfrabble shouts. “I want you to look at *every single frame* to see WHO has the most screen time!”

You slump onto your desk and consider going back for that degree in Computer Science. Sure, Hollywood is grand, but at least you wouldn’t be dealing with this sort of frame-by-frame tedium every day.

Well, you would, actually.

Everyone wants their programs to be fast, but it’s not always obvious how to make them fast. It is often necessary to dive *deep* to figure out where your program is spending most of its time running. Just like watching your film frame-by-frame, sometimes you have to watch your program run line-by-line to figure out which parts are fast and which are slow.

Fortunately, you don’t have to sit with a debugger counting line after line. There are tools called **profilers** that automate this process for you. You can think of them like souped up stop watches. They can give you detailed breakdowns of how your program runs, so that you can identify areas for improvement.

## Takeaways

- Realize that films should feature more dogs balancing on beach balls
- Gain a basic understanding of what a profiler is and how different profilers work

- Understand how to use and interpret results from `time`, `gprof`, `callgrind`, and `massif`

## Walkthrough

### Timing Programs with `time`

You can think of `time` like a fancy stopwatch. It tells you:

**Real time** This is how long your program takes to run start to finish. We often call this “wall time” because you could measure the time elapsed using a clock on the wall.

**User time** This is how long your program spends running on your computer’s CPU. Your computer actually runs a lot of programs at once. There’s a program that manages your monitor, one that handles keyboard input, one that manages your files, etc. The trouble is that if you have more programs than CPUs, they have to take turns. Your operating system will divide CPU time between the different programs.

**User time** tells us how much time *your program* spends on the CPU. If it has to share the CPU with other programs (which it probably will), the user time will likely be much less than the real time.

**System time** This is how long your program spends waiting to hear back from your operating system (OS). Whenever you do any sort of I/O<sup>2</sup> operations, your program makes a system call that asks your OS to do those things for you. If your OS is busy doing other stuff, your program will have to wait to hear back from the OS before it can continue running. **System time** reflects this time spent waiting to hear back from the OS.

`time` is very easy to use. In order to use it, just throw `time` in front of the program you want to run. It doesn’t care if the program has command line arguments.

```
# If we want to see how long it takes to list the files in /tmp
$ time ls /tmp
time ls -a /tmp
.    ..
```

```
real    0m0.004s
user    0m0.001s
sys     0m0.002s
```

```
# If we wanted to time a hello world program...
```

---

<sup>2</sup>Input/Output

```
$ g++ -o hello hello.cpp
$ time ./hello
```

## Profiling with gprof

Although `time` is handy for determining run times, it doesn't give you any indication about which parts of your programs are slow and which parts are fast. However, `gprof` can do that for you. `gprof` samples your program as it runs to tell you how much time you're spending in function calls. We'll discuss two different profiles that are included in `gprof`'s output: flat profiles and call graphs.

First, let's look at an example program. In this program, we have a few different functions. Each one has a for-loop that just wastes time for the sake of example.

```
#include <iostream>
using namespace std;

void new_func1()
{
    cout << "Inside new_func1" << endl;
    for (int i = 0; i < 2000000000; i++)
    {
    }
    return;
}

void func1()
{
    cout << "Inside func1" << endl;
    for (int i = 0; i < 2000000000; i++)
    {
    }
    new_func1();
    return;
}

void func2()
{
    cout << "Inside func2" << endl;
    for (int i = 0; i < 2000000000; i++)
    {
    }
    return;
}
```

```
int main(void)
{
    cout << "Inside main" << endl;
    for (int i = 0; i < 2000000000; i++)
    {
    }
    func1();
    func2();
    return 0;
}
```

In order to use `gprof` we have to pass an additional flag to `g++`. The `-pg` flag tells `g++` to record profile information whenever our compiled program runs. The generated file, called `gmon.out`, contains the information that is interpreted by `gprof`.

Let's compile the program above. We'll assume it's called `main.cpp`.

```
$ ls
main.cpp
$ g++ -pg -o main main.cpp
$ ls
main    main.cpp
```

In order to generate `gmon.out`, we need to run `main`.

```
$ ls
main    main.cpp
$ ./main
Inside main
Inside func1
Inside new_func1
Inside func2
$ ls
gmon.out    main    main.cpp
```

Now that we have `gmon.out`, we can ask `gprof` to show us the profile.

```
$ gprof main
```

## The Flat Profile

Whenever you run `gprof`, you'll see a flat profile at the top followed by a call graph. In its output, `gprof` includes detailed documentation to help you better understand what you see. For the flat profile, it describes the sampling procedure and explains the meaning of each column. The same documentation can be found in the Further Reading section below.

For our above program, we see the following **flat profile**:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
26.11	5.61	5.61				main
24.99	10.98	5.37	1	5.37	5.37	new_func1()
24.94	16.34	5.36	1	5.36	10.73	func1()
24.89	21.69	5.35	1	5.35	5.35	func2()
0.00	21.69	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9new_func1v
0.00	21.69	0.00	1	0.00	0.00	__static_initialization_and_dest

When your program runs, it makes a note in `gmon.out`...

- every time a function is called. This ensures that our function call counts are exact.
- Every 0.01 seconds. This gives us a rough idea as to how much time we're spending in each function. These are referred to as "samples".

You can see in the profile that `main`, `new_func1`, `func1`, and `func2` were each called one time, and we spent roughly 25% of our time in each one. That makes sense to see, given that each of those functions wastes time using the same kind of for-loop (one with two billion iterations).

The functions are sorted by the amount of time spent running in each. That is, the function with the most samples is the one we spent the most time in.

### The Call Graph

In addition to the Flat Profile, `gprof` shows you a Call Graph. The Call Graph goes one step further than the Flat Profile by showing you how much time you spent running a function and its children. Again, `gprof` will display a bunch of documentation for interpreting the Call Graph, and that same information is linked in the Further Reading section below.

For our program above, we see the following Call Graph:

index	% time	self	children	called	name
<spontaneous>					
[1]	100.0	5.57	16.17		main [1]
		5.41	5.35	1/1	func1() [2]
		5.41	0.00	1/1	func2() [3]
-----					
		5.41	5.35	1/1	main [1]
[2]	49.5	5.41	5.35	1	func1() [2]
		5.35	0.00	1/1	new_func1() [4]
-----					
		5.41	0.00	1/1	main [1]
[3]	24.9	5.41	0.00	1	func2() [3]
-----					
		5.35	0.00	1/1	func1() [2]



[4]	24.6	5.35	0.00	1	new_func1() [4]
-----					
		0.00	0.00	1/1	__libc_csu_init [18]
[11]	0.0	0.00	0.00	1	_GLOBAL__sub_I_Z9new_func1v [11]
		0.00	0.00	1/1	__static_initialization_and_destruction_0(int, i
-----					
		0.00	0.00	1/1	_GLOBAL__sub_I_Z9new_func1v [11]
[12]	0.0	0.00	0.00	1	__static_initialization_and_destruction_0(int, int)
-----					

Let's start by making some observations about index 1. The call graph shows...

- we spent 100% of our time running `main`. That makes sense, given that `main` is the entry point to our program.
- we spent 5.57 seconds in `main`, 5.41 seconds in `func1`, and 5.41 seconds in `func2`. These values agree, roughly, with our Flat Profile.
- `func1` spent 5.35 seconds running its children functions. This makes sense, because `func1` calls `new_func1`, which takes 5.35 seconds to run.
- `main` spent 16.17 seconds on its children. This makes sense because `func1` + `new_func1` + `func2` is roughly 16 seconds.

As you can see, the Call Graph essentially breaks down how much time `main` spends running itself, as well as how much time it spends calling other functions. By accounting for these separately, you can better see where your time is going.

In the following entries, you can see more detail about where time is spent for functions other than `main`. Index 2, for example, shows you the amount of time spent when `main` calls `func1`. The breakdown shows the amount of time spent running code in `func1`, as well as the amount of time running its only child: `new_func1`.

Although this example does not demonstrate it, `gprof` has the ability to show you details for more complicated call graphs. For example, if both `func1` and `func2` called `new_func1`, `gprof` would show you how much time you're spending in `new_func1` when it's called by `func1` and when it's called by `func2`.

If you have a situation where calling context changes its running time, you may find this extra information useful. Perhaps `new_func1` is very fast when `func1` calls it, but it's very slow when `func2` calls it. You could use this information as a clue to figure out why `new_func1` is sometimes slow.

## Profiling with `callgrind`

`gprof`'s approach to take samples every 0.01 seconds works for many people when they're trying to identify slow spots in their code. However, it is not perfect.

Remember our film predicament? `gprof` is like looking at every 50th frame.

It gives you a *good* idea of how much movie time we spend looking at Mr. St. Rumpferfrabble, but it's not perfect. If we take the time to go frame-by-frame, that's the most detailed we can possibly get.

`callgrind` is our frame-by-frame approach. Instead of going frame-by-frame, we're going instruction-by-instruction. `callgrind` is one of several tools built using the Valgrind framework. Tools built using Valgrind can see *every single instruction* that is run by a program. This level of detail can be very powerful, but it can also be pretty slow.

Let's consider our program from the `gprof` section, but let's use ten thousand iterations for each for-loop instead of two billion. This time when we compile it, we need to pass the `-g` flag instead of `-pg`. Then we'll run our program through `callgrind` as shown.

```
$ g++ -g -o main main.cpp
$ valgrind --tool=callgrind ./main
```

Like `gprof`, `callgrind` will generate a data file for us. However, the output file does not always have the same name. Each `callgrind.out.NNNN` file is named according to its process ID. When we use `callgrind_annotate` to view the profile information, we need to make sure we pass the right `callgrind.out.NNNN`.

```
# Be careful! The process ID (31147 in this case) will change!
$ callgrind_annotate --auto=yes callgrind.out.31147
```

In its output, `callgrind_annotate` will show you how many CPU instructions were run for each line of code. Let's have a look at the annotated source for `main`.

```
.      int main(void)
3      {
16          cout << "Inside main" << endl;
8,991 => ???:std::ostream::operator<<(std::ostream& (*)(std::ost...
2,425 => /build/eglibc-SvCtMH/eglibc-2.19/elf/./sysdeps/x86_64/...
4,735 => ???:std::basic_ostream<char, std::char_traits<char> >& ...
.
30,004     for (int i = 0; i < 10000; i++) {
.         }
.
1         func1();
61,383 => main.cpp:func1() (1x)
1         func2();
30,672 => main.cpp:func2() (1x)
.
1         return 0;
20     }
```

Let's break this down a bit:

- We spend 16 CPU instructions printing "**Inside main**" to the console. In reality, though, those 16 instructions simply make calls to other functions thanks to `cout` and the `<<` operator. We actually spend  $16 + 8,991 + 2,425 + 4,735$  instructions printing to the console if you count the functions that we called.
- We spent a total of 30,004 CPU instructions instantiating an `int` called `i`, checking that it's less than 10,000, and incrementing it. All 30,004 of those instructions were used to perform the loop initialization, check, and post-loop actions. Zero instructions were used in the body of the for-loop.
- It took 1 CPU instruction to call `func1` one time (1x), but running `func1` used 61,383 instructions.
- It took 1 CPU instruction to call `func2` one time (1x), but running `func2` used 30,672 instructions.

As you can see, `callgrind` gives us different details that `gprof` cannot. Based on where you run the most instructions, you can identify parts of your code that may need to be rewritten.

## Questions

Name: \_\_\_\_\_

1. Use `time` to time the execution of a simple Hello World program. What were the values for `real`, `user`, and `sys`? Do those values make sense?

## Quick Reference

## Further Reading

### **gprof**

- [Interpreting Flat Profiles](#)
- [Interpreting Call Graphs](#)



## 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



## Appendix A

# General PuTTY usage

In this course, we'll be writing, compiling, and running programs on the Linux operating system. Since our campus' Computer Learning Centers are mostly equipped with computers running Windows<sup>1</sup>, we need a way to connect to and use computers running Linux.

To do this, we'll be making extensive use of PuTTY.

### What PuTTY is

PuTTY is an **secure shell** (SSH) client for Windows. This means that we can use PuTTY to connect to a remote Linux computer that is running an SSH server. Once connected, we can run programs on that remote computer.

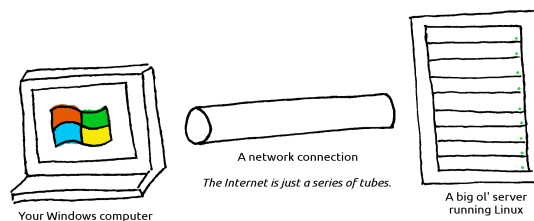


Figure A.1: PuTTY (running on your Windows computer) connects to a Linux computer over a network.

After you use PuTTY to log in to a remote Linux computer, you can type commands into a **bash** shell. It's important to understand that **the shell is**

---

<sup>1</sup>Windows is also an operating system.

**actually running on the Linux computer.** All programs you run in the shell actually run on that remote computer.

Those programs are *not* running on your Windows computer.

They are *not* running in PuTTY.

PuTTY is simply communicating with the Linux computer over the network to show you the shell. PuTTY is just a kind of window<sup>2</sup> into the remote Linux computer.

## What PuTTY is not

To reiterate: PuTTY **is not** Linux.

Instead, PuTTY allows us to *connect* to a computer that is running Linux. Whenever you type commands in the PuTTY shell, you're actually typing them in `bash`, which is running on the Linux server. Again, PuTTY is just a kind of window into the remote Linux computer.

## How to Use PuTTY

### Basics

After you log into a CLC Windows computer, simply locate PuTTY in the list of programs and start it. It should look like Figure A.2.

---

<sup>2</sup>Pun intended.



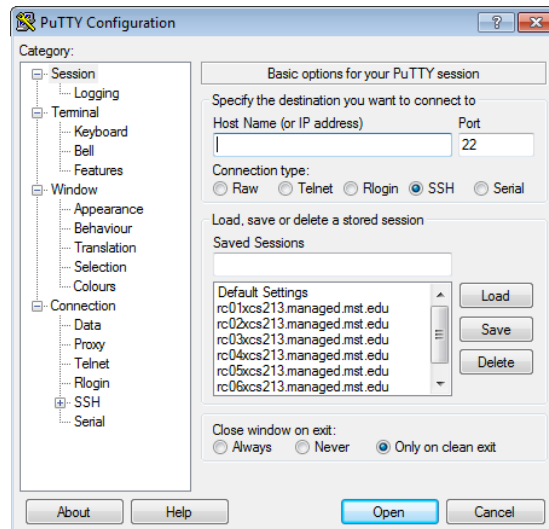


Figure A.2: The initial PuTTY screen

Once PuTTY is open, simply pick a connection configuration from the list. Click the configuration you'd like to load and press the **Load** button. Once you do that, you should see the corresponding hostname in the text field as shown in Figure A.3. You can also create your own configuration or modify the existing configurations and save them using the **Save** button.

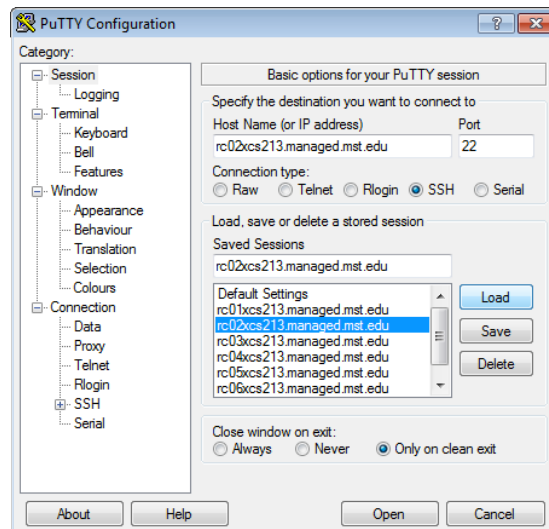


Figure A.3: Choose a configuration from the list and press the Load button to load it up.

Once your configuration is loaded and all the settings look right, press the **Open** button to start the connection. PuTTY will start communicating with the remote computer specified by the hostname. If it's unable to connect, PuTTY will complain.

If you've never connected to a particular Linux hostname before, PuTTY will warn you with a message similar to Figure A.4. It will show you its SSH fingerprint<sup>3</sup> and ask that you confirm the connection.

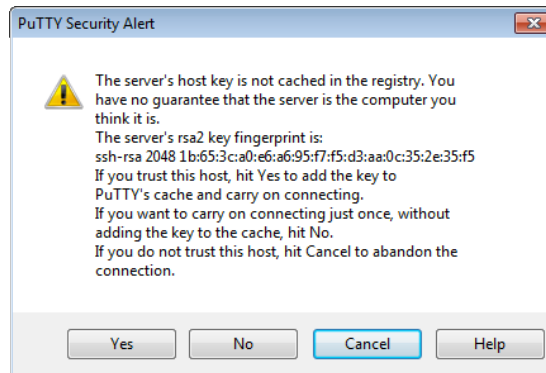


Figure A.4: If you've never connected to a particular Linux machine, PuTTY will ask if you're sure you want to connect.

If you confirm the connection, PuTTY just needs to know your login credentials. It'll start by asking for your username (Figure A.5) followed by your password (Figure A.6).

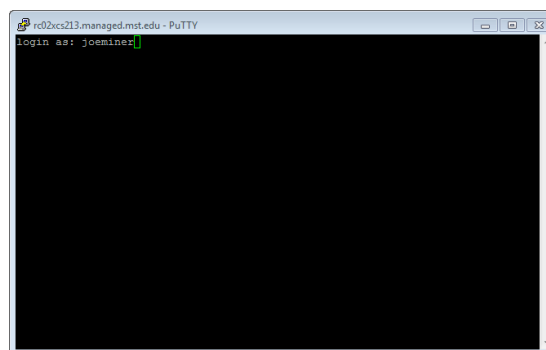


Figure A.5: If necessary, tell PuTTY your username.

---

<sup>3</sup>Uh... Google it.

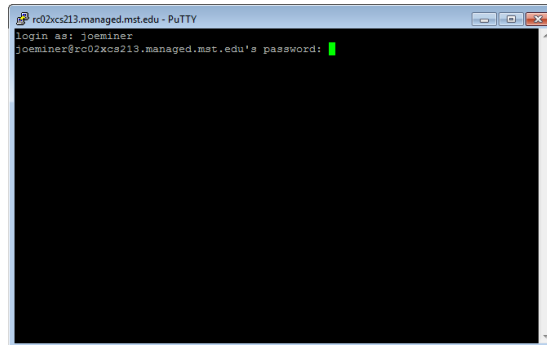


Figure A.6: Use your Single Sign-On password to sign in.

Assuming you entered your credentials correctly, PuTTY will present you with a shell as in Figure A.7. Take note of the number of users on your host. If there are a lot of users connected to the computer you're using, it'll be slower. You might consider trying a different hostname if you find the one you're using is sluggish.

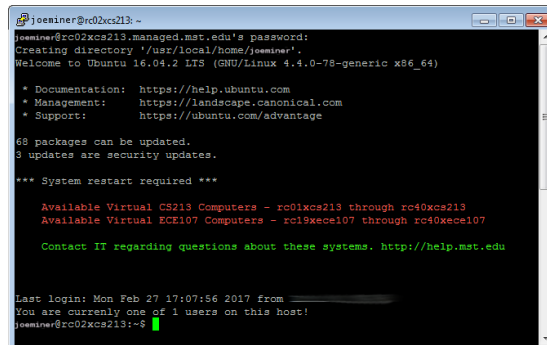


Figure A.7: Once you've connected, you'll be presented with a welcome message and a bash prompt.

## Other Tips

Here are a handful of tips:

- You have to be on the campus network to connect to the campus Linux hosts.
  - You can use any computer that is connected to the campus Ethernet or wireless networks. That includes CLC computers, your own desktop, your friend's laptop, etc.

- You can setup a VPN connection to connect to the campus network from off campus. Refer to IT's help pages to set that up.
- IT's has a list of the Linux hostnames on their website: <http://it.mst.edu/services/linux/hostnames/>. Since the PuTTY default only lists the first 16 or so, most people use those. Try using the higher-numbered machines. They often have far fewer users on them, and thus, they're notably faster. When you connect, the Linux host welcome message will tell you how many users are connected.

## Useful Settings

## Clipboard Tips

## Appendix B

# X-forwarding

True, we use the shell a lot in this course, but every now and then we have to run programs that have GUIs. When we're running GUI programs on *Windows* (such as Notepad or Microsoft Word), it's easy. Just find the program in your Start menu, click it, and off you go. If you need to start a GUI program on a remote Linux computer, though, things are more... complex.

## Remote GUIs

Linux uses the X Window System to display GUIs and interact with you, the user. Basically, GUI programs work like this:

**GUI Program:** Hey, X Server. I need you to draw a window on the screen for my user.

**X Server:** What's in it for me?

**X Server:** I'm just kidding. What's it look like?

**GUI Program:** Well, it's got a text box here, and some shapes over there.

**X Server:** That sounds great. I'll draw that on this **display** over here.

X Server then sends a bunch of data to a **display**. If that Linux computer has a monitor connected, the data would be sent to that monitor.

As it turns out, you can ask X Server to send that display data over a network. If you ask nicely, PuTTY can request that X Server send that display data to your Windows computer. Together with a program called Xming, we can see the

windows (and such) that would have been displayed if we connected a monitor directly to the Linux computer.

But it's all remote.

## Configuring X-forwarding

As previously mentioned, we're still going to use PuTTY to connect to the remote host. However, PuTTY doesn't know how to draw on the screen. All it can do is the shell stuff.

To help PuTTY out, we need to start its partner in crime: Xming. Find **Xming** within your start menu (as in Figure B.1) and click it. Don't start XLaunch or anything else. We just want Xming.

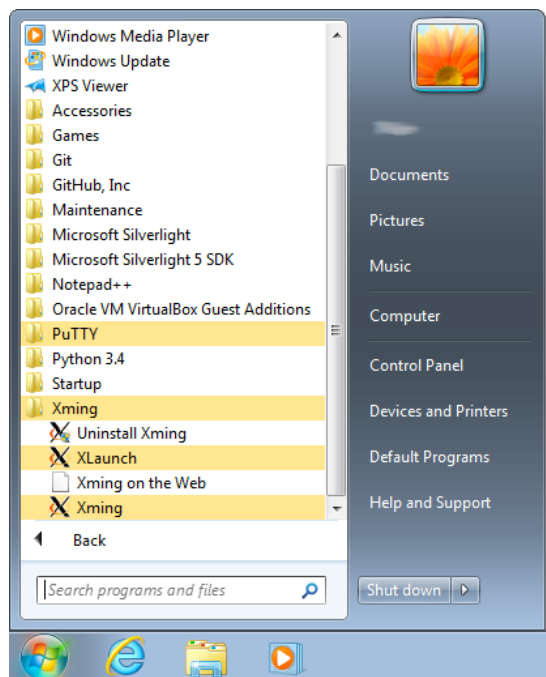


Figure B.1: We want to start Xming. **Not** XLaunch or anything else. Xming.

You only need to do this one time after you log in. Xming will run in the background until you stop it or log off. You can check to see if Xming is running by looking in your task bar as shown in Figure ???. If you see the logo down there, there's no need to start Xming again.

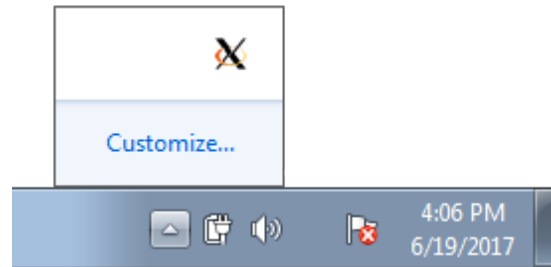


Figure B.2: You can check your task bar to see if Xming is running. ??

Now that Xming is running, we need to tell PuTTY to send all that display data to Xming. After you load a Putty configuration but before you connect, you need to **make sure** that X11 Forwarding is enabled.

So:

1. Open PuTTY
2. Click a hostname in the list
3. Click the **Load** button
4. Find the X11 Forwarding configuration and make sure it is enabled as shown in Figure B.3.

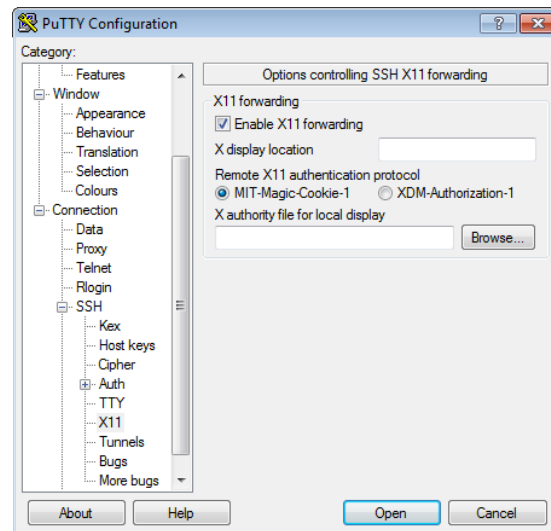


Figure B.3: Make **sure** that X11 Forwarding is enabled!

If Xming is running and X11 Forwarding is enabled, you can start your PuTTY connection by pressing the **Open** button. PuTTY will open a shell like normal.

Nothing actually looks different until you try to start a program.

Try running `gedit` (a GUI text editor for Linux), `firefox`, or `chromium-browser`. These are all GUI programs and should start up. Figure B.4

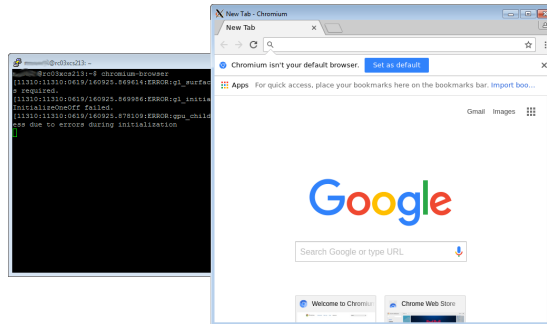


Figure B.4: The Chromium Browser forwarded to our Windows computer. Remember that Chromium is running *on the Linux computer*. PuTTY is forwarding the display data, and Xming is drawing the browser window for us.

It's important to keep in mind that while your GUI program is running, your shell will be busy. It's just like any other program you start in your shell. Until you close the GUI program, your shell will be unavailable. You may find it useful to have a couple of PuTTY windows open, so that you can multitask.



Appendix C

Markdown



## Appendix D

# Parsing command-line arguments in C++



## Appendix E

# Submitting homework with Git