

Tools For Computer Scientists  
Or, How To Use A Computer Real Good

Nathan Jarus and Michael Wisely

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

The source code for this book is available at <https://github.com/LearnYouSomeComputer/Tools-For-Computer-Scientists>. This book was built from commit [95ca56f](#).

We welcome questions, corrections, and improvements!

© 2016–2018 Nathan Jarus and Michael Wisely

This work is licensed under the Creative Commons Attribution–ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Zeroth edition: August 2017

# Contents

Introduction	i
1 Exploring Text Editors	1
2 Bash Basics	31
3 Git Basics	45
4 Bash Scripting	65
5 Building with Make	81
6 Debugging with GDB	95
7 Finding Memory Safety Bugs	109
8 Unit Testing	125
9 Integrated Development Environments	141
10 Profiling	153
11 Regular Expressions	171
12 Graphical User Interfaces with Qt	183
13 Typesetting with L <sup>A</sup> T <sub>E</sub> X	199
14 Using C++11 and the Standard Template Library	211
A General PuTTY usage	227
B X-forwarding	233
C Markdown	237

D Parsing command-line arguments in C++	239
E Submitting homework with Git	243

# Introduction

The field of Computer Science is an odd amalgamation of math, logic, statistics, philosophy, electrical engineering, psychology, and systems engineering. Some of your classes will be hands-on and easily applicable; others will focus on ideas and theory, leaving the application in your hands.

Throughout your career as a computer scientist, computer engineer, or software engineer, you will write a lot of software. This book focuses on software tools that will be useful in classes, internships, and personal projects. Much as chemistry students are taught how to use lab equipment and mechanical engineering students learn the basics of machining, this book teaches the basics of tools for writing and debugging software.

Each chapter of this book could be a book in itself — our goal here is to give you an idea of what these tools are useful for so that you'll know where to look when you need them in the future. We've included links to more information on the book's topics at the end of each chapter, as well as a quick reference on how to use each tool.

To get the most out of this book, you must do more than just read each chapter. As you read, try out the examples in the chapter. Once you get an example working, take it a step further — try something that seems like it might work, or combine it with something you learned previously! Your learning does not end when you finish this book either. Keep using the tools we present; practice will improve both how quickly you can accomplish tasks and your understanding of how the tools, and by extension computers and software, work.

A note on the copyright license of this book: copyright laws are typically used to restrict the rights of others to copy, modify, and distribute creative works. This book is distributed under what's known as a 'copyleft' license — rather than restricting your rights, it ensures that you retain these rights. The [Creative Commons Attribution-Share Alike \(CC BY-SA\)](#) license states that you have the right to copy, modify, and distribute your modifications to this book as you please so long as you follow two rules:

1. You have to credit the authors of the work and indicate if you have made any changes.

2. You have to distribute your changes under the same license, giving others the same rights to your modifications as you (and they) have to the original work.

We hope that learning the tools in this book make your life easier and that you have a little fun along the way. Happy hacking!

# 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 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.

## 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.

## 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.

---

<sup>4</sup>Thanks to cloud computing, it may not even be on the globe!

<sup>5</sup>Website: <https://notepad-plus-plus.org/>

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



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

### Indenting and commenting code

- **Tab** **↵**: Indent current line/block
- **Shift** + **Tab** **↵**: Outdent current line/block
- **Ctrl** + **q**: Single-line comment/uncomment current line/selection
- **Ctrl** + **Shift** + **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** + **Shift** + **↑** / **↓** / **←** / **→** 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 the column editor window.

### Multiple Cursors

Notepad++ supports multiple cursors, allowing you to edit text in multiple locations at once. To place multiple cursors, you'll need to enable it first. Navigate to **Settings** » **Preferences** » **Editing**, and toggle the box within the **Multi-Editing Settings** box. After that, just hold **Ctrl** and left-click everywhere you want a cursor.

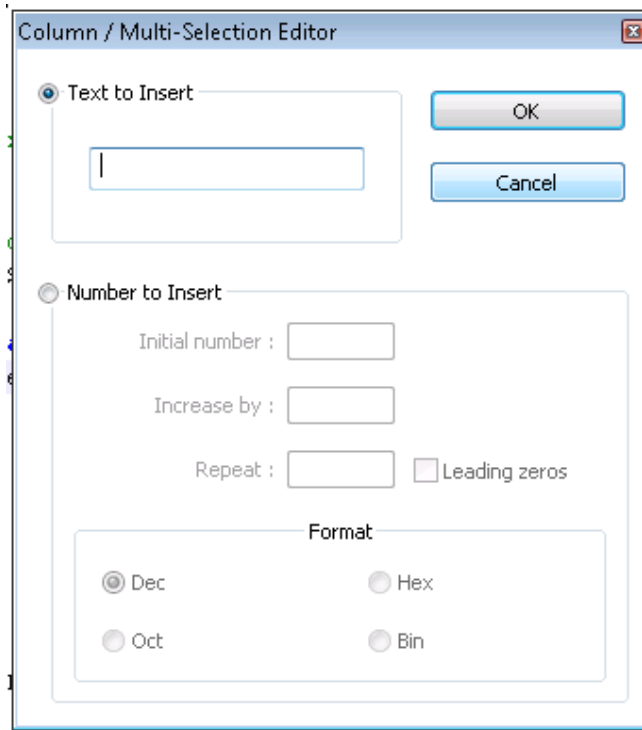


Figure 1.1: Column Editor

Then, you can type as normal and your edits will appear at each cursor location.

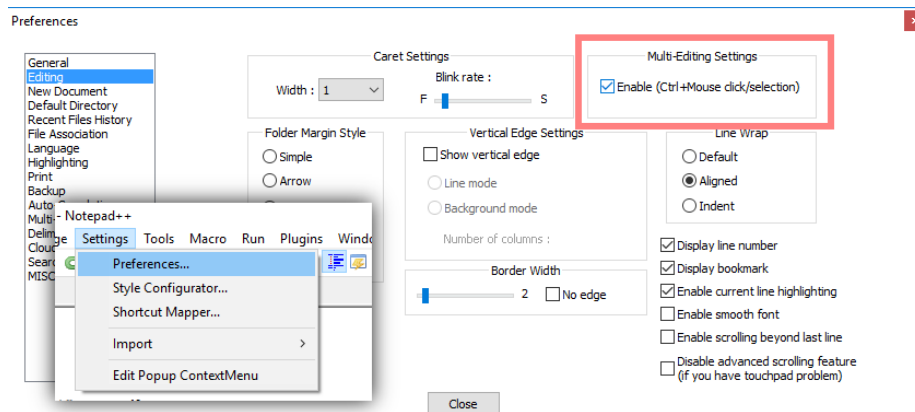


Figure 1.2: Enabling Multi-Cursor Editing

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:

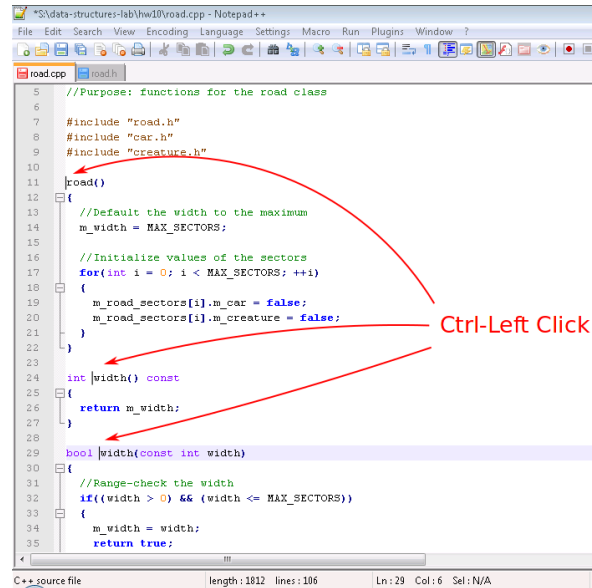


Figure 1.3: Placing multiple cursors with Ctrl + left-click

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

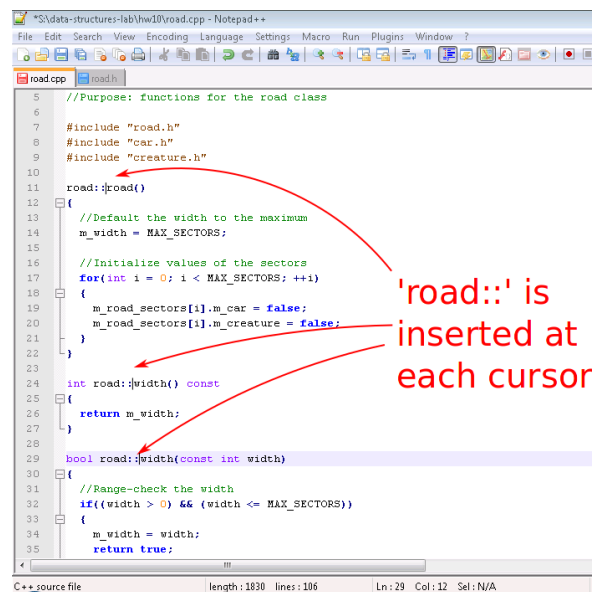


Figure 1.4: Typing `road::` inserts that text at each cursor location

## 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**

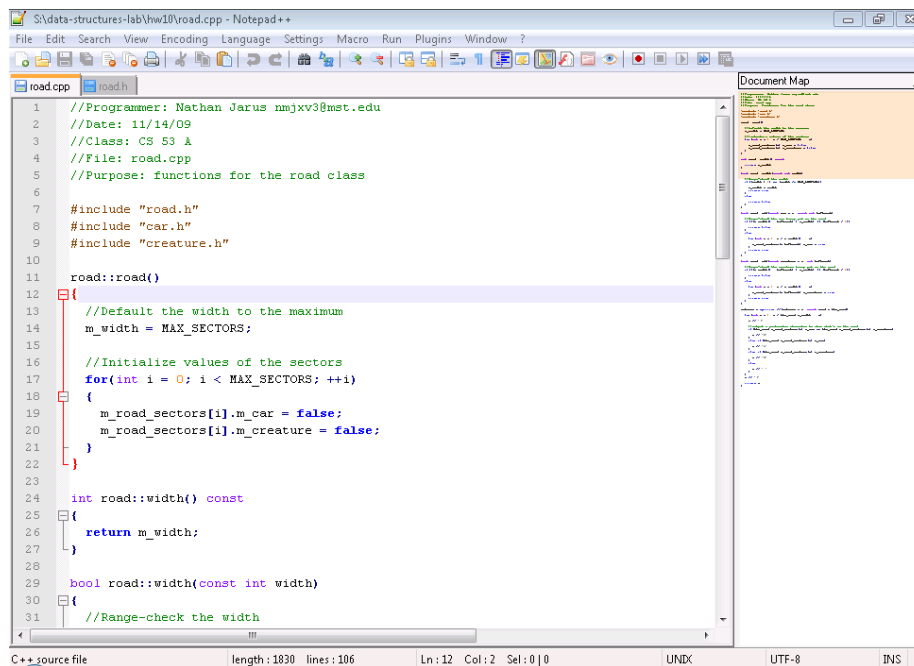


Figure 1.5: The document map

## 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...

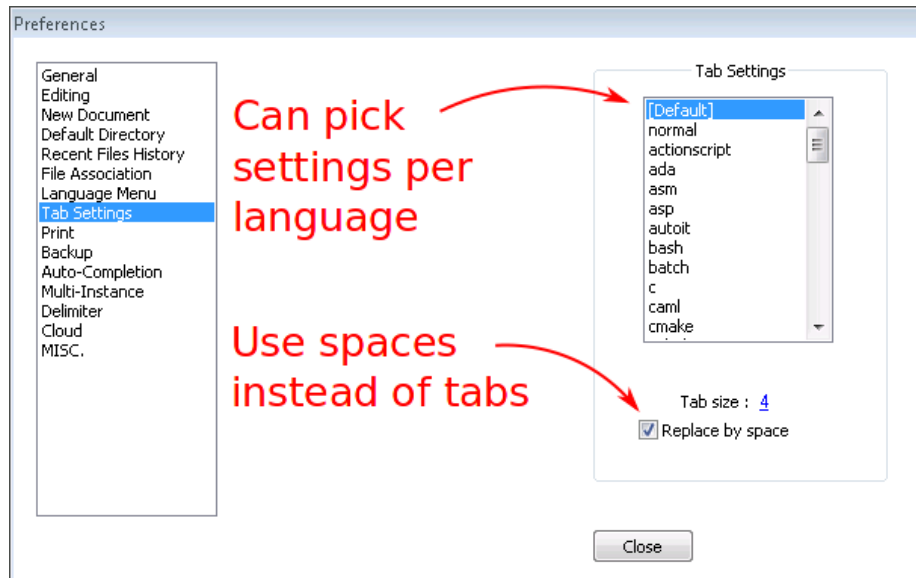


Figure 1.6: Configuring Notepad++ to use spaces rather than tabs

## 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™ 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 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

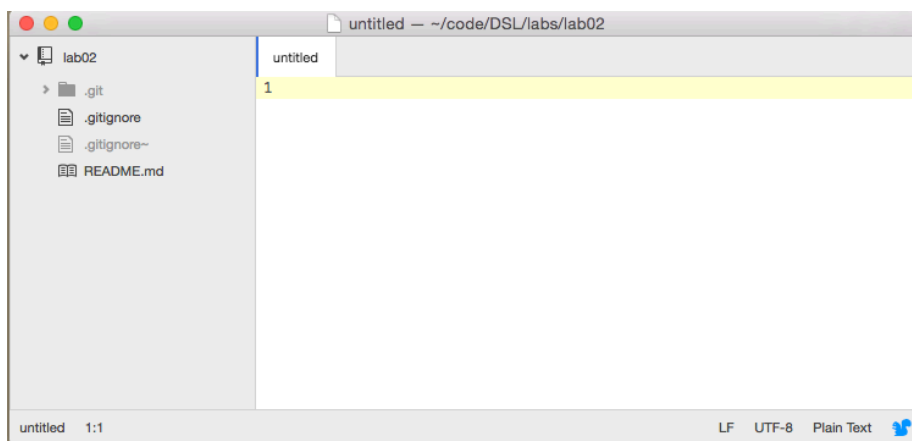


Figure 1.7: One Atom window with Tree View on the left and an empty pane on the right

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 of the window. 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?”.

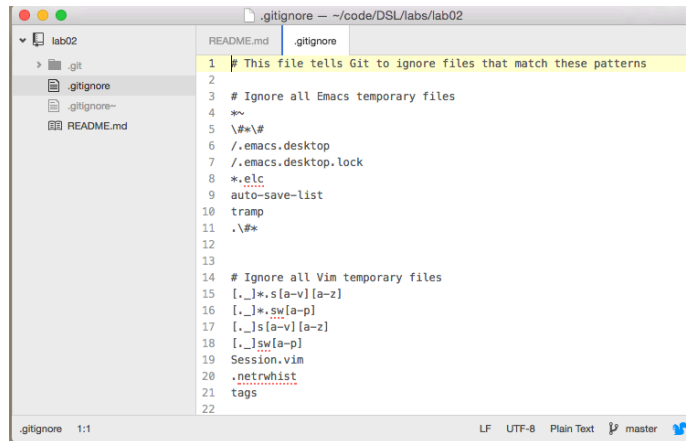


Figure 1.8: Atom with multiple tabs in one pane

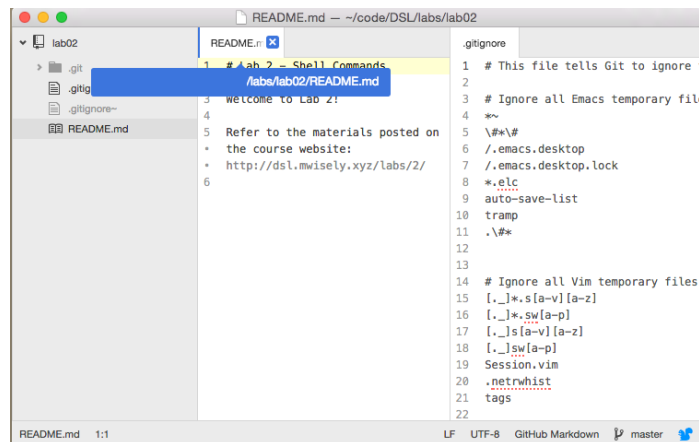


Figure 1.9: One window split into two panes

## 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

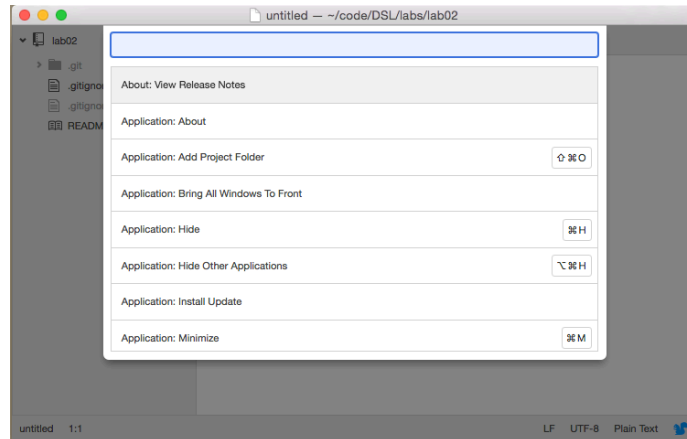


Figure 1.10: The Atom command palette (i.e., your best friend forever).

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` + `Shift` + `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.

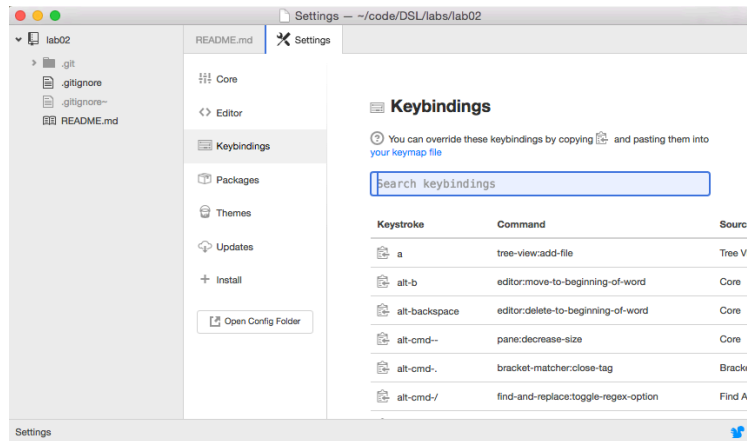


Figure 1.11: Atom’s settings open in a new tab. You can search through its keybindings here.

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!

## Nano

**nano** 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: **nano** was created as a Free as in Freedom<sup>13</sup> replacement for **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>)

You can open files in **nano** by passing them as command-line arguments:

```
$ nano my-cool-file.txt
```

<sup>13</sup>Or, less tongue-in-cheek: licensed so users can modify and distribute the software as they please — **nano** is licensed under the GNU GPL.

<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!

## How to Get Help

At the bottom of the **nano** screen are two lines that show commonly used commands. For more detailed documentation, **Ctrl**+**g** displays a summary of all the available keyboard shortcuts.

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**. **M-U** is read as “meta-U”; typically this is equivalent to pressing **Alt**+**u**.

## 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**+**→** or **Ctrl**+**Space** moves forward one word, and **Ctrl**+**←** or **Alt**+**Space** moves back one word.

Furthermore, **Alt**+**)** and **Alt**+**(** jump to the next or previous paragraph.

**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**+**home**, and to jump to the end, **Ctrl**+**end**.

If there’s a particular line number you want to jump to (for instance, if you’re fixing a compiler error), press **Ctrl**+**Shift**+**↑**+**-**, then type the line number to go to and press **Enter**. Alternatively, to see where you are in a file, press **Ctrl**+**c**.

## Undo and Redo

These are pretty easy. Undo is **Alt**+**u**; redo is **Alt**+**e**.

## Copy and Paste

You: “So, **nano** is kinda cool. But **Ctrl**+**c** and **Ctrl**+**v** both mean something other than ‘copy’ and ‘paste’. Can’t I just use the normal clipboard?”

Ghost of UNIX past: “It’s 1969 and what on earth is a clipboard?”

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

GOUP: “It’s 1969 and what is ‘select some text’?????”

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

GOUP: “The what now?”

You: “?????”

GOUP: “?????”

You: “?????????”

GOUP: “?????????!”

Seriously, though, the concept of a system-wide clipboard wasn’t invented until the 1980s, when GUIs first became available.<sup>17</sup> Before that, every terminal program had to invent its own copy/paste system! In **nano**, the clipboard is known as the ‘cutbuffer’; text can be cut or copied to the cutbuffer and then ‘un-cut’ (i.e., pasted).

You can cut one line with **Ctrl**+**k**, or cut from the cursor to the end of the file with **Alt**+**t**. **Alt**+**6** copies one line, rather than cutting it.

You can repeat a cut or copy command to add more to the last thing cut or copied. 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**+**Shift**+**↑**+**6** to start selecting text. Move the cursor around like normal; once you have completed selecting, press **Ctrl**+**k** to cut the selection or **Alt**+**6** to copy it.

Cut/copied text goes to the cutbuffer. To un-cut (paste) the contents of the cutbuffer, press **Ctrl**+**u**.

## Search and Replace

**Ctrl**+**w** lets you search for text. Type the text you want to search for and press **Enter**. **Alt**+**w** will jump to the next search result.

Alternatively, before pressing **enter**, you can toggle a number of search features:

- **Alt**+**c** to make the search case-sensitive
- **Alt**+**b** to search backwards, instead of forwards

---

<sup>17</sup>Command-line programs even predate computer screens! Before then, people used ‘teletypes’—electric typewriters that the computer could control. They were incredibly slow to output anything, and there was no way to erase what had already been printed, so having a ‘cursor’ didn’t really make much sense (how would you show it?). Some terminals didn’t even have arrow keys as a result!

- **Alt** + **r** to search using a regular expression (see the Regular Expressions chapter for more)
- **Ctrl** + **r** to interactively replace each match with a given replacement string (you can use backreferences in the replacement if you are searching by regular expression)

To search-and-replace only in part of a file, you can set a mark (with **Ctrl** + **Shift** + **6**) at one end of the region to replace in, navigate to the other, then press **Ctrl** + **w** to replace only in the marked area.

## Multiple Files

**nano** refers to open files as ‘buffers’. It displays only one buffer at a time, but it can have multiple buffers open simultaneously.

By default, **Ctrl** + **r** reads the contents of a file (or the output of a command) into the current buffer. This makes it easy to make copies of files or to save the output of commands. To do this, press **Ctrl** + **r** and enter a filename to read in. Once you’ve pressed **Ctrl** + **r**, you can press **Ctrl** + **t** to open a file browser, which is quite nifty! Alternatively, you can press **Ctrl** + **x** to execute a command and put its output in the current buffer.

To open a file in a new buffer, press **Ctrl** + **r**, then **Alt** + **f**. Then open a file or execute a command; the file or command output will appear in a new buffer.

You can switch between open buffers with **Alt** + **,** and **Alt** + **.**. The top left corner shows how many buffers are open and which buffer you’re currently displaying.

## Configuration

**nano** looks for a configuration file in `~/.nanorc` and gives second priority to settings in `/etc/nanorc`.

Some handy options:

- **set tabsize 4**: sets indentation width to 4 columns
- **set tabstospaces**: uses spaces for indentation, rather than tabs
- **set linenumbers**: show line numbers
- **set mouse**: enables the mouse!

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

## 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 **nano**, you can open specific files by listing them as arguments to the command.

```
$ emacs main.cpp
```

When it starts, Emacs will first check to see whether or not it has the ability to open any GUI windows for you.<sup>18</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>19</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

---

<sup>18</sup>For example, if you are using X forwarding, Emacs can detect the ability to open a GUI for you.

<sup>19</sup>Well, the **nw** in **-nw** stands for no window, but Emacs takes it much more dramatically.

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 **nano** 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
- **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 1.12: Emacs has commands for all kinds of things!<sup>20</sup>

## 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:

---

<sup>20</sup>© 2009 Will Willis. Used with permission.



- **C-a** Move to beginning of line
- **C-e** Move to end of line

Moving by sentence:

- **M-a** Move back to beginning of sentence
- **M-e** Move forward to end of sentence

Scrolling by page:

- **C-v** Move forward one screenful (Page Down)
- **M-v** Move backward one screenful (Page Up)

Some other useful commands:

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

## Undo and Redo

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

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

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

## Saving and Quitting

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

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

## Kill and Yank

In Emacs, your “copied” and “cut” information is stored in the “kill ring”.<sup>21</sup> The kill ring is...a ring that stores things you’ve killed (cut), so that you can yank (paste) them later.

Vocabulary:

- **Kill** - 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>21</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>22</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>22</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>23</sup> text editor for Linux, Windows, and macOS. It is popular for its power, configurability, and the composability of its commands.

For example, rather than having separate commands for deleting words, lines, paragraphs, and the like, Vim has a single delete command (**d**) that can be combined with motion commands to delete a word (**w**), line (**d**), paragraph (**{**), etc. In this sense, learning to use Vim is like learning a language: difficult at first, but once you become fluent it's easy to express complex tasks.

Vim offers a tutorial as a separate program: at a command prompt, run `vim-tutor`. 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`, which drops you into insert mode wherever your cursor is.

Some other ways to get into insert mode:

- `I`: Insert at beginning of line

---

<sup>23</sup>The graphical version is cleverly named `gvim`.

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

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

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

## Moving around in Normal mode

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

Some common motions:

- `j`/`k`/`h`/`l`: up/down/left/right<sup>24</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.

---

<sup>24</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.

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

## Undo and Redo

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

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

## Saving and Quitting

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

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

## Copy and Paste

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

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

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

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

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

## Indenting

You can indent code one level with `>` and outdent with `<`. Like `c`, these must be combined with a motion or repeated to apply to the current line. For

instance, `>%` indents everything up to the matching `'}'` (or bracket or parenthesis) one level.

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

## Multiple files

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

Working with buffers:

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

Working with windows:

- `:split` splits the current window in half horizontally
- `:vsplit` splits the current window vertically
- `Ctrl+W` 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
- Nano
- Vim
- Emacs

... figure out how to do the following:

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

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

For each editor, answer the following questions:

- What do you like about it?

- What do you dislike about it?

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

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

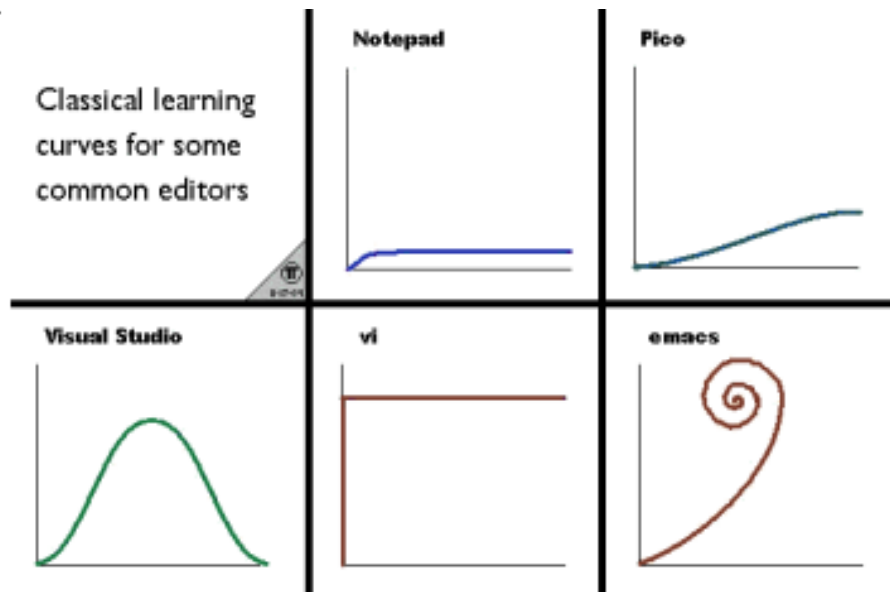


Figure 1.13: Learning editors is easy and fun!

## 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!

### Atom

- [The Atom Website](#)

- [Atom Manual](#)
- [Package List](#)
- [Atom Source Code](#)

## Nano

- [The Nano Website](#)
- [Nano FAQ](#), including some history of its development
- [Nano Manual](#)
- [Nano Source Code](#)

## Emacs

- [The Emacs Website](#)
- [Emacs Manual](#)
- [Emacs Wiki](#)
- [Emacs Packages](#)
- [Emacs Source Code](#)

## 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 an animal, typically 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 a Linux machine, 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

When you first log in, bash shows you a prompt, indicating that you can type a command to it. In this book, we'll show the prompt like so:

\$

Yours will probably have some additional information, such as your username, the name of the computer you're logged into, and the directory you're in.

To use bash, you 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**:

---

<sup>1</sup>The 'Bourne Again Shell', known for intense action sequences, intrigue, and being derived from the 'Bourne shell'.

```
$ pwd
/usr/local/home/nmjxv3
```

When you type `pwd` and press `Enter`, bash runs `pwd` for you. In turn, `pwd` outputs your **p**resent **w**orking **d**irectory (eh? eh?) and bash shows it to you.

## Arguments

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

```
$ 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.<sup>2</sup>

## Flags

In addition to command line arguments, we have **flags**.<sup>3</sup> A flag starts with one or more `-` and may be short (one character) or long (more than one character). Consider `g++` again:

```
$ 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.

---

<sup>2</sup>What if you want to pass an argument with a space in it? No dice, my friend. Adjust your wants accordingly.

...okay, fine, you can put single quotes (`'`) or double quotes (`"`) around your argument with spaces. We'll talk about this more in a later chapter.

<sup>3</sup>The distinction is mostly superficial; under the hood they look the same to the operating system. Don't worry about it too much.

- 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>4</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>5</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

---

<sup>4</sup>Thanks, old curmudgeons who can't be bothered to learn to type 'list'.

<sup>5</sup>We'll talk more about `*.cpp` later on in this chapter.



sizes (kilobytes, megabytes, big gulps, etc.), use `-h`.

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

```
$ 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.tar.gz
-rwxr-xr-x 1 nmjxv3 mst_users 586 Jan 15  2016 vector.h
-rwxr-xr-x 1 nmjxv3 mst_users 960 Jan 15  2016 vector.hpp
```

The columns in `ls`'s output are as follows:

1. File permissions — who can read, write, or execute your files
2. Number of hard links (don't worry about this for now)
3. The user who owns the file
4. The group that owns the file
5. The file size
6. The last time the file was modified
7. The file name.

Another `ls` option lets you show hidden files. In Linux, every file whose name begins with a `.` is a **hidden file**.<sup>6</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!

## A brief note on file permissions

Linux has separate permissions for the user who owns the file, users in the 'group' that owns the file, and everyone else. (Group permissions are useful in the case of shared documents — imagine making an **accounting** group that allows all accountants in a company to edit various spreadsheets on a shared drive.)

For each of these collections of users (the owning user, the users in the owning group, and other users), you can set whether those users can **read**, **write**, or **execute** the file. (Setting **execute** on a directory allows users to **cd** into it, which is why directories almost always are marked executable.)

To change file permissions (also known as the “file mode”), you use the **chmod** (change mode) command like so: `chmod <mode> <filename>`. Modes are

---

<sup>6</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.

written like so: `<collection><+/-><permission>`<sup>7</sup>

- `<collection>` is `u` for the owning user, `g` for users in the owning group, `o` for other users, or `a` for all users
- `+` adds a permission; `-` removes the permission
- `<permission>` is `r` for read, `w` for write, or `x` for execute.

So, for example, let's say you've downloaded some cool program (`not-a-virus`) from the internet and you want to run it.

```
$ ./not-a-virus
bash: ./not-a-virus: Permission denied
```

Darn! Guess you'll just have to install that ransomware on purpose...or, we could change the file permissions! We can see that right now, nobody can execute this cool program:

```
$ ls -l ./not-a-virus
-rw-r--r-- 1 nmjxv3 mst-users 31 Jan 15 00:07 ./not-a-virus
```

Let's change that! We'll make it so anyone can execute that file.

```
$ chmod a+x ./not-a-virus
$ ls -l ./not-a-virus
-rwxr-xr-x 1 nmjxv3 mst-users 31 Jan 15 00:07 ./not-a-virus
```

Nice! Now you can edit the file, and everyone can read it and execute it...

```
$ ./not-a-virus
HACKED
```

## 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

---

<sup>7</sup>There is another notation for these permissions that uses the octal (base 8, as opposed to base 10 or base 16) representation of the bitfield where the file permissions are stored. We won't go into it, but you can read about it in the `chmod` man page.

- `..` 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

Unlike Windows, on Linux, every file lives in `/` or a subdirectory of `/`. There are no drive letters! You can refer to files via their **absolute path**: a series of directories that starts with `/` and ends with the file you are referring to, such as `/home/flanders/bible/john.txt`. You can also refer to files via a *relative path*: a series of directories that does not start with `/` and is determined relative to where you are in the filesystem. For example, if you are in your home directory, you can edit a homework assignment directly by typing `emacs cs1585/lab02/cool-script.sh`.

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.

## Rearranging Files

If you want to move or rename 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, recycle can, ashtray, or other garbage receptacle,<sup>8</sup> 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 cs1585-TAs cool-people`.<sup>9</sup>

---

<sup>8</sup>Unless you want to count the whole computer as garbage. We won’t argue that point with you.

<sup>9</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).

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>10</sup> You can scroll up and down in `less` with the arrow keys, `PgUp` and `PgDn`, or `j` and `k` (like Vim). Pressing `Space` scrolls one page. If you want to explore more `less` features, `h` shows a help screen with a summary of various commands. 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`.

## The Manual

Many programs include help text; typically `--help` or `-h` displays 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.

Man pages look intimidating the first few times you look at them,<sup>11</sup> but don't worry. They are split into several sections. First, there's the **NAME** section that lists the name of the program. Following that is a **SYNOPSIS** section which very, very briefly summarizes the different arguments the program takes. Typically

---

<sup>10</sup>`less` is a successor to `more`, another paging utility, or as the authors would put it, `less` is `more`.

<sup>11</sup>Okay, they never really stop looking scary, but after a while they start to feel less like a horror movie jump scare and more like the monster you just know is there in the hall waiting to eat you if you were to get out of bed.

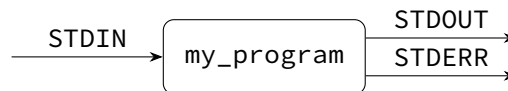
the section you want is the **DESCRIPTION** or **OPTIONS** section, which explains what each option does. Sometimes, interactive programs (such as **less** or **vim**) have a section on how to use the interactive features as well.

If you're just trying to remember the name of one option, it's best to use the search feature to look for relevant keywords. Otherwise, take your time and peruse the various features.

At the end of each man page, it may list related **FILES** or other commands and documentation that you should **SEE ALSO**. Following those sections is the **AUTHOR** section so you know whose name to curse when the program misbehaves, as well as a **BUGS** section which typically informs you that yes, this is software and yes, it has bugs in it.

## 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 terminal:

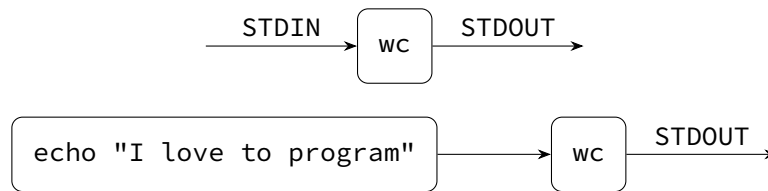
```
$ echo "hello"
hello
```

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 lines, words, and characters, 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:

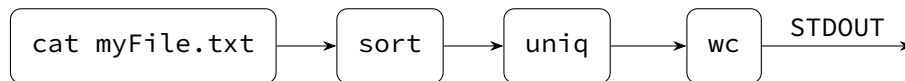


```
$ echo "I love to program" | wc
      1      4     18
```

Piping lets us compose all the utilities Linux comes with into more complex programs.<sup>12</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`.

So far, we've been redirecting output from one program to another, eventually putting some output on the screen. We can use `>` to redirect program output to files instead.

For example:

```
$ echo "hello world" > hello.txt
$ cat hello.txt
hello world
```

Now for a bit about `STDERR`. Bash numbers its output streams: `STDOUT` is `1` and `STDERR` is `2`. We can use these numbers with `>` to redirect specific streams to files. For instance, let's say you want to save your compiler errors to a file to look at later. If you were to run

```
$ g++ lots_o_errors.cpp > errors.txt
```

you'd be met with a screenful of errors and an empty text file. That's no good! By default, `>` only redirects `STDOUT`.

<sup>12</sup>And each program in a pipeline can run in parallel with the others, so you can even take advantage of multiple CPU cores!

To redirect STDERR instead, you'd do the following:

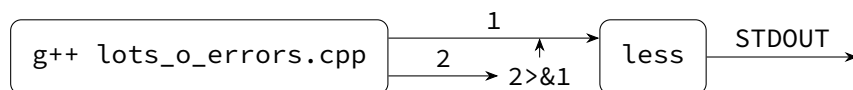
```
$ g++ lots_o_errors.cpp 2> errors.txt
```

Now you can page through your errors by running `less errors.txt`!

But what if you don't want to create that intermediate file? With STDOUT, you could just pipe that output right into `less`. If you want to pipe both STDERR and STDOUT into another program, you need to *redirect* STDERR into to STDOUT first. This is done like so: `2>&1`. The `&` tells Bash that you are referring to an output stream, rather than a file named `1`.

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

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



Output redirection is one of those things that seems a little odd when you first learn it, but once you get used to using it, you'll wonder how you lived without it. We'll see plenty of examples where output redirection comes in handy throughout the rest of this book.

## Questions

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

1. In your own words, 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. Let's say you're in some directory deep in your filesystem and you discover a file in your current directory named "cow.txt". What command would you run to move this file to a directory named "animals" that is located in your home directory?
4. Suppose you have a file containing a bunch of scores and names, 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 or path>`: Make a new directory

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

`cat [<filenames>]`: Output contents of files or STDIN

`less [<filename>]`: Interactively scroll through long files or STDIN

`head [<filename>]`: Display lines from beginning of a file or STDIN

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

`tail [<filename>]`: Display lines from the end of a file or STDIN

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

`sort [<filename>]`: Sort lines of a file or STDIN

- `-u`: Remove duplicate lines from output
- `-n`: Perform numerical sort instead of string sort

`wc [<filenames>]`: Count characters, words, and lines in files or STDIN

`echo <string>`: Print string to STDOUT

- `-e`: Interpret backslash escapes (i.e., `\n` is printed as a newline)

`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](#)
- [Software Carpentry Shell Tutorial](#)

## 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.<sup>4</sup>

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

---

<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.

<sup>4</sup>If you read the first footnote, close them first.

add features to your code, change the way it works, and sometimes introduce bugs. 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.

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>

In addition to local repositories, Git also has the concept of **remote** repositories: repositories connected to your local repository that you can push snapshots to or pull snapshots from. Remote repositories allow you to collaborate with others on a project without everyone having to share the same computer.

This is the point at which I want to compare Git to Dropbox or Google Drive, but that would be a dangerous comparison. Realize<sup>6</sup> that Git will feel similar to these services in some ways, but there are many features that make them *very* different.

## Trying out GitLab

To backup<sup>7</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>8</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.

---

<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>Using your mind.

<sup>7</sup>And other things. Remotes are actually *extremely* useful.

<sup>8</sup>GitLab, GitHub, BitBucket, etc.

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`
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. Use `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.

For the sake of this example, create a very simple Hello World C++ program and name it `hello.cpp`. 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. The `git add` command does two things: it tells Git to track that file, if Git isn't already tracking it, and it stages the changes in that file for the next commit.

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.

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! You only need to do this the first time you use Git on a machine. Git stores your configuration in a file (`~/.gitconfig`).

```
$ git config --global user.name "<your_name>"
$ git config --global user.email "<your_email>"
```

Let’s also tell Git which text editor you prefer to use. You need to choose a console editor such as `jpico`, `nano`, `emacs`, or `vim`. ~ `$ git config --global core.editor ~`

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>9</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
```

---

<sup>9</sup>Such as “asdf”, “stuff”, “work”, or “finished the lab”.



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.

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
commit af9db9b5681ff748847eca6ddedb46069e1b0366 ↵
 (HEAD -> master)
Author: Homer Simpson <simpsonh@lardlad.donuts>
Date:   Wed Aug 30 10:58:28 2017 -0500
```

Add `hello.cpp`

That's great! Our timeline 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.
- **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 anything about 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`.

## Seeing What Happened in a Commit

Let's make another commit. Maybe we'll make our hello world program ask the user how their day is going as well.

```
$ vim hello.cpp
$ git add hello.cpp
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   hello.cpp
$ git commit
```

Now that we've done this, let's see what our log looks like:

```
$ git log
commit bdf0003404901363f05b14f20bb7f7becf4b2dd5 ↵
(HEAD -> master)
Author: Homer Simpson <simpsonh@lardlad.donuts>
Date:   Wed Aug 30 12:54:59 2017 -0500
```

Add Faux Politeness

```
commit af9db9b5681ff748847eca6ddedb46069e1b0366
Author: Homer Simpson <simpsonh@lardlad.donuts>
Date:   Wed Aug 30 10:58:28 2017 -0500
```

Add hello.cpp

Neat! There are both our commits there, listed in order from newest to oldest. We can see who wrote them, when, and what the commit message was. Usually this is all you need, but sometimes you might want to see what actually changed in a commit. Maybe you've just cloned someone else's repository and you're confused about something they did, or maybe you just want to savor a particularly delicious line of code you wrote. We don't judge.

If you want to inspect what changed in one commit, you need some way of identifying that commit to Git. Git assigns a **hash** to each commit — that is the long string of letters and numbers on the line starting with **commit** in the log.<sup>10</sup> You can refer to commits by their hash whenever you need to. Since it's pretty unlikely that two commit hashes start with the same handful of characters, you can refer to a commit by just the first several letters of its hash;

---

<sup>10</sup>If you're curious, it's a SHA1 hash of the changes, the commit message, the author and time, the parent commit hash, and one or two other things. Basically, it gives you some way of turning all the interesting parts of a commit into a string that uniquely identifies that commit.

five or so is usually plenty.<sup>11</sup> (`git commit` shows the first seven characters in its output.)

The `git show` command shows you what changes happened in a commit. So, if we want to see our beautiful, lovely, astute changes that we made in that latest commit, we can do that:

```
$ git show bdf00
commit bdf0003404901363f05b14f20bb7f7becf4b2dd5 ↵
    (HEAD -> master)
Author: Homer Simpson <simpsonh@lardlad.donuts>
Date:   Wed Aug 30 12:54:59 2017 -0500
```

Add Faux Politeness

```
diff --git a/hello.cpp b/hello.cpp
index 9a52d30..e0d3a48 100644
--- a/hello.cpp
+++ b/hello.cpp
@@ -5,5 +5,7 @@ using namespace std;
 int main()
 {
     cout << "Hello world!" << endl;
+   cout << "How are you?" << endl;
+   //ignore user feelings and exit
     return 0;
 }
```

This shows a ‘diff’ of your changes. If you see a + in the first column, that means you added that line; a - means that line was removed. Git works line-by-line, so if you change something on a line, it shows up as you deleting one line and adding another. The rest of the output provides some context around your changes so you have a vague idea of where they happened.

## 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 are two commits 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 commits to GitLab first.

---

<sup>11</sup>There are about a million 5-digit hex numbers, so unless you’re the kind of person to get hit by lightning twice, odds are in your favor that you won’t have two commits starting with the same 5 digits, even if you’ve got a couple hundred commits in your repository!

```
$ 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. You can view your commit history and even make commits!

## Oh, Fork.

Close your eyes again.<sup>12</sup>

Here you are working on that Next Big Thing again. As you code, you see a beautiful person emerge<sup>13</sup> from the field's tall grass.

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

You 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>14</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 users happy.

You have two choices:

- Buck up and commit to redoing your entire project, leaving your pals (the users) grumpy about the bugs you need to fix.
- Ignore the idea from the muse and continue fixing 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 last option sounds *much* better.

---

<sup>12</sup>Maybe just half-closed this time.

<sup>13</sup>It's as though they were laying there in the grass the whole time. So weird.

<sup>14</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>15</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>16</sup> smoking a pipe,<sup>17</sup> sipping on bourbon,<sup>18</sup> and wondering what commands you use to actually work with branches in Git. It's time to get your hands dirty.

First, let's notice something about our repo's **status**:

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

See that first line? Git starts you off on a branch named **master**. Think of **master** as "pointing at" the newest commit on the master branch. Every time you make a new commit while you're "on branch master", Git moves the **master** pointer to point to the commit you just made.

Let's also examine the **log** with this knowledge in mind. For this section, we're going to pass a few flags to **git log** so that it prints out beautiful ASCII art:

- **--oneline** summarizes each commit in just one line.

---

<sup>15</sup>You should *absolutely* see Primer if you haven't. Who doesn't love a good indie time travel movie?

<sup>16</sup>I assume.

<sup>17</sup>I reckon.

<sup>18</sup>That's a given.

- `--graph` draws an ASCII commit graph on the left.<sup>19</sup>
- `--all` shows all branches in the log so that we get a view of the whole repo, not just the branch we're on now.

```
$ git log --oneline --graph --all
* bdf0003 (HEAD -> master) Add Faux Politeness
* af9db9b Add hello.cpp
```

Ok, ok, it's not *very* beautiful. But, it does show our two commits and indicates that **master** is right now pointing at commit **bdf0003**. It also shows this other thing, **HEAD**. What's that about?

You can think of **HEAD** as a pointer to the branch you're currently on.<sup>20</sup> When you make a new commit, the branch **HEAD** is pointing at is the one that gets updated to point to your new commit.

Let's see what happens when we make a new branch. Obviously, the command we use for this is `git checkout`, not `git branch`. `git checkout` moves your **HEAD** around; passing the `-b` flag tells it to

1. Create a new branch pointing at the commit **HEAD** is at right now, then
2. Point **HEAD** at the branch it just made.<sup>21</sup>

Let's do something with those feelings the user has.

```
$ git checkout -b handle-feelings
Switched to a new branch 'handle-feelings'
$ git status
On branch handle-feelings
nothing to commit, working tree clean
```

Neat! We've made a new branch and now we're ready to make some commits on it. Let's also check the log real quick:

```
$ git log --oneline --graph --all
* bdf0003 (HEAD -> handle-feelings, master) Add Faux Politeness
* af9db9b Add hello.cpp
```

As expected, now we have *two* branches pointing at commit **bdf0003** and our **HEAD** is pointing at the **handle-feelings** branch.

---

<sup>19</sup>Git thinks of commits as a directed acyclic graph — you'll learn about this in discrete math. In short, a 'graph' is effectively connect-the-dots for mathematicians: there are vertices (in Git's case, commits) and arrows drawn between the vertices (in Git's case, each commit has an arrow pointing to the commit before it).

<sup>20</sup>As an aside, **HEAD** can also point directly to commits. This is called a 'detached HEAD' state, and if you're imagining unscrewing your head from your neck and using your arms to put it someplace you can't easily get to, you've got the right idea: it's sometimes handy, but it's a little weird and not something you'd want to do that often.

<sup>21</sup>Okay, okay, you can also do this with the `git branch` command in two steps: first create the branch with `git branch handle-feelings`, then switch to that branch with `git checkout handle-feelings`. Why `git checkout -b` instead of `git branch -c`? I have no clue; go ask Linus Torvalds.

Make a few changes to `hello.cpp` — perhaps read the user's feelings in to a string and then tell them that you (the computer) empathize with them. Then commit your changes:

```
$ vim hello.cpp
$ git add hello.cpp
$ git commit
[handle-feelings e146ae9] Empathize with the user!
1 file changed, 5 insertions(+), 1 deletion(-)
```

Alright, close your eyes for a second and imagine how the log will look. Knotty? Maybe a mushroom or two growing from a split in the side? Is it a bright, cheery home to a colony of termites?

Ok, open your eyes and let's see if your imagination was right:

```
$ git log --oneline --graph --all
* e146ae9 (HEAD -> handle-feelings) Empathize with the user!
* bdf0003 (master) Add Faux Politeness
* af9db9b Add hello.cpp
```

The `master` branch is still pointing at commit `bdf0003`, but `handle-feelings` is pointing at our brand new commit, `e146ae9`. No termites, but hopefully that's more or less what you imagined.

So far our commit graph is a straight line. Let's stick a fork in it by making some changes to the `master` branch. First, we switch which branch `HEAD` points at:

```
$ git checkout master
Switched to branch 'master'
$ git log --oneline --graph --all
* e146ae9 (handle-feelings) Empathize with the user!
* bdf0003 (HEAD -> master) Add Faux Politeness
* af9db9b Add hello.cpp
```

The commit log doesn't change, but it does show that we've moved our `HEAD`.

Now, let's ask for the user's name!

```
$ vim hello.cpp
$ git add hello.cpp
$ git commit
[master 0a405b6] Ask for the user's name!
1 file changed, 7 insertions(+)
```

This is where the log gets interesting. As you can see, there's a commit on the `master` branch that's not on the `handle-feelings` branch, and vice-versa:

```
$ git log --oneline --graph --all
* 0a405b6 (HEAD -> master) Ask for the user's name!
| * e146ae9 (handle-feelings) Empathize with the user!
|/
```

```
* bdf0003 Add Faux Politeness
* af9db9b Add hello.cpp
```

This is all well and good — in one timeline (**handle-feelings**), we’ve written a program that says hi and asks about the user’s feelings; in the other (**master**), we’ve asked for their name. We can switch between these two timelines as much as we want by using **git checkout**.

But, what if we want a program that does *both things*? Can we un-split these two timelines?

Yes!

For this task we use the **git merge** command. This command merges another branch into the branch we’re currently on.

Let’s merge **handle-feelings** into **master**. (If you’re not already on the master branch, switch to it with **git checkout master**.)

Usually merges go off without a hitch — Git is pretty good at figuring out how to smush code together. However, sometimes trouble does arise, typically when both of your branches have modified the exact same bit of code. Rather than letting you become your own grandma or otherwise introducing classic sci-fi timetravel paradoxes, Git asks you, the omniscient time-lord,<sup>22</sup> to sort out the paradox yourself.

Let’s see what happens when we merge our two branches together:

```
$ git merge handle-feelings
Auto-merging hello.cpp
CONFLICT (content): Merge conflict in hello.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

When a merge conflict happens, Git marks the conflicting changes in the file it’s confused about. The bits between <<<<<< HEAD and ===== are the lines from the file on the branch you’re on now. The bits between ===== and >>>>>> **handle-feelings** are the lines from the file on the branch you’re trying to merge.

You need to look at these changes and decide for yourself how to merge them. Sometimes you’ll keep one bit or the other, sometimes both, and sometimes parts of each.

Here’s what **hello.cpp** looks like:

```
$ cat hello.cpp
#include<iostream>
#include<string>

using namespace std;
```

---

<sup>22</sup>Well, time-lord of your repository, anyhow.



```

int main()
{
<<<<<<< HEAD
    string name;
=====
    string feelings;
>>>>>>> handle-feelings

    cout << "Hello world!" << endl;

    cout << "What's your name, friend? ";
    cin >> name;

    cout << "How are you?" << endl;
    cin >> feelings;
    cout << "I empathize with how you feel, pal" << endl;
    return 0;
}

```

This is not too bad — Git is just confused because both variables got declared on the same line. It's as if, when we split time, in one timeline we put a banana on the table, and in the other we put an apple in the exact same spot. When we try to merge the timelines together, rather than giving us a banappleana, Git asks us what to do. We just need to move the banana or the apple over a little so they aren't occupying the same spot on the table.

In other words, we want to keep both variables, so we can just delete the <<<<<<< HEAD, =====, and >>>>>>> handle-feelings lines. Once we do this, we need to tell Git that we've resolved the conflict by **git adding** the file, then **git committing** the result. Git writes the merge commit message for you, so you can just save and quit your editor right away.

If you check **git status** as you go, it will tell you what step to do next.

```

$ vim hello.cpp
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

```

```

Unmerged paths:
  (use "git add <file>..." to mark resolution)

```

```

    both modified:   hello.cpp

```

```

no changes added to commit ↵

```

```
(use "git add" and/or "git commit -a")
$ git add hello.cpp
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)
```

Changes to be committed:

```
    modified:   hello.cpp
```

```
$ git commit
[master 40996e7] Merge branch 'handle-feelings'
```

Let's see how this looks in the log:

```
$ git log --oneline --graph --all
* 40996e7 (HEAD -> master) Merge branch 'handle-feelings'
|\
| * e146ae9 (handle-feelings) Empathize with the user!
* | 0a405b6 Ask for the user's name!
|/
* bdf0003 Add Faux Politeness
* af9db9b Add hello.cpp
```

We can see the merge commit has *two* lines going into it! This is because it is made from two commits — one from each branch we merged. The **master** branch now has both its changes and the changes from **handle-feelings** in it; the **handle-feelings** branch is still pointing exactly where it used to be.

To get a better feel for the mechanics of branching and merging, we recommend you try out <http://learngitbranching.js.org/>. At the very least, 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.

## Ignoring Stuff

If you're working on a programming project, you'll probably end up with executables (**a.out** and friends) hanging around in your directory. Every time you run **git status**, you'll see something like

```
On branch master
Untracked files:
```

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

a.out

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

At first, you may be tempted to just go ahead and commit that executable too. But, this is not typically the best of ideas. It's very easy to forget to compile right before committing, and if you don't, you'll end up with a commit where the source code says one thing and the executable does another!

The general advice is to track your *source files* with Git and to *not* track any files that you *generate* from those (so, executables). So, if you don't want to ever commit a.out, do you have to settle for just always seeing it show up in git status?

No!

You can tell Git to ignore files by listing their names in a file in your repository named .gitignore. So, to ignore a.out, you could do something like this:

```
$ echo 'a.out' > .gitignore # or use a text editor
$ cat .gitignore
a.out
```

```
$ git status
On branch master
Untracked files:
```

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

.gitignore

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

No more a.out! And, if your cat happens to walk across your keyboard and type git add a.out, Git will tell you, "Hey, you said to ignore that! No way am I tracking it!"

(It's not a bad idea to commit your .gitignore.)

## 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. In your own words, what happens when you clone your `my-fancy-project` repository?
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. In your own words, what does merging two branches together do?
4. If you have a merge conflict in `rats.cpp`, what series of commands would you run to fix the conflict and complete the merge?

## 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:

```
1  #!/bin/bash
2
3  g++ *.cpp
4  ./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.

For simple scripts, this may be all you need! Just stick a list of commands you always run together into a file, and you have a shell script.

However, more complicated tasks will require an actual programming language with variables and loops and all that jazz. Fortunately, Bash has all of these things! Let’s check out some of those features and use them to improve this example.

## Variables

### Using Variables

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 `$`:

```
1 #!/bin/bash
2
3 # Assign the value "big" to the variable COW
4 # (by the way, things that start with # are comments)
5 COW="big"
6
7 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.

---

<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” (`!`).



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, 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`:

```
$ g++ no-such-file.cpp
g++: error: no-such-file.cpp: No such file or directory
g++: fatal error: no input files
compilation terminated.
$ 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 `$0`; this is commonly used for looping over all arguments passed to a command. The number of arguments is stored in `$#`; if no arguments are passed, its value is `0`. (If you've read Appendix D, these are analogous to the `argv` and `argc` parameters passed to `int main()` in C++ programs. However, `argc` counts the 0th argument; `$#` does not.)

---

<sup>4</sup>This is the very same value as what you return from `int main()` in a C++ program!

<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.

## 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 example to allow the user to set the name of the executable to be produced:

```
1 #!/bin/bash
2
3 g++ *.cpp -o "$1"
4 ./"$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:

```
1 #!/bin/bash
2
3 # Emit the appropriate greeting for various people
```

---

<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.

```

4
5 if [[ $1 = "Jeff" ]]; then
6     echo "Hi, Jeff"
7 elif [[ $1 == "Maggie" ]]; then
8     echo "Hello, Maggie"
9 elif [[ $1 == *.txt ]]; then
10     echo "You're a text file, $1"
11 elif [ "$1" = "Stallman" ]; then
12     echo "FREEDOM!"
13 else
14     echo "Who in blazes are you?"
15 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 type of comparison to use, rather than hoping that Bash guesses right.<sup>8</sup>

### Comparing Strings:

- `=,==` means either:
  - String equality, if both operands are strings, or
  - Pattern (glob) matching, if the RHS is a glob (e.g., `*.txt`).
- `!=` means 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.
- `-n`: The string is not empty (e.g., `[[ -n "$var" ]]`).
- `-z`: The string is empty (length is zero).

### 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.

---

<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`.

## 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.

Here's an example that, given a directory, lists the files in it; and given a file, prints the contents of the file:

```
1  #!/bin/bash
2
3  # First, check to make sure we got an argument
4  if [[ $# -eq 0 ]]; then
5      echo "Usage: $0 <file or directory name>"
6      exit 1
7  fi
8
9  # Then, determine what we ought to do
10 if [[ -f "$1" ]]; then
11     cat "$1"
12 elif [[ -d "$1" ]]; then
13     ls "$1"
14 else
15     echo "I don't know what to do with $1!";
16 fi
```

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:

```
1  #!/bin/bash
2
3  num=5
4
5  if [[ ($num -lt 3) && ("story.txt" == *.txt) ]]; then
6      echo "Hello, text file!"
7  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 \$:

```
1  #!/bin/bash
2
3  x=5
4  y=7
5
6  if (( x < y )); then
7      echo "Hello there"
8  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:

```
1  #!/bin/bash
2
3  case $1 in
4      a)
5          echo "a, literally"
6          ;;
7      b*)
8          echo "Something that starts with b"
9          ;;
10     *c)
11         echo "Something that ends with c"
12         ;;
13     "*d")
14         echo "*d, literally"
15         ;;
16     *)
17         echo "Anything"
18         ;;
19  esac
```

Do not forget the double semicolon at the end of each case — `;;` is *required* to end a case. They are analogous to `break` in C++; bash case statements do not have fallthrough.

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 `[[ ]]`.

```
1  #!/bin/bash
2
3  # If the user specifies an executable name, use that name;
4  # otherwise, name the executable 'a.out'
5  if (( $# > 0 )); then
6      g++ *.cpp -o "$1"
7      exe="$1"
8  else
9      g++ *.cpp
10     exe=a.out
11 fi
12
13 # Run the program only if it successfully compiled!
14 if [[ $? -eq 0 ]]; then
15     ./"$exe"
16 fi
```

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

## Arithmetic

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

For example,

```
1  #!/bin/bash
2
3  x=5
4  y=7
5  (( sum = x + y ))
6  echo $sum
```

Operator names follow those in C; `(( ))` supports arithmetic, bitwise, and logical operators. You can also write ternary expressions! One difference between

C and (( )) is that \*\* can be used for exponentiation: ((four = 2\*\*2)). 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:

```
1  #!/bin/bash
2
3  for file in *.sh; do
4      echo $file
5  done
```

Or sum all command-line arguments:

```
1  #!/bin/bash
2
3  sum=0
4
5  for arg in "$@"; do
6      (( sum += arg ))
7  done
8
9  echo $sum
```

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

```
1  #!/bin/bash
2
3  for (( i=1; i < 9; i++ )); do
4      echo $i;
5  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 will also work with a while loop.

For example,

```
1  #!/bin/bash
2
3  input=""
4  while [[ "$input" != "4" ]]; do
5      echo "Please guess the random number: "
6      read input
7  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. It is quite similar to `getline()` in C++.

## “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:

```
1  #!/bin/bash
2
3  # Declare a function named 'parrot'
4  parrot() {
5      while (( $# > 0 )); do
6          echo "$1"
7          shift
8      done
9  }
10
11 # Call parrot() with some arguments
12 parrot These are "several arguments"
```

(`shift` is a built-in that 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:

```
1  #!/bin/bash
2
3  average() {
4      sum=0
5      for num in "$@"; do
```



```

6         (( sum += num ))
7     done
8
9     (( avg = sum / $# ))
10    echo $avg
11 }
12
13 my_average=$(average 1 2 3 4)
14
15 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`.

One word of warning: in Bash, all variables are globally scoped by default, so it is easy to accidentally clobber a variable:

```

1  #!/bin/bash
2
3  # create a bunch of files with "hello" in them
4  create_some_files() {
5      for ((i=0; i < $1; i++)); do
6          echo "What file would you like to create?"
7          read input
8          echo "hello" > "$input"
9      done
10 }
11
12 # get the number of files the user wants to create
13 echo "How many files do you want me to create?"
14 read input
15 create_some_files "$input"
16
17 # BUG: this prints the last created filename!
18 echo "Created $input files"

```

You can scope variables to functions with the `local` builtin; run `help local` in a Bash shell for details.

## Tips

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

When writing scripts, sometimes you will want to change directories — for instance, maybe 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`.

```
$ pwd
~/cool_code
$ dirs
~/cool_code
```

**pushd /tmp** changes the current directory to `/tmp`.

```
$ pushd /tmp
$ pwd
/tmp
$ dirs
/tmp ~/cool_code
```

Calling **popd** then removes `/tmp` from the stack and changes to the next directory in the stack, which is `~/cool_code`.

```
$ pwd
/tmp
$ popd
~/cool_code
$ pwd
~/cool_code
$ dirs
~/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.

## Customizing Bash

Let's say you've typed out our example script for compiling and running C++ code and you've put it in a directory called **scripts** in your home directory. So, the path to the script is `~/scripts/runit.sh`.

Now, if you're off in some other directory (say `~/cs1510/hw3`), you can run that script like so:

```
$ ~/scripts/runit.sh
```

But this isn't very cool; you want to be able to run it like a regular ol' command and not have to type the dang path out each time! Well, there is a solution. When you type a command in at the Bash prompt that isn't the full path to the program you want to run, Bash has a list of places it looks. This list is stored in the `$PATH` variable; the list of directories is delimited by colons.

Your `$PATH` may look something like this:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

When you type in a command, say, `g++`, Bash looks through each directory in `$PATH` in order until it finds one with an executable named `g++` in it. You can see which one it picks out using the `which` command:

```
$ which g++
/usr/bin/g++
```

You can add your own directories to `$PATH`! So, if you want to be able to call `runit.sh` from wherever, you could do the following:

```
$ PATH=~/.scripts:$PATH
$ which runit.sh
~/.scripts/runit.sh
```

The first command puts your `scripts` directory as the first directory in the path to search. The second command demonstrates that now Bash knows where to find `runit.sh` regardless of where you execute it from!

There's one problem with this setup: Bash won't remember that you added `~/.scripts` to the `$PATH`. You need some way to run that assignment every time a new Bash instance executes. Fortunately, there is a file in your home directory called `.bashrc` that Bash executes every time it runs.

So, open up `~/.bashrc` and at the end, enter the following line:

```
export PATH=~/.scripts:$PATH
```

(The `export` command causes the value of `$PATH` set in the script to apply even after the script exits.)

You can also add other customizations in your `.bashrc`. Rather than writing a script as its own file, you can write it as a function instead and store it in your `.bashrc`.

There are also a number of variables that you can set to configure other parts of your environment — for instance, setting `$PS1` changes your shell prompt's text. See `help variables` for a list of knobs and dials you can fiddle with.

## 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

### Conditions

#### `[ [ ] ]`

Comparing Strings:

- `=,==` means either:
  - String equality, if both operands are strings, or
  - Pattern (glob) matching, if the RHS is a glob (e.g., `*.txt`).
- `!=` means 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.
- `-n`: The string is not empty (e.g., `[ [ -n "$var" ] ]`).
- `-z`: The string is empty (length is zero).

Comparing Numbers:

- `-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:

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

Boolean Logic:

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

#### `(( ))`

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

## Statements

```
if <condition>; then
    <commands to run>
fi

case <variable or expression> in
    first-case)
        <commands>
        ;;

    second-case)
        <commands>
        ;;
esac
```

### Iterating loop:

```
for <variable-name> in <array>; do
    <commands>
done
```

### Counting loop:

```
for ((i=0; i < 7; i++)); do
    <commands>
done
```

### While loop:

```
while <condition>; do
    <commands>
done
```

## Further Reading

- [Bash Manual](#)
- [Bash Guide](#)
- [Bash Tutorial](#)
- [Advanced Bash Scripting Guide](#)

## Chapter 5

# 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:

```
1 pickles: cucumbers vinegar dill
2     brine --with=dill cucumbers
3
4 cured-brisket: brisket vinegar
5     brine brisket
6
7 paprika: red-peppers
8     sun-dry red-peppers | grind > paprika
9
10 rub: paprika garlic mustard-seed peppercorns
11     grind paprika garlic mustard-seed peppercorns > rub
12
13 smoked-brisket: cured-brisket rub
14     season --with=rub cured-brisket -o seasoned-brisket
15     smoke --time=8 hours seasoned-brisket
16
17 pastrami: smoked-brisket
18     steam --until=tender smoked-brisket
19
20 dough: rye wheat coriander yeast sugar water
21     mix --dry-first --yeast-separately rye wheat coriander yeast \
22         sugar water --output=dough
23
24 rye-loaf: dough
25     rise --location=beside-smoker dough && bake -t 20m
26
27 mustard: mustard-seed vinegar
```

---

<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.



```
28     grind mustard-seed | mix vinegar > mustard
29
30 sandwich: pastrami pickles rye-loaf
31     slice rye-loaf pastrami
32     stack bread-slice --spread=mustard pastrami bread-slice
33     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 doesn't know exactly what memory address that thing will end up at. So, instead of putting in a memory address, it 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>

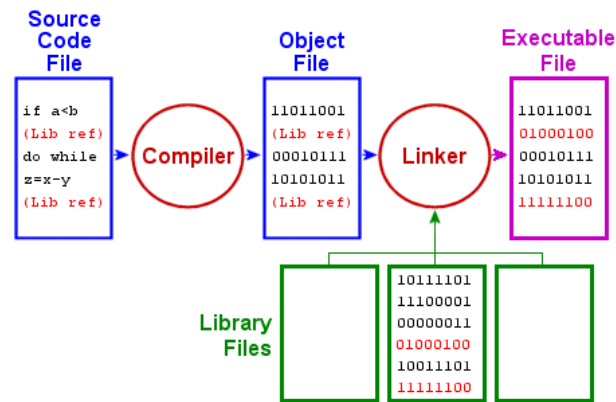


Figure 5.1: How Executables are Made<sup>12</sup>

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

<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.

<sup>12</sup>© 2003-2016 Keith Parkansky. Used with permission. Source: <http://www.aboutdebian.com/>

why we have **make**!

## 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:

```
1 program:
2     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>13</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 your **program** executable by running **make program** in your shell.<sup>14</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. Why doesn't **make** want to build our program again? **make** determines if it needs to re-build a target by comparing when the target

---

<sup>13</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>14</sup>Don't try to execute the makefile itself. **bash** is confused enough without trying to interpret **make**'s syntax!

was last modified to the modification times of the files the target depends on. We haven't listed any dependencies for the **program** target, so **make** doesn't think anything needs to happen!

Let's fix that. Since **make** is not omniscient,<sup>15</sup> we need to explicitly state what files each target depends on. For our example, let's suppose we have a classic CS 1570 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:

```
1 program: main.cpp funcs.h funcs.cpp
2      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 have one target depend on files produced by other targets! Then, **make** 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**:

```
1 program: main.o funcs.o
2      g++ main.o funcs.o -o program
3
4 main.o: main.cpp funcs.h
5      g++ -c main.cpp
6
7 funcs.o: funcs.cpp funcs.h
8      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>16</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

---

<sup>15</sup>Not yet, at least.

<sup>16</sup>Or tofu and potatoes, if that's your thing.

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>17</sup>

## Phony Targets

Building files is great and all, but there's more to life than just compiling code.<sup>18</sup> Sometimes you'll have a somewhat-complicated command that you'll want to run often — for example, commands to clean up all the files **make** generates in your current directory. (We'll also want something similar to run unit tests later on in this book.) You *could* write a shell script, but you've already got a **makefile**; why not use that?

However, **make** expects targets to generate files, so if you make a target named **clean** that just deletes stuff, it's possible for **make** to get a little confused. 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 targets that don't produce files through something called *phony targets*. You can tell **make**, “Hey, this target doesn't actually produce 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.

```
1 program: main.o funcs.o
2     g++ main.o funcs.o -o program
3
4 main.o: main.cpp funcs.h
5     g++ -c main.cpp
6
7 funcs.o: funcs.cpp funcs.h
8     g++ -c funcs.cpp
9
10 .PHONY: clean
11
12 clean:
13     -@rm -f program
14     -@rm -f *.o
```

Now when you run **make clean**, it will delete any object files, as well as your compiled program.<sup>19</sup> There are a few pieces to this target:

---

<sup>17</sup>At this point, have your past self pat you on the back. Good work!

<sup>18</sup>I hope so, at least...I've been stuck in this basement compiling the Linux kernel by hand for the past 82 years!

<sup>19</sup>The **clean** target is commonly used in anger when the dang compiler isn't working right

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>20</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 was supposed to run after it probably won't work either. Rather than trying anyway, it stops and lets you know what command failed so you can fix whatever is broken. In this case, we don't care if there isn't a file named `program` to delete; we just want to delete it if it exists.
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.

## Variables

Variables come in handy in a number of places in makefiles. Maybe you don't want to have to manually list out every object file, or maybe you want to make it easy to switch which compiler you're using (I hear all the cool kids are using `clang++`<sup>21</sup>).

Here's the syntax for variables:

- `var=value` sets the variable `var` to `value`.
- `${var}` or `$(var)` accesses the value of `var`.<sup>22</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.

```

1 CFLAGS = -O2
2 DEBUGFLAGS = -g -Wall -Wextra
3
4 program: main.o funcs.o
```

---

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>20</sup>In other words, a non-zero return value from that command.

<sup>21</sup>And that it has nice error messages!

<sup>22</sup>Be careful not to confuse this with `bash`'s `$( )`, which executes whatever is between the parentheses.

```

5      g++ ${CFLAGS} main.o funcs.o -o program
6
7      .PHONY: debug clean
8      debug: CFLAGS=${DEBUGFLAGS}
9      debug: program
10
11     main.o: main.cpp funcs.h
12         g++ ${CFLAGS} -c main.cpp
13
14     funcs.o: funcs.cpp funcs.h
15         g++ ${CFLAGS} -c funcs.cpp
16
17     clean:
18         -@ 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.)

**Note:** 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 a lot more files, adding all those targets by hand would be a lot of work. Instead of writing each of these out manually, **make** has *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 in the current directory 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 file ending in **.o** from a matching **.cpp** file.

Hmm, but how would we write a command for this target? `g++` still needs to know the actual names of our files! `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:

```
1 SOURCES=$(wildcard *.cpp)
2 OBJECTS=$(SOURCES:%.cpp=%.o)
3 HEADERS=$(wildcard *.h)
4
5 program: ${OBJECTS}
6     g++ $^ -o program
7
8 %.o: %.cpp ${HEADERS}
9     g++ -c $<
```

Let's walk through this example step by step. First, we create three variables: `SOURCES` will keep track of all our `.cpp` files, `OBJECTS` contains the names of the object files we want to produce from our `.cpp` files, and `HEADERS` contains all our header files.

The `program` target now depends on all our object files. To produce the `program` executable, we tell `g++` to link all our object files into a `program` executable.

How do we get these object files? From the pattern rule at the end! The rule `%.o: %.cpp ${HEADERS}` says "to make a file named `whatever.o`, you need a file named `whatever.cpp` and every header file in the current directory". To produce `whatever.o`, we compile the first dependency (which will be `whatever.cpp`) with `g++`. With this pattern rule, you won't need to update your makefile if you add more files to your program later!

Why have every object file depend on all headers? In all honesty, this is something of a hack: we don't have a simple way to automatically figure out exactly which headers each `.cpp` file `#includes`. So, we just make it so any header change causes everything to recompile. Typically this is a fine assumption — in practice, you spend most of your time editing `.cpp` files, and when you do change a header (say, changing the type of a function parameter), that usually requires changes in a bunch of places anyway. If you're interested in more accurately calculating dependencies, check out the `makedepend` program!

Another upside of using patterns in makefiles is being able to easily copy an existing makefile into a new project. Done right, you'll have to change the program name (and if you make a variable for that, it's a change in one place) and little else!



## Questions

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

Consider the following makefile:

```
1 default: triangles
2
3 triangles: main.o TrianglePrinter.o funcs.o
4     g++ $^ -o triangles
5
6 main.o: main.cpp TrianglePrinter.h funcs.h
7     g++ -c main.cpp
8
9 TrianglePrinter.o: TrianglePrinter.cpp TrianglePrinter.h funcs.h
10    g++ -c TrianglePrinter.cpp
11
12 funcs.o: funcs.cpp funcs.h
13    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

### Useful **make** Flags

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

- **-j3** runs up to 3 jobs (i.e., builds 3 targets) in parallel.<sup>23</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**.

### Syntax

Do not forget: you must use tabs, not spaces, to indent commands in makefiles!

### Targets

Creating a target:

```
target-name: list of dependencies
    command-that-produces target-name from list of dependencies
```

Pattern targets:

```
%.o: %.cpp
    command that produces a .o file from a .cpp file
```

Telling **make** that a target doesn't produce a file:

```
.PHONY: target-name
```

### Variables

- **var=value** sets the variable **var** to **value**.
- **\${var}** or **\$(var)** accesses the value of **var**.<sup>24</sup>
- The line **target: var=thing** sets the value of **var** to **thing** when building **target** and its dependencies.
- **txtfiles = \$(wildcard \*.txt)** stores the name of every file matching the glob **\*.txt** in the **txtfiles** variable.

---

<sup>23</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.

<sup>24</sup>Be careful not to confuse this with **bash**'s **\$( )**, which executes whatever is between the parentheses.

- `cowfiles = $(txtfiles:%.txt=%.cow)` turns every `.txt` filename in `txtfiles` into a corresponding name ending in `.cow` instead.

Special variables in targets (known as [automatic variables](#)):

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

## 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 6

# 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>

It was all fun up to this point. But now you're here: standing in front of at an array of red-hot lasers, trapped in the vault of the Big Bank of New York, wondering where it all went wrong, and wearing a fancy turtle neck. At least you look good in a turtle neck. Not everyone can pull that off.

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

*BWEEEEEEEEEEEEEEEEEE!*<sup>3</sup>

With a deafening *bwuee*, 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.

After you gather yourself, you analyze the nest of wires pouring from the control panel and wonder what you did to set off the vault's laser defense system. You slow time and begin cutting the wires with the procedure you used last time.

<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.

<sup>3</sup>BWEE

Red wire, other red wire, otherer red wire. You look behind you: no lasers – everything’s good.

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.

You wake up.<sup>4</sup>

In real life, you may not be able to slow down or reset time to 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 bugs, but a debugger can definitely give you some clues to make it easier to find them.

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 chapter, we’ll be using the command line interface.

## Takeaways

- Learn general debugging practices
- Step through the execution of a compiled C++ program
- Inspect the contents of program variables

---

<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.

# Walkthrough

## Fatherly Debugging Advice

It's a sunny afternoon, so I take off work early and come by to pick you up from school. You run up and hop in the front seat of the car. "Hey kiddo, how was school? Do you want to get some ice cream?"

We drive over to the ice cream shoppe and I pull the ol' steering-with-my-knees trick that always makes you scream. "Dad! Stop it, you'll hit someone!" I laugh and mock-begrudgingly put my hands back on the wheel.

We sit down in a booth to enjoy a couple of sundaes. "How was your day? Are the other kids treating you well?" I ask.

"School was fine. I did great on my English paper! But..." You furrow your brow.

"What's on your mind?"

"Well, some of my friends have been programming. And sometimes their programs don't work, and they have to fix them. And I guess I'm worried that when my program doesn't work I won't be able to fix it!"

"My child," I begin.

I put a hand on your shoulder the way only a father can.

"My dearest,"

I put another hand on your shoulder.

"It seems like only yesterday you were but a wee babe. How you did cry and scream and do other things that babies do! I lost more sleep over you than I did working 26 hour shifts at the smoke alarm testing factory."

I put another hand on your shoulder.

"But there was one thing that always calmed you down. The instant I'd sit you next to a computer you'd smile and coo! Many an evening I rocked you to sleep cradled between the screen and keyboard of a Thinkpad 701C. That's when I knew you'd grow up to be a great programmer."

I put another hand on your shoulder. You are visibly embarrassed at this point.

"And now look at you! Practically all grown up! Your mother and I are so proud of you. Everyone has trouble with programs from time to time."

I put another hand on your shoulder.

"Here's what I do when I have to fix a program:"

## 1. What happened? What should have happened?

You should have an idea of how your program ought to work in your mind while programming. Maybe you've got a homework assignment writeup to go off of, or a design document that you wrote, or just an idea in your head that you're implementing.

Regardless of where that idea comes from, take a second to identify the wrong behaviour you're seeing and to figure out what your program should have done instead. If you can, get a real piece of paper and actually draw out a picture of what your program is doing; for example, if you're making a binary tree, draw the tree, or if you're communicating to a server, draw/write out the protocol your server and client should be following.

## 2. What the heck did you do to break it?!

Was your program working earlier? If so, think through the changes you just made. (You may find consulting `git diff` to be helpful!) How might they have caused the bug?

(If your program didn't work right in the first place, you'll have to try some other debugging technique. Sorry.)

## 3. How is the program getting the wrong result?

Start from where you saw the bug and work your way back through your program. What execution path did it follow? What values do your variables have? How do those values get set?

Make generous use of print (`cout`) statements! Print out the contents of your variables and print out messages that tell you what code is executing. Don't take anything for granted! Use those print statements to check your assumptions about how your code works.

## 4. Talk it out.

On the next page you will find Bertha.<sup>6</sup> She's very curious about people and would love to hear about your program. So, prop that page up next to your computer and explain exactly what is wrong with your program, just like you would explain it to a friend. Why do you think it's broken? What fixes did you try?

---

<sup>6</sup>Well, a picture of her. The publisher wouldn't let us include a live chicken tucked between the pages of the book.





Figure 6.1: Bertha!

(This may seem foolish, but this technique once revealed to me that I had been debugging code for an hour under the misapprehension that 9 was a prime number. You'd be amazed at the things you don't realize until you think about how to explain something to someone else.)

5. When all else fails, try again later.

If you've got a bug that you just can't seem to figure out, take a break. If there are other obvious bugs, fixing those may reveal the root cause of your problem bug. Otherwise, take a walk, or even better, get a good night's sleep and come back to it in the morning.

These debugging tips work for any program (and plenty of not-computer-related problems to boot). However, a debugger tool can make life easier for you, especially when you are tracing the execution of your program and inspecting how it behaves.

## Compilin' for Debuggin'

Compiled programs don't contain all the information that their source code does. 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. So, if you try to debug a program that's compiled in the usual manner, you'll have to wade through a lot of hex numbers and assembly code.

As a human person, you'd rather be able to look at your actual source code in the debugger. Fortunately, there is a way 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 fun.

## 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>7</sup>

```
$ 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 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   0x40086c in main() at main.cpp:8  
2      breakpoint    keep y   0x40088e in main() at main.cpp:10  
3      breakpoint    keep y   0x400894 in main() at main.cpp:12  
(gdb) delete 2  
(gdb) info breakpoints  
1      breakpoint    keep y   0x40086c in main() at main.cpp:8  
3      breakpoint    keep y   0x400894 in main() at main.cpp:12  
(gdb) delete  
Delete all breakpoints? (y or n) y  
(gdb) info breakpoints  
No breakpoints or watchpoints.
```

---

<sup>7</sup>Don't actually do this, though. We're just demonstrating how you'd pass command line 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
```

The backtrace shows you the function stack: starting from `main()`, which functions were called to get to the line that's currently executing? Think of it like a family history: this line of code lives in this function, which was called by this other function, which was called by *another* function, and so on and so forth all the way back to Adam, err, `main()`. 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 where the function is defined, along with the values of any arguments passed to it.

`backtrace` is usually the first tool you should reach for when debugging a segfault. It'll tell you right where your program is, and it might even show you the null pointer (if you're passing it as an argument to the function).

You can 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. It even knows how to print out struct and class instances!

`print` will also happily print out the results of C++ expressions. You can dereference pointers, do arithmetic, and even call functions!

For a summary of all the variables in the current function, use `info locals`. `info` can show a bunch of other interesting data about your program and about `gdb`.

### Setting the contents of variables

Sometimes you'll want to test out a theory — maybe you think you've found your bug, and you want to make a quick change to see if you know how to fix it. You can't change the code while `gdb` is running it, but you can poke stuff in memory to your heart's content.

You can change the values of variables on the fly with the `set` command:

```
(gdb) print x
$1 = 7
(gdb) set x = 30
(gdb) print x
$2 = 30
(gdb) step
11      if (x >= 10)
```

Here, instead of continuing execution on line 9, the code jumps to line 11 since we changed the value of `x`!

There's a lot more that `gdb` can do — disassembling functions, poking around in memory, debugging code running on other machines — but the stuff in this chapter will cover all of your usual debugging needs. Happy debugging!

## 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

- **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

- **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

- [GDB manual](#)
- [More on breakpoints](#)
- [KDbg](#), a GUI for gdb
- [rr](#), tool that can record and replay program execution!

## Chapter 7

# 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. You can also unintentionally overwrite other variables in your program, or interpret one kind of variable as another (say, treating a class as an `int`)! These sorts of memory bugs can cause your program to behave unpredictably and thus can be quite difficult to find. 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.

---

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

The first widespread internet worm, the [Morris worm](#), used a memory safety bug to infect other machines in 1988. Unfortunately, *thirty years* later, memory safety bugs are still incredibly common in popular software and many viruses continue to exploit them. For one example, the [WannaCry](#) and [Petya](#) viruses use a memory safety exploit called [EternalBlue](#) developed by the NSA<sup>2</sup> and released by Russian<sup>3</sup> hackers early in 2017.



Figure 7.1: 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>4</sup> but any allocation only lives as long as the function that makes it is executing. When a function is called, a *stack frame* is pushed onto the stack. The stack frame holds variables declared in that function as well as some bookkeeping information. Crucially, this bookkeeping 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.

---

<sup>2</sup>Allegedly

<sup>3</sup>Allegedly

<sup>4</sup>It's a joint effort between the compiler and the operating system.

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>5</sup> While the automatic management of the stack is nice, the freedom to be able to make memory allocations that live longer than the function that created them is essential, especially in large programs.

To help you get the mental picture right, 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>6</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>7</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:

```
1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << "Stack uninitialized value" << endl;
7      int x;
8      if(x > 5)
```

---

<sup>5</sup>Or if you're writing C, with the `malloc` and `free` functions.

<sup>6</sup>It's not even a good source of random numbers, unless you like random numbers that aren't very random.

<sup>7</sup>It doesn't always do this because statically analyzing software (i.e., at compile time) is Really Hard, but it still catches most obvious things.

```

9      {
10         cout << "> 5" << endl;
11     }
12
13     cout << "Heap uninitialized value" << endl;
14     int* y = new int;
15     if(*y < 5)
16     {
17         cout << "< 5" << endl;
18     }
19
20     delete y;
21
22     return 0;
23 }

```

Your first hint that this isn't right is from the compiler itself if you use the `-Wall` flag.<sup>8</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>9</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. To run the above program (named `uninitialized-value`) through Valgrind, do the following:

```
$ valgrind --track-origins=yes ./uninitialized-value
```

When we do this, we get messages about both of our variables.

The stack-allocated uninitialized value is 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

```

<sup>8</sup>Lines that are too long to fit on the page have been split over multiple lines; this is indicated by the ↵ symbol.

<sup>9</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==      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:

```
==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>10</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 (as opposed to a pointer *to* 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 runtime 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.

---

<sup>10</sup>And more generally, any uninitialized value being accessed through a pointer.

To enable address sanitizer, you must use the following flags to `g++`: `-g -fsanitize=address -fno-omit-frame-pointer`.<sup>11</sup> Furthermore, you need to set two environment variables<sup>12</sup> if you want function names and line numbers to appear in `asan`'s output:

```
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.

## Out-of-bounds Stack Access

First up, out-of-bounds accesses on a stack-allocated array:

```
1 #include<iostream>
2
3 int main()
4 {
5     int array[5];
6     array[5] = 5; // Out-of-bounds write
7     std::cout << array[5] << std::endl; // Out-of-bounds read
8
9     return 0;
10 }
```

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. Compile the program with this command:

```
$ g++ -g -fsanitize=address -fno-omit-frame-pointer ↵
    invalid-stack.cpp -o invalid-stack
```

`asan`'s output is somewhat terrifying to see, but the relevant parts look like this:<sup>13</sup>

---

<sup>11</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>12</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``

<sup>13</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.



```

==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`.

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!

## Out-of-bounds Heap Access

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

```

1  #include<iostream>
2
3  int main()
4  {
5      std::cout << "Invalid write" << std::endl;
6      int *arr = new int[5];
7      for(int i = 0; i <= 5; i++)
8      {
9          arr[i] = i;
10     }
11
12     delete[] arr;

```

```

13     return 0;
14 }

```

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 0x7fffee305a690 ↵
    sp 0x7fffee305a680
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>14</sup> (in other words, after the end of) our allocated chunk, indicating that we are writing one index past the end of the array.

## Use After Free

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>15</sup> Like out-of-bounds accesses, this type of bug can go undetected if you don't check for it; the below example appears to work, even though it is incorrect!

<sup>14</sup>Did you know that chickens also visually organize smaller quantities on the left and larger quantities on the right?

<sup>15</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.

```

1  #include<iostream>
2
3  int main()
4  {
5      int* x = new int[5]; // Declare and initialize an array
6      for(int i = 0; i < 5; i++)
7      {
8          x[i] = i;
9      }
10
11     int* y = &x[1]; // "Accidentally" hold a pointer to an array element
12
13     delete [] x; // Delete the array
14
15     std::cout << *y << std::endl; // Uh-oh!
16
17     return 0;
18 }

```

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)

```

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 (the size of an `int`) read happened 4 bytes (so, at index 1) inside the block of 20 bytes that is our array of 5 `ints`.

Invalid read or write bugs can have a number of fixes. Sometimes they're as simple as adding a bounds check somewhere to make sure you don't write off the end of an array. Other times, you'll have to think carefully about where your code went astray — see the Debugging chapter for more advice on this.

## 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>16</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:

```
==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
```

---

<sup>16</sup>The implementation of `delete []` isn't specified, but the size of the allocation has to be 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.

#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.

A double-delete occurs when you delete the same block of memory twice. 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,

```
1  #include<iostream>
2
3  int main()
4  {
5      int* x = new int(5);
6      delete x;
7
8      int* y = new int(7); // Probably allocated where x was
9      // If we were to print out *x and *y, we would likely
10     // see "7" for both values!
11
12     delete x; // Either deletes y or explodes
13     delete y; // If the last line didn't explode, BOOM!
14
15     return 0;
16 }
```

(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. Sometimes a quick fix for this is to set your pointers to **NULL** after you call **delete**. Other times, you'll get a double-delete if you forget to write a copy constructor for a class (or write a buggy one).

## 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:

```
1 struct List
2 {
3     int value;
4     List* next;
5 };
6
7 int main()
8 {
9     List l;
10    l.value = 5;
11    l.next = new List;
12
13    l.next->value = 6;
14    l.next->next = new List;
15
16    //These two lines would fix the memory leaks
17    //delete l.next->next;
18    //delete l.next;
19
20    return 0;
21 }
```

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 ↵
==649==      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>17</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.

Sometimes memory leaks come from a missing or buggy destructor. Other times, they happen when you accidentally overwrite a pointer, say in a buggy `insert()` function. These can be more difficult to track down; again, see the Debugging chapter for advice.

---

<sup>17</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>18</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>18</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 8

# Unit Testing

### Motivation

You're the quality control manager for the Confabulator, a hot new product in development by Acme Inc. LLC. Every day, engineers give you new prototype confabulators to test and make sure they meet the ill-defined and entirely made-up requirements upper management has set for the product. You put each confabulator through its paces — to the best of your memory of what those paces are — and make a report for the designers telling them what works and what doesn't.

Usually they give you the same prototype back with some fixes for the problems you reported the previous day. Testing confabulators takes up time you could otherwise use productively, maybe for complaining about life to your coworkers or taking extended lunch breaks. So you don't always put the fixed-up prototype through the full battery of tests — you just test the stuff that got fixed. Unfortunately, sometimes (as engineers are wont to do) a fix gets added that affects something else in an entirely different spot in the system! You'll never forget the endless meetings after it took you a week to discover that a fix to the frobnicator interfered with the rotagrator arm. Those engineers were *mad*. How could you tell them everything is fine when something was broken?

Even worse, it's hard to remember exactly what tests you're supposed to do each time, and management seems to keep changing their minds on exactly what features a confabulator is supposed to have. Last month it was just supposed to be for annularity congruification, but then a contract with Statorfile Exceed GmbH. came through and now it also has to calabricate the vibrosity of splinal conformities. The number of things you have to check for just seems to get bigger and bigger, and of course once you add on a vibrous harmonicator, you have to check that it works regardless of whether the radiometer intensimission

is engaged or disengaged. It seems like every new whizbang adds a half a dozen whatsits and thus a gross more tests<sup>1</sup> for you.

But the worst thing of all is that you have to do this all by hand, day in and day out. Nothing rots the brain faster than the dull monotony of doing something you're ambivalent about. Heck, if you're going to be experiencing dull monotony either way, the least your boss could do is let you watch some reality TV. But nooooo, apparently all the monotony must be job-inflicted.

Fortunately, computers have solved this problem! Instead of manually testing your program, you can write *unit tests* that test each piece for you. Once you add a new feature and write tests for it, you can run your program through a full test suite to make sure everything works as intended. Rather than mind-numbingly checking everything by hand, you can experience the monotony of writing a test once and then forget about it forever (or at least until it fails and you have to figure out what you broke).

Unit testing is widely used in industry because it is quite effective at keeping bugs out of code.<sup>2</sup> You can even measure how much of your code is tested by unit tests — 100% code coverage means that you've found at least most of the obvious bugs!

This chapter will focus on the Catch unit testing framework for C++. There are a number of popular unit testing frameworks; Boost has one,<sup>3</sup> Google makes one called **gtest**, etc. However, Catch is easy to install, easy to write tests in, and downright beautiful compared to Boost's test framework. (It's also popular, in case you were wondering.)

## Takeaways

- Learn how to write unit tests with Catch
- Organize your code and tests to preserve your sanity
- Measure how much of your code is covered by the tests you've written

---

<sup>1</sup>That is, 144 more disgusting tests.

<sup>2</sup>Pedantry: unit tests technically cannot show the absence of all bugs; they can just show that under certain circumstances your program does not have bugs. In logical terms, unit tests are a bunch of “there exists” statements; whereas a proof of correctness is a “for all” statement. Unfortunately, proving programs correct is a difficult task and the tools to do so are not exactly ready for widespread use yet. In the meantime, while we wait for math and logic to catch up to the needs of engineering, we'll have to settle for thorough unit testing.

<sup>3</sup>Of course it would — Boost even has a kitchen sink library.

# Walkthrough

## Setting up Catch

Catch is distributed as a single `.hpp` file that you can download and include in your project. Download it from [GitHub](#) — the link to the single header is in the README.

In *exactly one* `.cpp` file, you must include the following lines:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

This generates a `main()` function that runs your unit tests. You will have two programs now — your actual program, and a program that runs your unit tests.

Every other file you write tests in should include Catch:

```
#include "catch.hpp"
```

Later, we'll discuss how best to organize your tests, so don't worry too much about the "right" place to put the main function yet. For now, just throw it at the top of a `test.cpp` file, and let's write some tests!

## Basic Tests

We are going to test a function that generates Fibonacci numbers (1, 1, 2, 3, 5, 8, ...). Here's our function:

```
1  /* Generate the Nth Fibonacci number */
2  int fibonacci(int n)
3  {
4      if(n <= 1)
5      {
6          return n;
7      }
8      else
9      {
10         return fibonacci(n - 1) + fibonacci(n - 2);
11     }
12 }
```

In Catch, every test lives inside a `TEST_CASE` block. You can make these as fine-grained as you want, but generally you'll find it easy to collect a bunch of related checks into one `TEST_CASE`. Each test case has a name and a tag; generally you'll tag all test cases for a function/class with the same tag. (You can tell Catch to run only tests with specific names or tags if you like.)

Inside a test case, you can put one or more **REQUIRE** or **CHECK** assertions. A **REQUIRE** statement checks that a certain condition holds; if it does not, it reports a test failure and stops the execution of that test case. **CHECK** is similar to require, but if the condition does not hold, it reports a test failure and keeps running the test case. Usually, you use **REQUIRE** when a failure indicates that the code is broken enough that it does not make sense to keep going with the test.

In general, when writing tests, you want to test every path through your code at least once. Here's a pretty good test for our Fibonacci function:

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch.hpp"
3
4 TEST_CASE("Fibonacci", "[Fibonacci]") // Test name and tag
5 {
6     CHECK(fibonacci(0) == 1);
7     CHECK(fibonacci(1) == 1);
8     CHECK(fibonacci(2) == 2);
9     CHECK(fibonacci(5) == 8);
10 }
```

If we compile and run this code, we get the following output:

```
fibonacci.cpp:19: FAILED:
  CHECK( fibonacci(0) == 1 )
with expansion:
  0 == 1
```

```
fibonacci.cpp:21: FAILED:
  CHECK( fibonacci(2) == 2 )
with expansion:
  1 == 2
```

```
fibonacci.cpp:22: FAILED:
  CHECK( fibonacci(5) == 8 )
with expansion:
  5 == 8
```

```
=====
test cases: 1 | 1 failed
assertions: 4 | 1 passed | 3 failed
```

Oh no! We have a bug!<sup>4</sup> In fact, it is in the **return n;** statement in our function on line 6 — it should be **return 1;** instead. If we fix that and re-run our tests, everything is kosher:

---

<sup>4</sup>Yes, you with your hand up in the back? You saw the bug before the test failed? Yes, yes, you're very clever.

All tests passed (4 assertions in 1 test case)

Now, you may notice that Catch expands the thing inside the `CHECK` function — it prints the value that `fibonacci` returns. It does this by the power of *template magic*. This magic is only so powerful.<sup>5</sup> So, if you want to write a more complex expression, you'll need to either break it into individual assertions or tell Catch to not attempt to expand it. For 'and' statements, rather than `CHECK(x && y);`, write `CHECK(x); CHECK(y);`. For 'or' statements, enclose your expression in an extra pair of parentheses: `CHECK((x || y));`. (The extra parentheses tell Catch to not attempt to expand the expression; you can do this with 'and' statements as well, but expansion is nice to have.)

There are also matching assertions `REQUIRE_FALSE` and `CHECK_FALSE` that check to make sure a statement is false, rather than true.

## Testing Exceptions

Let's modify our Fibonacci function to throw an exception if the user passes us a number that's not within the range of our function.

```
1  #include<stdexcept> // for domain_error
2  using namespace std;
3
4  /* Generate the Nth Fibonacci number */
5  int fibonacci(int n)
6  {
7      if(n < 0)
8      {
9          throw domain_error("Fibonacci not defined for negative indices");
10     }
11     else if(n <= 1)
12     {
13         return 1;
14     }
15     else
16     {
17         return fibonacci(n - 1) + fibonacci(n - 2);
18     }
19 }
```

Catch provides a number of assertions for testing whether expressions throw exceptions and what kinds of exceptions are thrown. As before, each assertion comes in a `CHECK` and a `REQUIRE` flavor.

---

<sup>5</sup>It could be made more powerful, but the downside is that the compiler errors from misuse would become even more terrifying.

- `CHECK_NOTHROW(expression)`: Asserts the expression does not throw an exception.
- `CHECK_THROWS(expression)`: Asserts the expression throws an exception. Any ol' exception will do; it just has to throw something.
- `CHECK_THROWS_AS(expression, exception_type)`: Asserts the expression throws an exception of a specified type.
- `CHECK_THROWS_WITH(expression, string)`: Asserts that the expression thrown, when converted to a string, matches the specified string.<sup>6</sup>

For example, we can check that our Fibonacci function properly verifies that its input is in the domain by testing when it throws exceptions and what exceptions it throws:

```

1 TEST_CASE("Fibonacci Domain", "[Fibonacci]")
2 {
3     CHECK_NOTHROW(fibonacci(0));
4     CHECK_NOTHROW(fibonacci(10));
5     CHECK_THROWS_AS(fibonacci(-1), domain_error);
6     CHECK_THROWS_WITH(fibonacci(-1), "Fibonacci not defined for"
7         "negative indices");
8 }
```

## Organizing Your Tests

At this point you know enough to start writing tests for functions. Before you go too hog-wild, shoving test cases every which where, let's talk about how to organize tests so they're easy to find and use.

First, we can't have our `main()` function and Catch's auto-generated `main()` in the same program. You'll need to organize your code so that you can compile your test cases without including your `main()` function. So make your program's `main()` as small as possible and have it call other functions that can be unit tested.

Second, we don't want our test code included in our actual program. A generally good pattern to follow is to divide your code into multiple files as usual, then make a separate test file for each implementation file.

For example, if we made `fibonacci.h` and `fibonacci.cpp` files for our function above, we'd also make a `test_fibonacci.cpp` file that contains our unit tests.

Third, compiling Catch's auto-generated `main()` function takes a while. This is doubly annoying because it never changes! Rather than rebuilding it all the time, we can harness the power of makefiles and incremental compilation by

---

<sup>6</sup>You can also use a string matcher in place of the string if you want to match multiple strings; we'll talk about these later in the chapter.



making a separate `test_main.cpp` file that just contains Catch's `main()`. This file looks exactly like this:

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch.hpp"
```

Then in `test_fibonacci.cpp`, we just have the following includes:

```
1 #include "fibonacci.h"
2 #include "catch.hpp"
3
4 // insert unit tests here
```

Building this code is done as follows:

```
$ g++ -c test_main.cpp           # Compile Catch's main()
$ g++ -c test_fibonacci.cpp      # Compile Fibonacci tests
$ g++ test_main.o test_fibonacci.o -o testsuite # Link testsuite
```

Now you can add new unit tests and just recompile `test_fibonacci.cpp` and re-link the test suite. Much faster! (Hint: a Makefile is *very* handy for this process!)

## Testing Classes

Testing classes works more-or-less like testing functions. You'll still write `TEST_CASEs` with various `CHECK` and `REQUIRE` statements.

However, when testing classes, it's common to need to set up a class instance to run a bunch of tests on. For example, let's suppose we have a `Vector` class with the following declaration:

```
1 template<class T>
2 class Vector
3 {
4     public:
5         // Constructor
6         Vector();
7
8         // Copy Constructor
9         Vector(const Vector<T>& v);
10
11        // Destructor
12        ~Vector();
13
14        // Add elements to the vector
15        void push_back(T v);
16
17        // Access elements of the vector
```

```

18     T& operator[](const unsigned int idx);
19
20     // Number of elements in the vector
21     unsigned int length() const;
22
23     // Capacity of the underlying array
24     unsigned int capacity() const;
25
26     private:
27         unsigned int len;
28         unsigned int cap;
29         T* array;
30 };

```

To test the `[]` operator or the copy constructor, we need to make a vector that contains elements to access or copy. You could write a bunch of test cases and duplicate the same test setup code in each, but there is a better option! Each `TEST_CASE` can be split into multiple `SECTIONS`, each of which has a name. For each section, Catch runs the test case from the beginning but only executes one section each run.

We can use this to set up a test vector once to test the constructor and accessor functions:

```

1  TEST_CASE("Vector Elements", "[vector]")
2  {
3      Vector<int> v; // Re-initialized for each section
4
5      for(int i = 0; i < 5; i++)
6      {
7          v.push_back(i);
8      }
9
10     SECTION("Elements added with push_back are accessible")
11     {
12         for(int i = 0; i < 5; i++)
13         {
14             CHECK(v[i] == i);
15         }
16     }
17
18     Vector<int> copy(v); // Only run before the sections below it
19
20     SECTION("A copied vector is identical")
21     {
22         for(int i = 0; i < v.length(); i++)
23         {

```

```

24     CHECK(v[i] == copy[i]);
25 }
26 }
27
28 SECTION("Vector copies are deep copies")
29 {
30     for(int i = 0; i < v.length(); i++)
31     {
32         v[i] = -1;
33         CHECK(v[i] != copy[i]);
34     }
35 }
36 }

```

In this example, Catch runs lines 1–16, then starts over and runs lines 1–8 and 18–26, then lines 1–8, 18, and 28–35. Since we get a fresh `v` vector for each section, the code inside each section can mutate `v` however it likes without impacting any of the other sections’ tests! Even better, we can add more setup as we go through the test case; our `copy` vector is only created for the sections that test the copy constructor.

In general, you’ll want to group related tests into one `TEST_CASE` with multiple `SECTION`s that provide more fine-grained organization of your test assertions.

## Advanced Tests

Catch provides some advanced features that come in handy when testing code that uses strings and floating point arithmetic.

When testing code that produces strings, sometimes you do not know what the entire string produced will be, but you want to check that it contains some particular substring. Catch offers a pair of assertions, `CHECK_THAT` and `REQUIRE_THAT`, that use a *matcher* to check only parts of strings. For instance, we can test that a string starts with “Dear Prudence” like so:

```
REQUIRE_THAT(my_string, StartsWith("Dear Prudence"));
```

In addition to the `StartsWith` matcher, there is an `EndsWith` matcher and a `Contains` matcher. These matchers can be combined using logical operators; for example:

```
REQUIRE_THAT(my_string, StartsWith("Dear Prudence") &&
    !Contains("Sincerely"));
```

These matchers can also be used in the `THROWS_WITH` assertions!

Testing floating point code presents a challenge because floating point operations may have some round-off that prevents exact equality checks from working. Catch provides a class named `Approx` that performs approximate equality

checks; for instance, `CHECK(PI == Approx(3.14));`. By default, the comparison can be off by 0.001%, but you can change this! For a more precise comparison, you can set the `epsilon` to a smaller percentage: `CHECK(PI = Approx(3.1415).epsilon(0.0001));`.

## Code Coverage

Unit tests are most valuable when all your important code is tested. You can check this by hand, but that's no fun, especially on a big codebase. Fortunately, there are tools to check for you! We'll use `gcov` to generate code coverage reports for us.

First, a few words about template classes. `gcov` only gives meaningful results if each function in the template class is actually generated one place or another. Fortunately, you can explicitly ask the compiler to instantiate a copy of every template class function. For example, at the top of our `test_vector.cpp` file, we would put

```
template class Vector<int>;
```

so that `gcov` properly reports if we forget to test any member functions of our `Vector` class.

As with Address Sanitizer and `gprof`, `gcov` requires some compile-time instrumentation. Compile your test files with the `--coverage` flag.

Once you have compiled your tests, execute them as normal. In addition to running your tests, your executable will also produce a number of files ending in `.gcda` and `.gcno`. These are data files for `gcov`. They're binary, so opening them in a text editor will not be particularly enlightening. To get meaningful coverage statistics, you run `gcov` and give it a list of `.cpp` files whose behavior you want to see. (Generally this will be all your test `.cpp` files.)

There are a couple of flags that you definitely want to use for `gcov`:

- `-m`: De-mangle C++ names. For whatever reason, C++ names are mangled by the compiler and look very odd unless you tell programs to demangle them.
- `-r`: Only generate information for files in the current directory and sub-directories. This prevents you from generating coverage information about stuff in the standard library, which I hope you are not attempting to unit test.

So, for our `Vector` example above, we would do something like the following:

```
$ g++ --coverage -c test_vector.cpp
$ g++ --coverage -c test_main.cpp
$ g++ --coverage test_main.o test_vector.o -o testsuite
```

```
$ testsuite
=====
All tests passed (25 assertions in 2 test cases)
```

```
$ gcov -mr test_vector.cpp
File 'test_vector.cpp'
Lines executed:100.00% of 39
Creating 'test_vector.cpp.gcov'
```

```
File 'catch.hpp'
Lines executed:62.50% of 64
Creating 'catch.hpp.gcov'
```

```
File 'vector.hpp'
Lines executed:100.00% of 34
Creating 'vector.hpp.gcov'
```

(Hint the second: *makefiles are very nice for automating this process.*)

When looking at `gcov`'s output, you are mostly concerned that all the code you set out to test is being executed. In this case, that means we are looking for `vector.hpp` to be 100% executed, and it is!

If you are curious, you can open `vector.hpp.gcov` and see the number of times each line is executed. Here's a snippet for `Vector`'s constructor:

```
16:      3:Vector<T>::Vector()
    -:      4:{
16:      5:   cap = 4;
16:      6:   len = 0;
16:      7:   array = new T[cap];
16:      8:}
```

The numbers in the left margin are the number of times each line is executed. If a line isn't executed, you will see `####` in the left column instead. This makes it easy to spot code that isn't covered by your tests!

## Writing Quality Unit Tests

Alright, now that you know how to write unit tests, let's talk about the philosophy of writing good unit tests. Unit tests are most useful if you write them as you go, rather than writing a big chunk of code and then writing tests. Some people prefer to write their tests first, then write the code needed to make the tests pass. Others write the code first and then the tests. Either way, testing as you go will help you think through how your program ought to work and help you spot bugs that come from changing or refactoring your code.

Tests should be small and designed to test the smallest amount of functionality possible — typically a single function, or a single feature of a function. Typically, more code means more bugs; we do not want our unit tests to be complex enough to introduce their own bugs! Tests should be obviously correct as much as is possible. Start by testing basic functions, such as accessors and mutators. Once those have been tested, you can use them in more complex functionality tests; if one of those tests fails, you know that the bug does lie somewhere other than your basic functions.

If you come across a bug in your program, write a unit test that reproduces it, then fix your code so that the test passes. This way, you won't accidentally reintroduce the bug later on!

When writing tests, think about the different ways your code can be executed. Consider your `if` statements and loops — how might each be executed or not? What side effects does your code have? Does it modify member variables? Write to a file? Thorough tests cover as many of these possible execution paths as is feasible. Think about your preconditions and postconditions and write tests that verify your interface conforms to its documentation! Test your edge cases, not just the “happy path” that normal execution would take.

Finally, a word on code coverage. Like any metric, code coverage is not a perfect measure of your tests' quality. Practically speaking, 100% code coverage is difficult to achieve; it is better to have 90% coverage and well thought out tests than 100% coverage with tests that don't reflect how your code will actually be used.

## Questions

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

1. In your own words, what is the goal of unit testing? How do you know you have written good tests?
2. What is the difference between the `CHECK` and `REQUIRE` test assertions?
3. Write the test assertion you would use if you wanted to assert that a call to `frobnicate()` throws an exception of type `bad_joke` and to bail out of the test case if it does not.

## Quick Reference

### Assertions

Boolean:

- `CHECK/REQUIRE`: Assert that an expression is true
- `CHECK_FALSE/REQUIRE_FALSE`: Assert that an expression is false

Exceptions:

- `CHECK_NOTHROW/REQUIRE_NOTHROW`: Assert that no exception is thrown
- `CHECK_THROWS/REQUIRE_THROWS`: Assert that an exception is thrown
- `CHECK_THROWS_AS/REQUIRE_THROWS_AS`: Assert that an exception of a specific type is thrown
- `CHECK_THROWS_WITH/REQUIRE_THROWS_WITH`: Assert that an exception with a specific error string is thrown

String Matchers:

- `CHECK_THAT/REQUIRE_THAT`: Assert that a string satisfies a match expression
- `StartsWith`: Verify that a string starts with a given string
- `EndsWith`: Verify that a string ends with a given string
- `Contains`: Verify that a string contains a given string

Floating Point:

- `Approx`: Perform approximate floating point comparison (by default up to 0.001% error)
- `epsilon`: Set the precision of the comparison

### Coverage

- Compile your tests with the `--coverage` flag
- Run your test suite executable
- Run `gcov -mr [ .cpp files ]` with all `.cpp` files in your project to compute code coverage

## Further Reading

- [Catch Tutorial](#)
- [Catch Manual](#)
- [Floating Point Comparisons](#)
- [Matcher Expressions](#)



- [Catch GitHub Repository](#)
- [gcov Manual](#)



## Chapter 9

# 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!

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

---

<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.

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

- Explore editor features
- Try build (compilation) features
- Experiment with built-in debugging tools

## Walkthrough

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. 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 **codeblocks**. You may find it useful to keep a couple of PuTTY windows open while you work.

We'll be using the Code::Blocks IDE. To open this IDE, run the **codeblocks** command in a shell. Hopefully, after a while of thinking about it, the system will deign to display a window somewhat like the following:

---

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

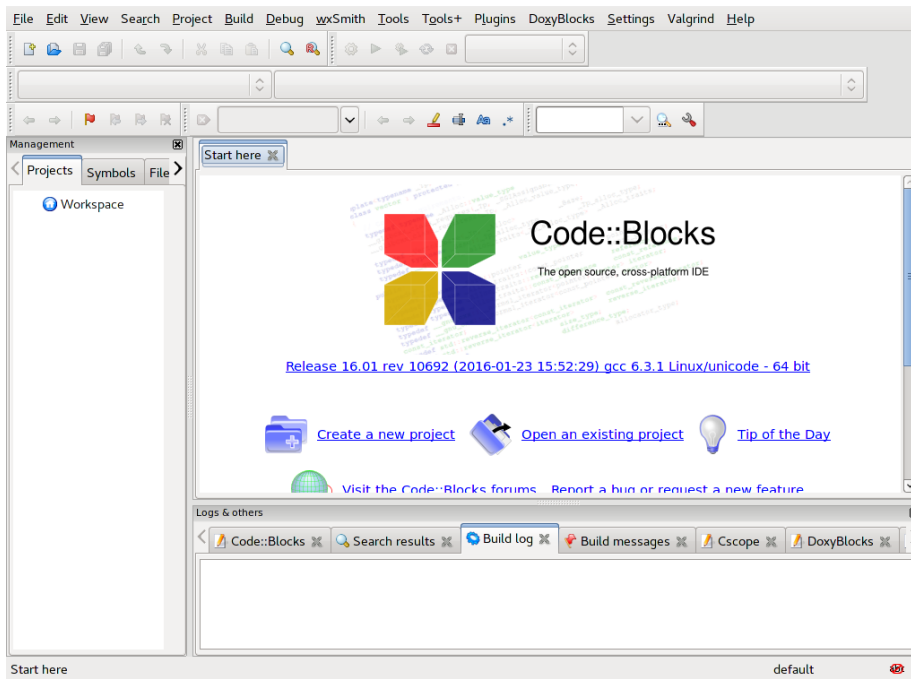


Figure 9.1: Code::Blocks

Look! Buttons! Windows! Tabs!

## Getting started

Code::Blocks organizes your code into projects. It's best to make one project per program. Each project has an associated project file that ends in **.cbp**; this file keeps track of the code files in your project, how to build them, and various other sundries.

To create a new project, click any of the numerous “new project” buttons. Code::Blocks knows how to make a bunch of different kinds of software, but if you want to make a plain 'ol terminal application in C++, you'll want to choose the “Console application” option:

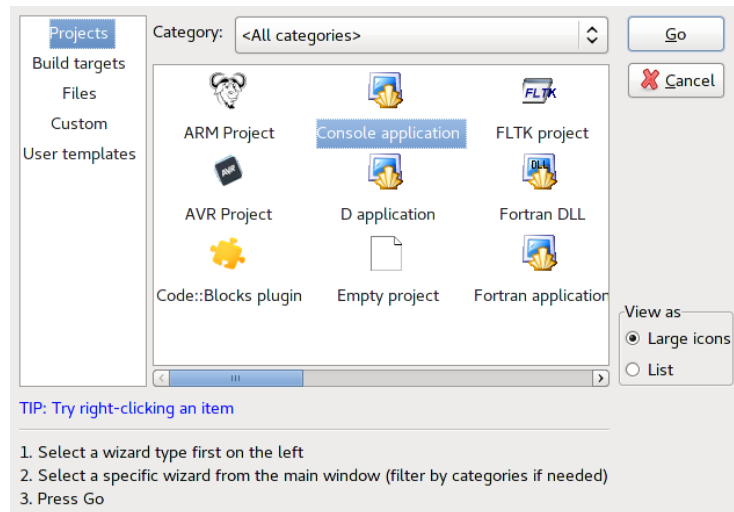


Figure 9.2: New project dialog

Code::Blocks will ask you for a name and a location to store the project in. It will also ask you about compilers and targets — the default values are fine; we’ll talk about them later.

Once you’ve created a project, you can add new files to it via any of the “new” buttons. Or, you may have some files that you’d like to add to your project — maybe you have some starter code for an assignment, or you’re making a project for something you were already working on. To do this, first put your files where you want them (usually in the project directory — or you can make the project directory be the location of your files), then right-click the project and click “Add files”:

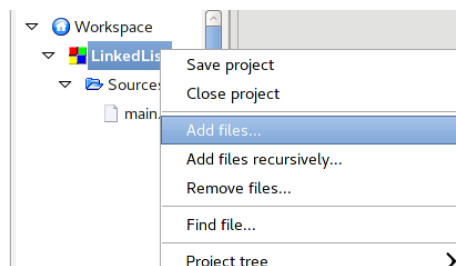


Figure 9.3: Adding files

## Building

Once you’ve created a project and put some files in it, you’ll probably want to compile your code! Below you can see the various compilation options Code::Blocks offers — you can build your code and run it in one step!

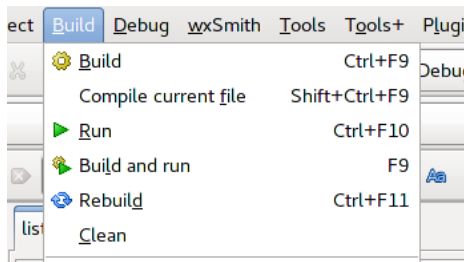


Figure 9.4: The Build menu

When you click “build”, Code::Blocks only compiles the files that have changed since you last built the project. Typically this works fine, but sometimes you’ll want to recompile everything; for this task, the “rebuild” button is what you want.

Projects have one or more “build targets” — by default, a “debug” target and a “release” target. These targets primarily change the flags passed to the compiler; the debug target usually builds faster and includes debugging symbols, but the code produced by the release target is more efficient. If you are planning to use a debugger or Valgrind, you should use the “debug” target.

You can see your project’s target configurations in the project build options (right-click on the project and choose “Build options”). For now, the options Code::Blocks chooses for you should be sufficient; later on in this book we’ll discuss some additional flags that you might want to add.

## Navigation and Editing

One big draw of using an IDE is the out-of-the-box editing and navigation features. Since Code::Blocks is primarily designed for writing C++ programs, it offers special features for editing C++ code.

If you type a class instance name followed by a `.`, Code::Blocks will show a list of potential member functions and variables you may want:

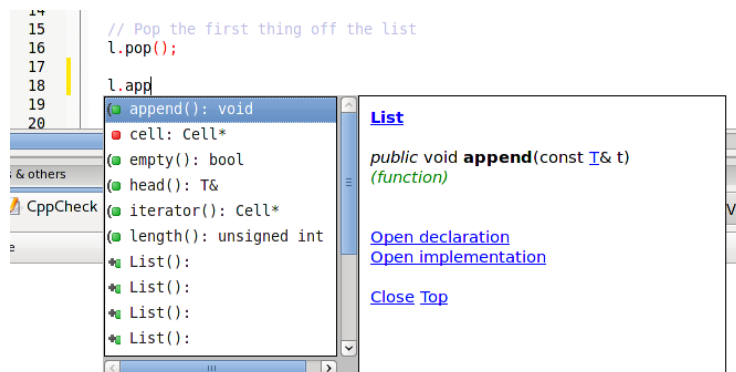


Figure 9.5: Class member completion list

You can type the first few letters of the desired method or variable to narrow the list down, then press `Tab` to have Code::Blocks complete the name for you.

Once you’ve typed out a complete function name, Code::Blocks will show you the type signature, so you know what parameters it takes:

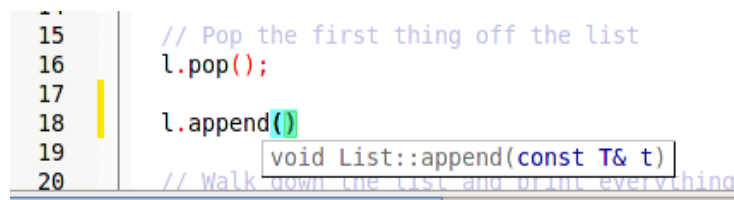


Figure 9.6: Type signature hint

You can also access the “Open Declaration” and “Open Implementation” features of the tab completion window by right-clicking on any function or variable name:



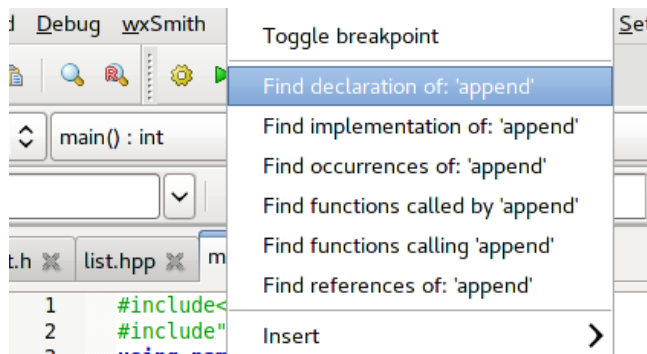


Figure 9.7: The result of right-clicking on `append`

In the same menu, under the “Code refactoring” submenu, you can rename a variable or function across all files in your project:

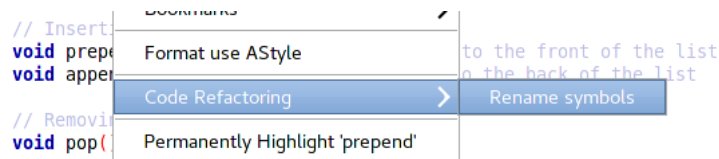


Figure 9.8: Renaming a symbol

If you are navigating through a large file, you can jump to function implementations with the “Code Completion” toolbar. First, select the scope your function is declared in (typically either `<global>` or the name of the class your function is a member of):

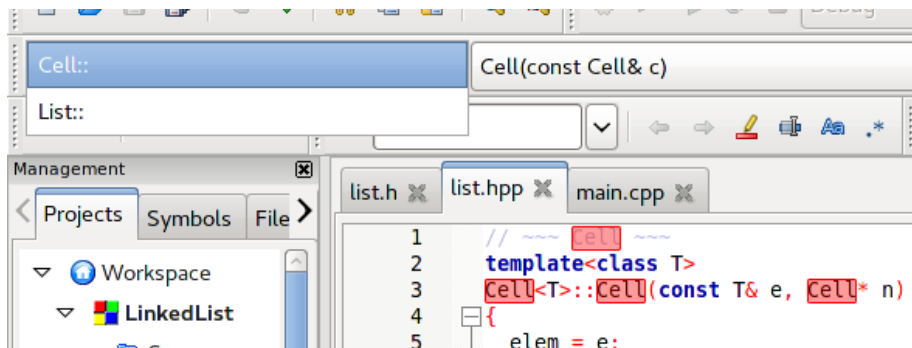


Figure 9.9: Selecting a scope

Then select your function from the next drop-down, and Code::Blocks will jump you to that point in the file!

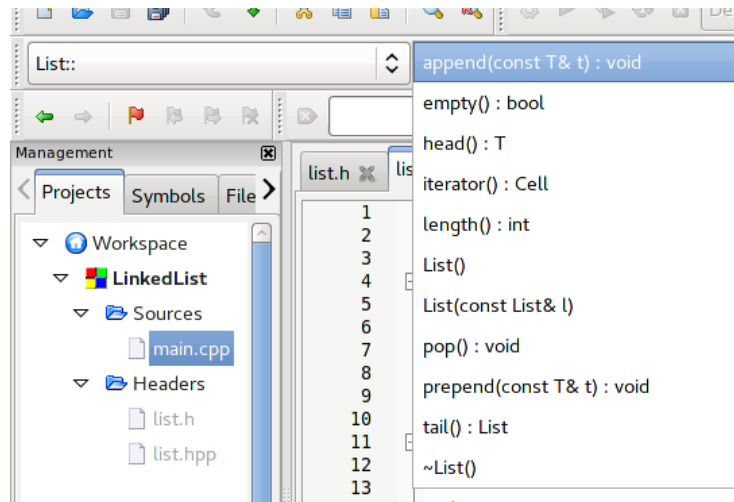


Figure 9.10: Selecting a function

You can collapse (or 'fold') code between braces by clicking the - button in the margin by the line number. This is particularly handy when working with long functions that are hard to fit all on one screen.

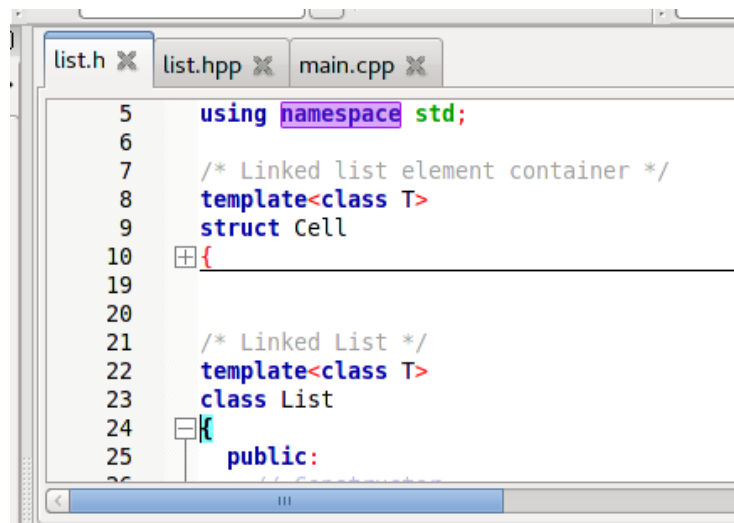


Figure 9.11: Folding the definition of the `Cell` class

## Debugging

Code::Blocks integrates with the GNU debugger (**gdb**) to make debugging code easier for you. We'll talk more about **gdb** in a future chapter, but for now we'll introduce the features Code::Blocks integrates with.

In short, a debugger allows you to pause the execution of your program, inspect variables, and step through your code line-by-line. You can tell the debugger to stop execution at a specific line by setting a breakpoint on that line: click to the right of the line number, and a little stop sign will appear:

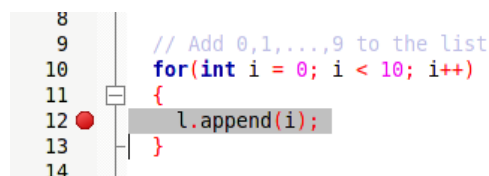


Figure 9.12: Setting a breakpoint

Once you've set one or more breakpoints, you can click the red arrow on the debugger toolbar to run your program and have it stop at that breakpoint.



Figure 9.13: The debugger toolbar

Other buttons on the toolbar allow you to step your code by line, or by CPU instruction.

When you are running a program in the debugger, you can right-click on variables and “watch” them—this will open a new window that displays that variable and its value, along with function argument values and local variable values. Once you’re watching a variable, you can edit its value while the program is running!

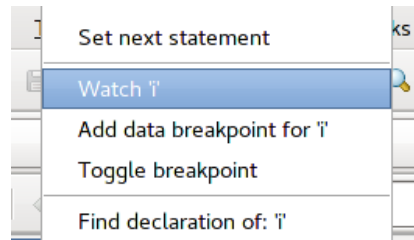


Figure 9.14: Watching a new variable

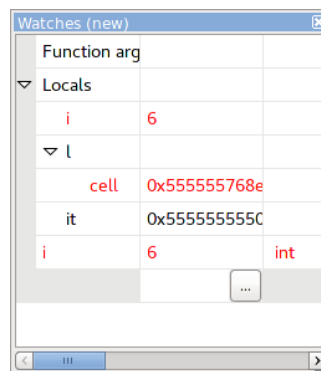


Figure 9.15: The “Watches” window

Code::Blocks also opens a tab at the bottom of the window that allows you to send commands to `gdb`, just as you would when using `gdb` from the command line. While Code::Blocks doesn’t add any new functionality to `gdb`, it does make some common debugging tasks a lot easier!

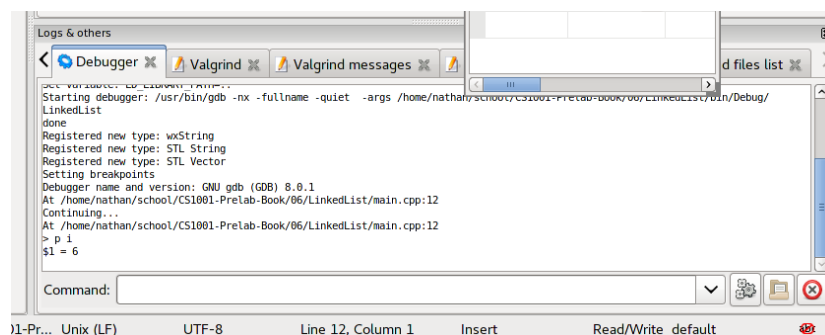


Figure 9.16: The `gdb` tab

## Questions

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








1. In your own words, list three features of an IDE.
2. Think back to chapter 1. How do IDEs and text editors differ?

## Quick Reference









### 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>/cs1585/lab09/` (or whatever you call your repository.)

### Building with Code::Blocks

-  – Build and run
-  +  – Build
-  +  – Run
- Enable `-Wall` in  

### Writing code with Code::Blocks

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

## Further Reading

- 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 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 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

---

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

I 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. Stembleburgiss.”

“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 or instruction-by-instruction to figure out which parts are fast and which are slow.

Fortunately, you don’t have to sit with your 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 spends its time, 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`, `call-grind`, 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
```

---

<sup>2</sup>Input/Output

```
# If we wanted to time a hello world program...
$ 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...
0.00	21.69	0.00	1	0.00	0.00	__static_initializati...

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 [...
		0.00	0.00	1/1	__static_initialization_and_de...
-----					
		0.00	0.00	1/1	_GLOBAL__sub_I_Z9new_func...
[12]	0.0	0.00	0.00	1	__static_initialization_and_de...
-----					

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 (checking every two seconds of film). It gives you a *good* idea of how much movie time we spend looking at Mr. St. Rumpsterfrabble, 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<sup>3</sup>, 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<sup>4</sup>. This time when we compile it, we need to pass the **-g** flag instead of **-pg**<sup>5</sup>. Then we'll run our program through **callgrind** as shown.

```
# -g this time! NOT -pg
$ 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!
# Run `ls` to check the name of your callgrind output file.
$ 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/...
```

---

<sup>3</sup>Since we're talking about programs... not movies.

<sup>4</sup>Otherwise we'll be waiting all day for **callgrind** to do its thing.

<sup>5</sup>Uh... T.D.A. 2 is rated PG for Pretty Good.

```

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.

## Profiling memory usage with **massif**

The tools we've seen so far (`time`, `gprof`, and `callgrind`) are all concerned with the speed of our program. They help us figure out how we're spending our time as we run our programs. However, there are profilers that are concerned with resources other than time.

**massif**, for example, is another profiling tool built with Valgrind. Instead of looking at program timing, **massif** looks at memory usage. The information that it gives you can be useful for identifying code that hogs memory unnecessarily.

Since we're talking about memory, let's use a memory-oriented example.

```

#include <iostream>
#include <unistd.h>

using namespace std;

int main()
{
    // Allocate an array of arrays of chars
    char** strings = new char*[10];
    for(int i = 0; i < 10; i++)
    {
        strings[i] = new char[1000];
    }

    // Sleep (i.e., waste -- do nothing) for one second.
    sleep(1);

    // Deallocate everything
    for(int i = 0; i < 10; i++)
    {
        delete[] strings[i];
    }
    delete[] strings;

    return 0;
}

```

When we compile this program, we'll need to use the same flags that we used when compiling for `callgrind`. Then we can run the compiled program through `massif`. It's worth mentioning that `massif`, by default, is only concerned about watching memory on the heap. If you need to see information about the stack, you can use the `--stack=yes` flag.

```

# -g this time! NOT -pg
$ g++ -g -o main main.cpp
$ valgrind --tool=massif --time-unit=B ./main

```

Like `callgrind`, we'll get a numbered output file with a name like `massif.out.NNNN` (where `NNNN` is the process ID). `ms_print` is the tool we use to get information from our data file.

```

# Be careful! The process ID (5029 in this case) will change!
# Run `ls` to check the name of your massif output file.
$ ms_print massif.out.5029

```

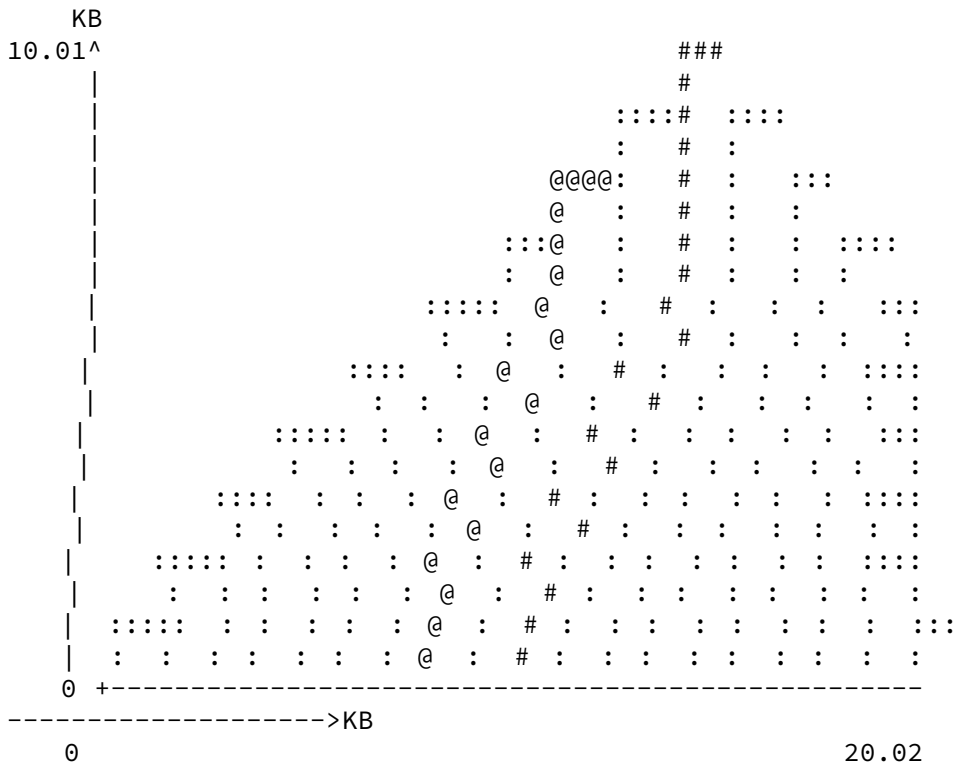


## Reading the Memory Graph

The output from `ms_print` starts off with a neat graph of memory usage drawn in ASCII. The Y-axis is the total amount of memory that your program has allocated on the heap. As you can see, our program allocates more and more memory (1000 bytes at a time actually) until it reaches its peak at 10,080 bytes. After that, we start deallocating memory until it reaches back to zero.

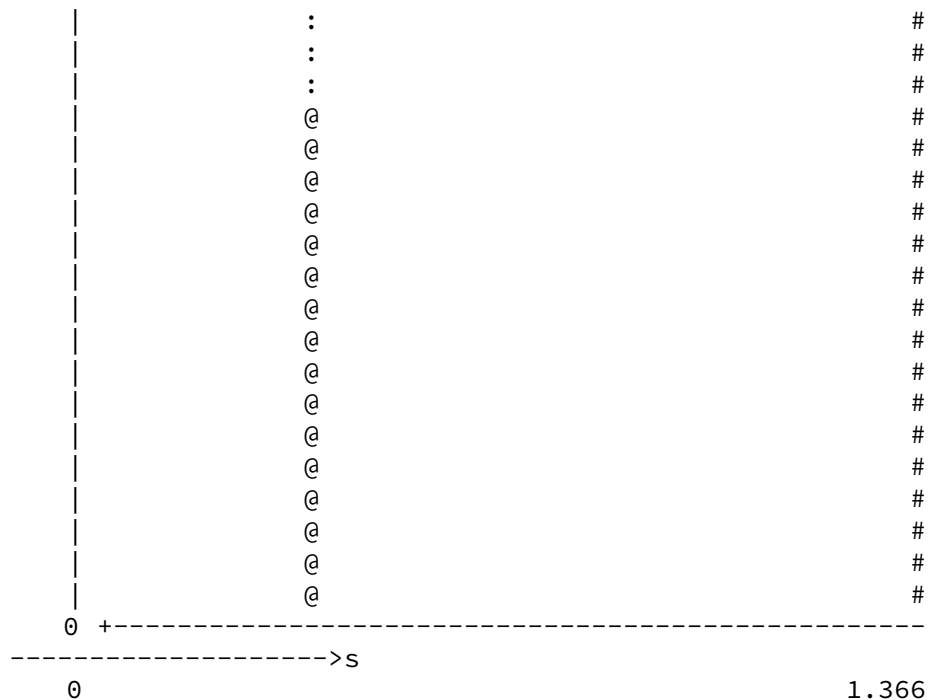
So what's with the @ and #?

- Columns drawn using `:` are plain snapshots. They show how much is allocated on the heap at that time.
- Columns drawn using `@` are detailed snapshots. `massif` includes extra data in its tabular output for detailed snapshots. We'll have a look at detailed snapshot here in a minute.
- Columns drawn using `#` indicate the peak memory usage.



The X-axis is time. You're probably wondering "why is time measured in bytes instead of seconds?" Well, actually, you can totally use seconds. Here's what that looks like:





Looks ridiculous, right?

Our program is *very* fast. We allocate every single array within one millisecond. That means that even if we use milliseconds as our `--time-unit`, all of the allocations happen (apparently) at the same time. Additionally, we've got that `sleep(1)` in there, so our plot is stretched *way* out. By using bytes as our time unit, the amount of memory allocated is a function of our memory operations (allocations and deallocations). This makes our plot look a lot saner.

### Reading the Snapshot Table

The memory graph is handy for getting a quick idea of how memory is allocated by your program. To give you more information about the memory snapshots, `ms_print` includes details in a table below the graph.

n	time(B)	total(B)	useful-heap(B)	extra-
heap(B)	stacks(B)			
0	0	0	0	0
1	88	88	80	8

2	1,104	1,104	1,080	24	0
3	2,120	2,120	2,080	40	0
4	3,136	3,136	3,080	56	0
5	4,152	4,152	4,080	72	0

Let's look at the **useful-heap** column.

- We start with zero bytes allocated. If we haven't allocated anything yet, the count should be zero. Makes sense.
- Our first allocation requires 80 bytes. Given that our first allocation is for an array of 10 8-byte pointers (since this was run on a 64-bit computer), this seems sane.
- The following allocations each cause our **useful-heap** to grow by 1,000 bytes at a time. If you look at the for-loop in our program, every loop allocates 1,000 chars at 1 byte a piece.

If you add the **useful-heap** and the **extra-heap** together, you achieve the **total** amount of memory allocated at that point. Since we're only allocating memory for the first half of the program, the time matches the **total** exactly. For a discussion of the **extra-heap** column, refer to **massif**'s documentation in the Further Reading section. The short version is that those bytes are "administrative."

n	time(B)	total(B)	useful-heap(B)	extra-heap(B)	
heap(B)	stacks(B)				
10	9,232	9,232	9,080	152	0
11	10,248	10,248	10,080	168	0
12	10,248	10,248	10,080	168	0
13	11,264	9,232	9,080	152	0
14	12,280	8,216	8,080	136	0

Once we reach the peak (snapshot #12), we can see that the time (in bytes) continues to increase, but the total starts to go down. With each deallocation of **N** bytes, the total will decrease by **N** bytes, but the time will increase by **N** bytes.

In addition to normal snapshots, you'll see a small number of detailed snapshots. Each detailed snapshot shows you what the heap looks like. In the detailed snapshot below, you'll see that as of snapshot 9, the memory we allocated on line 12 totals 8,000 bytes. This makes sense, since we've run line 12 a total of 8 times by the time we reach snapshot 9.

```
98.34% (8,080B) (heap allocation functions) malloc/new/new[], -
-alloc-fns, etc.
->97.37% (8,000B) 0x4007DA: main (main.cpp:12)
```

```
|
->00.97% (80B) in 1+ places, all below ms_print's threshold (01.00%)
Toward the end of our program, we see a final detailed snapshot at snapshot 22.
90.91% (80B) (heap allocation functions) malloc/new/new[], -
-alloc-fns, etc.
->90.91% (80B) 0x4007AE: main (main.cpp:9)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

By the end, we've deallocated everything except our `char**`. The only memory remaining on the heap is 80 bytes. Snapshot 23 shows our final memory situation.

```
-----
-----
n          time(B)          total(B)  useful-heap(B) extra-
heap(B)    stacks(B)
-----
-----
23         20,496           0           0           0           0
```

All of our heap memory has been deallocated <sup>6</sup>, and we've allocated and deallocated a total of 20,496 bytes. In other words, if we summed the memory that we `new`'d and `delete`'d we would have 20,496 bytes.

---

<sup>6</sup>Yay!

## 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?
2. Briefly explain, in your own words, the difference between the way `gprof` (`g++ -pg`) and `callgrind` profile programs.
3. Briefly summarize the output you get when you use `time`, `gprof`, and `callgrind_annotate` to profile a program.
4. Briefly explain, in your own words, why we would use bytes as our unit of time in `massif` instead of seconds (or milliseconds).

## Quick Reference

### **time**

- Just run `time <program> [<args...>]!`
- Reading `time`'s output:
  - **Real**: The wall-clock or total time the program took to run.
  - **User**: The time the program (and libraries) spent executing CPU instructions.
  - **System**: The time the program spent waiting on system calls (usually I/O).

### **gprof**

- First, You must compile with the `-pg` flag.
- Then, run your program like normal. It will create a file named `gmon.out`.
- `gprof <program>` reads `gmon.out` generated by `<program>` and displays profiling statistics!

### **Flat Profiles**

- A **flat profile** is an overview of function usage.
- Time measures are based on sampling 100 times/second.
- Function call counts are exact.

### **Call Graphs**

- A **call graph** is a listing of which functions called each other.
- The line with the index entry is the function under consideration.
- Lines above that are functions that called this function.
- Lines below that are functions that this function called.

### **callgrind**

- Compile with the `-g` flag.
- Running `callgrind` will create a file named `callgrind.out.NNNN`.
  - `valgrind --tool=callgrind ./program`
- The `callgrind_annotate` tool reads `callgrind.out` files and prints some statistics on your program.
  - `callgrind_annotate --auto=yes callgrind.out.NNNN`
- Callgrind counts instructions executed, not time spent.

- The annotated source shows the number of instruction executions a specific line caused.
- Function calls are annotated on the right with the number of times they are called.

## Recursion and callgrind

- Recursion can confuse both `gprof` and `callgrind`.
- The `--separate-recs=N` option to Valgrind separates function calls up to N deep.
- The `--separate-callers=N` option to Valgrind separates functions depending on which function called them.
- In general, when you have recursion, the call graph and call counts may be wrong, but the instruction count will be correct.

## massif

- Compile with the `-g` flag
- Running `massif` will create a file named `massif.out.NNNN`.
  - `valgrind --tool=massif --time-unit=B ./program`
- To get information on stack memory usage as well, include `--stack=yes` after `--time-unit=B`.
- The `ms_print` tool reads `massif.out` files and prints statistics for you.
  - `ms_print massif.out.NNNN`
- Snapshots: `massif` takes a snapshot of the heap on every allocation and deallocation.
  - Most snapshots are **plain**. They record only how much heap was allocated.
  - Every 10th snapshot is **detailed**. These record where memory was allocated in the program.
  - A detailed snapshot is also taken at peak memory usage.
- The graph: Memory allocated vs. time. Time can be measured in milliseconds, instructions, or bytes allocated.
  - Colons (:) indicate plain snapshots, at-signs (@) indicate detailed snapshots, and pound-signs (#) indicate the peak snapshot.
- The table shows the snapshot number, time, total memory allocated, currently-allocated memory, and extra allocated memory.
- The table also shows the allocation tree from each detailed snapshot.

## Further Reading

### **gprof**

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

### **callgrind**

- [Callgrind's Manual](#)

### **massif**

- [Massif's Manual](#)





## Chapter 11

# Regular Expressions



Figure 11.1: <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 each part of 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

### Character Classes

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.

### Repetition

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).

## Groups

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

What we really need is a way to group parts of the match together. Fortunately, you can do this with `()`s! You can then apply modifiers (like `+`) to the group as a whole. `\d{5}(-\d{4})?` matches any ZIP code with an optional +4 extension — either it matches a 5 digit ZIP and none of `-\d{4}` or a 5 digit ZIP and all of `-\d{4}`.

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`. The backreference there says "Match another of whatever `(ab|cd)` matched".

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.

## Anchors

By default, regular expressions match a substring anywhere in the string. So if you have the regex `a+b+` and the string "cccaabbbbb", 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.

## Greedy and Non-greedy matching

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 characters, followed by a period) is fine, as long as there is just one sentence per line, like so:

```
This is one sentence.
And this is another.
Maybe we'll be daring and include a third.
```

But if there’s more than one sentence on a line, this regex will match all of them, because `.` matches “.”! For example, if we were to run it on the following file:

```
This file has three sentences. But only two lines.
I guess technically the second sentence isn't one.
We lied to you--or did we?
```

The first match would be “This file has three sentences. But only two lines.”, which contains two sentences.<sup>5</sup> If you want it to match one and only one sentence, you have to tell the `.\+` to match only as much as needed, so `.\+?\.`

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

## grep

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

---

<sup>4</sup>POLITICS!

<sup>5</sup>English professors, please look away.

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 input) 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.

If you don’t enable Perl-style regexes, **grep** requires you to escape special characters to get the special meaning.<sup>6</sup> In other words, **a+** matches “a+”, whereas **a\+** matches one or more “a”s. If you want to use the syntax shown in the previous section, you’ll want to pass the **-P** flag to **grep**.

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

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

Here’s a more practical example of where **grep** comes in handy:

```
$ grep -i todo big_homework_file.cpp
// TODO: reticulate the splay tree
// ToDo copy this part from stackoverflow
/* It would be nice todo some more consistent comment formatting */
/* TODO remove completed todos from code */
```

---

<sup>6</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.

## sed

**grep** is great and all but it just prints 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. Speaking of which, what is **sed** good for?

- Renaming variables, functions, etc. in code.
- Making changes to text file databases (CSV files, etc.).
- Impressing your friends with your ability to write arcane commands.

## Questions

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

1. Suppose, for the sake of simplicity,<sup>7</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”. (**Hint:** you’ll need to use some of the I/O redirection stuff you learned in Chapter 2.)

---

<sup>7</sup>In practice, email addresses can have all sorts of things in them! Like spaces! Or quotes!

## Quick Reference

### Regex

Character classes:

- `.`: 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).

Repetition:

- `{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.

Common commands:

- `/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 12

# Graphical User Interfaces with Qt

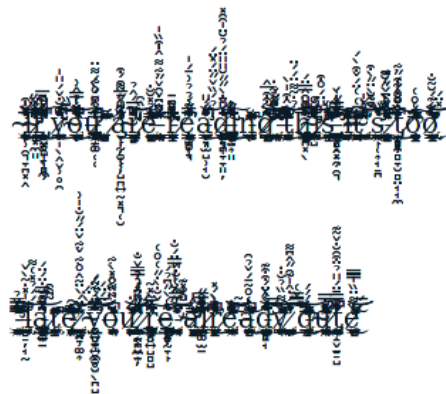
### Motivation

Close your eyes.

Wait, no. Open them! Look around you!

Unless you're a 1337 cyber hacker, you probably don't see just text in terminals. If you do, you should really see a doctor about that.

Ok, this next part might be a little weird, but trust us, it's necessary for your understanding. You may feel some slight discomfort as you read the next bit:



You lift your hands to sip your coffee, but instead of hands, you see a large mouse cursor. Disturbed but undeterred (coffee is important, after all), you move your cursor towards your mug, but you pause once the cursor is on the mug. How do you sip coffee with a cursor? A piece of paper with the word ‘Coffee’ written on it suddenly appears over the mug. You move your ~~hand~~ cursor slightly and the paper vanishes as mysteriously as it appeared.

You take a moment for some introspection, and discover you know how to *left-click* and *right-click*. Oh. Of course.

You left-click the coffee mug. A dashed outline appears around it. Obviously.

Maybe right-clicking works? You try, and a board appears below the mug. It looks like it’s made out of a cafeteria tray. A list of words is written on it: ‘Tip Over’, ‘Hurl through Window’, ‘Upend’... and, at last, ‘Sip’. You finally sip your coffee. It’s gotten cold. Darn.

Well, time for a change of scenery. You lift your cursor to the heavens, where another strip of cafeteria tray material hovers. You select ‘View’, then ‘Outdoors’ from the board that appears.

Without warning, you’re dumped into the middle of a field. After the dizziness subsides, you look around. Where are you? What is happening? Why can’t things just go back to being text on a screen?

You would scream, but you have no menu entry for that.

## Takeaways

- Experiment with Qt to build Graphical User Interfaces
- Get a taste of event-driven programming
- Learn a bit about how large libraries and projects are organized
- Appreciate the simplicity of programming Command Line Interface applications.

## Walkthrough

It’s worth mentioning before we get to deep that you should spend some time looking through your starter repository’s example code in addition to this walkthrough. Qt requires a non-negligible amount of code to get anything interesting done. We’ve included a few examples here, as well as a discussion of some conceptual stuff, but you’ll need to spend time looking at the example code to see how all the pieces fit together.

## Building Qt Projects

To make your life a little easier, the Qt framework includes a preprocessor that generates some C++ code for you. You still have to write C++, of course. It's just a little less.

Qt's preprocessor is called the Meta Object Compiler (**moc**). Fortunately, you don't have to work with it (or **g++**) directly, since Qt can generate a **Makefile** for you!

So...you don't need to run **moc**, **g++**, or even make your own **Makefile**...so what *do* you have to do?

Qt uses project files, which end in **.pro**, to determine how to build your projects. You can generate a new **.pro** file for a Qt project by running the command **qmake -project**.

Qt is a big library, so to speed up compile times not everything gets added in all at first. We're going to use the 'widgets' part of the library, so we'll need to open up our **.pro** file and add the following line:

```
QT += widgets
```

That tells **qmake** to include the widgets library files when it generates the makefile. Once we've added this line, we can just **#include <QtWidgets>** and use all the widgets Qt provides to our hearts' content.

Once you have a **.pro** file, you can run **qmake** to generate a **Makefile**. Then you can run **make** to compile everything together!

```
# Run this to create a project file
```

```
$ qmake -project
```

```
# Edit the file to enable widgets
```

```
$ echo "QT += widgets" >> project_name.pro
```

```
# Run this once to create your Makefile
```

```
$ qmake
```

```
# ...then this whenever you want to recompile
```

```
$ make
```

## Parts of a Qt Application

Whenever you write a Qt application, you will instantiate **one** instance of **QApplication**. The **QApplication** object represents your entire application. It allows you to work with the application as a whole. The only thing we'll be

using it for is shutting our application down. Although you instantiate the **QApplication** in `main()`, you can access it throughout your program through the `qApp` pointer (as long as you `#include <QApplication>`).

So that's nice, right? A **QApplication** is an object that represents your entire application. Not the windows, no buttons...it's the *whole* thing.

All of those clickable things that we all love to click: those are called **widgets**. If you want a useful application, you can't work with just a **QApplication**. You need to spice it up with some widgets. Just pepper it with buttons.

BAM!

So let's consider this application:

```
1  #include <QApplication>
2  #include <QTextEdit>
3
4  int main(int argc, char** argv)
5  {
6      QApplication app(argc,argv);
7
8      QTextEdit te;
9      te.setWindowTitle("Not Vim");
10     te.show();
11
12     return app.exec();
13 }
```

First, we instantiate our application. That's just dandy!

Next, we create a single **QTextEdit** widget, which is a big box you can type text in. We instantiate the widget just like we would instantiate any ol' C++ object. Since it's the only widget in our application, it gets its own window. We'll go ahead and set a window title for it. We then ask Qt to display our text editor window using the `.show()` member function.

So we've got an **application** set up with a text editing **widget**, and we've asked Qt to show it. In order to see our application in action, we need to ask it to run using `app.exec()`.



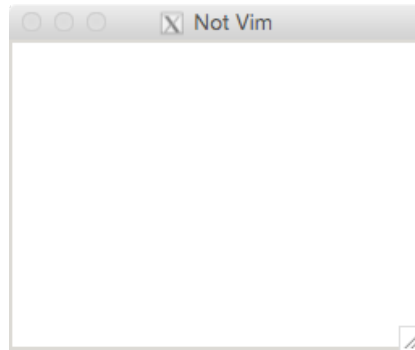


Figure 12.1: Our first app! Boy howdy. That sure ain't vim.

## Laying out your App

Alright!

That sure was an app. It leaves a lot to be desired, though.

We can construct more interesting applications by being smart about our widgets. We can add widgets to other widgets to create complex applications. To position a bunch of widgets on screen, we use a **layout** widget.

For example, let's say we want to put a quit button above our text editor (in the same window of course). We can use a `QVBoxLayout` to **vertically** (hence the V) stack our widgets.

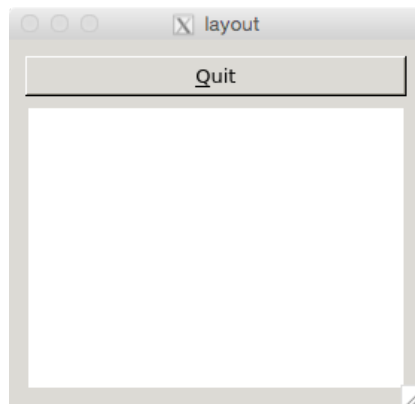


Figure 12.2: We can organize the widgets in our application by creating a vertical stack. First we add the button (to the top) then we add the text editor (beneath the button).

```

1 // #includes left out for the sake of brevity
2 int main(int argc, char** argv)
3 {
4     QApplication app(argc,argv);
5
6     QTextEdit* te = new QTextEdit;
7     QPushButton* quit = new QPushButton("&Quit");
8
9     QVBoxLayout* layout = new QVBoxLayout;
10    layout->addWidget(quit);
11    layout->addWidget(te);
12
13    QWidget window;
14    window.setLayout(layout);
15
16    window.show();
17
18    return app.exec();
19 }

```

Now let's walk through this biz:

1. We first create our app, like normal.
2. We create a couple of widgets: our text editing widget and our quit button.
3. We create a vertical layout and add our quit button followed by our text editor. This essentially tells Qt we want to create a vertical stack in our window: the quit button on top, and the editor beneath.
4. We then make a window and add our layout to it.

A couple of odd things to note:

1. On line 7, the `&` tells Qt to set up a keyboard shortcut, `Alt+q`, that 'presses' the button.<sup>1</sup>
2. You may notice that on lines 6, 7, and 9, we allocate memory with `new` but never call `delete`. Unlike typical C++ objects, Qt objects are written so that they clean up their children when they are destructed. In this case, our `QTextEdit` and `QPushButton` are added as children to our `QVBoxLayout` object, and that layout object is added as a child to the `QWidget` created on line 13. This means that as long as we clean up that `QWidget`, all our other objects will get cleaned up automatically! (And, since that `QWidget` is a stack-allocated variable, it gets destructed whenever `main()` returns.)<sup>2</sup>

The rest is similar to the last example. With layouts, we have the ability to specify how we want our widgets organized on screen. In addition to vertical layouts

<sup>1</sup>See <http://doc.qt.io/qt-5/qshortcut.html#mnemonic> for more about this.

<sup>2</sup>See <http://doc.qt.io/qt-5/objecttrees.html> for more about this.

there are horizontal layouts (`QBoxLayout`), grid layouts (`QGridLayout`), and a handful of others.

So, that's dandy...but our quit button doesn't actually do anything! To make our buttons work, we need to talk about Signals and Slots.

## Signals and Slots

Qt is **event-driven**. It waits for stuff to happen. Once something happens, it reacts to it. It's up to you to decide how it reacts to stuff that happens.

### What is a signal?

If you press a button, it emits a **signal**. "HOLY GUACAMOLE" the button says. "DANG DANG GOSH I DONE BEEN PRESSED!" That's about it.

Lots of things can emit signals:

- Buttons
- Text fields
- Other widgets
- You can even emit your own signals!

All of that is dandy, but if no one is listening to you, what's the point?

### What's is a slot?

A slot is a big ol' ear.

Just the biggest ear you can imagine. All goofy and just a-waitin' to hear something. The thing is — big goofy ear ain't just listenin' for any ol' thing. It's listening for a **specific** signal.

You can create a slot to listen to any signal. A slot is basically just a function. When the signal it is listening for is emitted, the slot (function) is executed.

Once you've got a signal to listen to and a slot to listen for it, you can `connect()` them.

### Connecting a Signal to a Slot

Let's talk about how we get that big ear to listen to that screaming button.

The `connect()` function connects a signal to a slot. It takes four parameters:

- The object that is sending the signal (**Source**)

- The signal sent by that object (button pressed down, button lifted up, etc)
- The object that is receiving the signal (**Destination**)
- The slot that is receiving the signal (whatever the slot function is called)

```

1 // #includes left out for the sake of brevity
2 int main(int argc, char** argv)
3 {
4     QApplication app(argc,argv);
5
6     QTextEdit* te = new QTextEdit;
7     QPushButton* quit = new QPushButton("&Quit");
8
9     QObject::connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
10
11     QVBoxLayout* layout = new QVBoxLayout;
12     layout->addWidget(quit);
13     layout->addWidget(te);
14
15     QWidget window;
16     window.setLayout(layout);
17
18     window.show();
19
20     return app.exec();
21 }

```

The above example is almost the same as the previous except for this line:

```
QObject::connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
```

This line `connect()`s the quit button to the `quit()` member function of `qApp`. So what's going on here? Let's take this parameter-by-parameter.

1. `quit` – The object that's emitting (screaming) the signal of interest is our quit button. The object pointer is `quit`.
2. `SIGNAL(clicked())` – The signal that `quit` is emitting is the `clicked()` signal to indicate it's been clicked. (The `SIGNAL(...)` part of it is for `moc`. Don't worry about that bit.)
3. `qApp` – The object that's listening for the emitted signal. We want it to know when our `quit` button has been pressed.
4. `SLOT(quit())` – The slot (function) that should run whenever the signal is emitted. Remember that the `qApp` object represents our whole application. We want to call its `quit()` member function whenever our `quit` button is pressed. (Again, the `SLOT(...)` part of it is for `moc`, so don't worry about that.)

`connect()` is what allows a slot to run whenever a signal is emitted. The slots are often member functions of Qt objects. Sometimes they're Qt widgets,

sometimes they're not (such as is the case with `qApp`). We'll see some more examples of signals in the next section.

## Menus and Toolbars

By now, we know how to create widgets and organize them on screen. There are a couple of additional things we might want to add: menus and toolbars.

There's a special Qt class that comes with these things for free. `QMainWindow` is a class for making standard applications with menus and toolbars. The `QMainWindow` has one big ol' widget that goes in the middle of the window and fills that window. `setCentralWidget()` is a member function of `QMainWindow` that sets this widget.

To create your menus (File, Edit, whatever you want) you need to add them to your `QMainWindow`. It has a `menuBar()` member function that returns a pointer to the menubar, which you can use to add new menus. Similarly, there is an `addToolBar()` member function that creates a new toolbars.

Since toolbar buttons and menu items do the same thing (fire an event when clicked), Qt represents them both as `QActions`. You can add the same `QAction` to both a menu and a toolbar — so there's less code repetition as well! `QActions` have a `triggered()` signal that is emitted whenever their menu item or button is clicked.

Let's do an example of this, building out a more complex notepad application. For this, we are going to need to define our own slots for saving and opening files. To do that, we'll need to create our own class that inherits from `QMainWindow`. This is how you'll build most complex applications in Qt.

Let's talk about our code architecture for a minute:

1. Our `main()` function is just going to create a `QApplication` and create and `show()` a `Notepad`.
2. `Notepad` will have member variables for essential widgets so we can refer to them throughout the program as needed.
3. The constructor for `Notepad` will do most of the setup we've been doing in `main()` so far — create various widgets, add them to layouts, and connect signals to slots.
4. We will create slot functions in `Notepad` that will let the user choose files to open or save and do the necessary work to open or save those files.

Ok, ok, enough talk, let's see some code. First, `main()`:

```
1  #include "notepad.h"
2  #include <QApplication>
3  int main(int argc, char** argv)
4  {
5      QApplication app(argc,argv);
```

```

6
7     Notepad n;
8     n.show();
9
10    return app.exec();
11 }

```

Alright, nothing terribly fancy here. Let's take a look at `notepad.h`:

```

1  #pragma once
2  #include <QtWidgets>
3
4  class Notepad : public QMainWindow
5  {
6      Q_OBJECT // Tells moc to include slots 'n signals magic
7      public:
8          Notepad();
9          virtual ~Notepad() {}
10
11     private slots: // 'slots' is a new keyword moc understands
12         void quit();
13         void open();
14         void save();
15
16     private:
17         QTextEdit* textEdit;
18         QAction* quitAction;
19         QAction* openAction;
20         QAction* saveAction;
21
22         QMenu* fileMenu;
23         QToolBar* fileToolbar;
24 };

```

Notice that we have 4 functions to implement: the `Notepad` constructor and our `quit()`, `open()`, and `save()` slots. Our private member variables contain the `QTextEdit` for the text, three `QActions` for the editor actions we'll have, and objects for our file menu and toolbar.

First, let's see how these are all connected together by having a look at `Notepad`'s constructor, defined in `notepad.cpp`:

```

1  #include "notepad.h"
2
3  Notepad::Notepad()
4  {
5      // Make our text edit box the central widget
6      textEdit = new QTextEdit;

```

```

7   setCentralWidget(textEdit);
8
9   // Connect each action's triggered() signal to
10  // the appropriate slot on this Notepad
11  openAction = new QAction("&Open", this);
12  connect(openAction, SIGNAL(triggered()), this, SLOT(open()));
13
14  saveAction = new QAction("&Save", this);
15  connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));
16
17  quitAction = new QAction("&Quit", this);
18  connect(quitAction, SIGNAL(triggered()), this, SLOT(quit()));
19
20  // Add actions to file menu
21  fileMenu = menuBar()->addMenu("&File");
22  fileMenu->addAction(openAction);
23  fileMenu->addAction(saveAction);
24  fileMenu->addAction(quitAction);
25
26  // Add actions to toolbar
27  fileToolbar = addToolBar("File");
28  fileToolbar->addAction(openAction);
29  fileToolbar->addAction(saveAction);
30  fileToolbar->addAction(quitAction);
31 }

```

Now let's look at how our slots are implemented. These functions will be called when their associated toolbar button or menu item is clicked. First, the `quit()` slot. We'll add a confirmation dialog so our users don't lose their unsaved work if they accidentally click the quit button:

```

33 void Notepad::quit()
34 {
35     QMessageBox messageBox;
36     messageBox.setWindowTitle("Quit?");
37     messageBox.setText("Do you want to quit?");
38     messageBox.setStandardButtons(
39         QMessageBox::Yes | QMessageBox::No
40     );
41     messageBox.setDefaultButton(QMessageBox::No);
42     if(messageBox.exec() == QMessageBox::Yes)
43     {
44         qApp->quit();
45     }
46 }

```

`QMessageBox` does what it says — makes a message box that asks the user a

question. Calling `exec()` causes the message box to display; the return value of that function is the button that was clicked. If the user clicks 'Yes', then we quit the application!

Next, let's have a look at the `open()` function:

```
48 void Notepad::open()
49 {
50     QString fileName = QFileDialog::getOpenFileName(
51         this, // Parent object
52         "Open File", // Dialog Title
53         "", // Directory
54         "Text Files (*.txt);;C++ Files (*.cpp *.h)" // File types
55     );
56
57     if(fileName != "") // Empty string indicates user canceled
58     {
59         QFile file(fileName); // Like a std::ifstream, but cuter
60         if(!file.open(QIODevice::ReadOnly))
61         {
62             QMessageBox::critical( // a message box, but more serious
63                 this, // Parent
64                 "Error", // Dialog Title
65                 "Could not open file" // Dialog Text
66             );
67         }
68     }
69     else
70     {
71         QTextStream in(&file);
72         textEdit->setText(in.readAll());
73         file.close();
74     }
75 }
```

Here we ask the user to select a file to open, try to open it, and if we succeed, read the contents of the file and put them into the `textEdit`. (Note that right now, this overwrites whatever was in the `textEdit` without warning!) If we can't open the file, we show them an error message.

The `save()` slot is very similar to the `open()` slot. The primary difference is that we write the `textEdit`'s contents to an output file stream instead:

```
75 void Notepad::save()
76 {
77     QString fileName = QFileDialog::getSaveFileName(
78         this, // Parent
79         "Save File", // Dialog Title
```



```

80         "", // Directory
81         "Text Files (*.txt);;C++ Files (*.cpp *.h)" // File Types
82     );
83
84     if(fileName != "")
85     {
86         QFile file(fileName);
87         if(!file.open(QIODevice::WriteOnly))
88         {
89             QMessageBox::critical(
90                 this, // Parent
91                 "Error", // Dialog Title
92                 "Could not write to file" // Dialog Text
93             );
94         }
95         else
96         {
97             QTextStream out(&file);
98             out << textEdit->toPlainText();
99             file.close();
100         }
101     }
102 }

```

It is worth mentioning that these slots can also be called just like normal functions! And, as you may have anticipated, you can declare public slots as well that other objects can connect to signals.

## Slots and signals that carry data

Signals can carry data when they are emitted! Any data a signal carries gets passed as parameters to the slot function. You can only `connect()` a signal to a slot if the signal and slot take the same parameters. In the examples we've done so far, none of our signals have carried data, and none of our slots have taken parameters. Thus, we could use them together.

Let's walk through an example where we create a signal and slot that carry some information along with them. Right now, our notepad doesn't have a very interesting window title. Let's put the name of the program and the filename the user is editing in the title!

To do this, we're going to create a new signal named `useFile`, and a new slot named `setTitle`. Whenever we open or save a file, we will emit the `useFile` signal. That signal will carry the filename as a `QString`.

We define our signal in the `Notepad` class definition in `notepad.h`:

```
signals:
    void useFile(QString fileName);
```

This tells Qt that **Notepad** is capable of hootin' and hollerin' about files. **signals** is another keyword understood by moc. Unlike with slots, you do **not** implement the signal function — you just have to declare it.

We'll also want to add our new slot in **Notepad**'s class definition;

```
private slots:
    void setTitle(QString fileName);
```

First, let's implement our slot:

```
void Notepad::setTitle(QString fileName)
{
    setWindowTitle("Not Vim (" + fileName + ")");
}
```

Next, we need to connect our signal to our freshly-written slot. We'll do this in **Notepad**'s constructor, right next to **connect()**s for our **QActions**.

```
connect(this, SIGNAL(useFile(QString)),
        this, SLOT(setTitle(QString)));
```

When connecting signals and slots that carry data, you must specify the types of the parameters of the signal and slot. Do **not** put parameter names here (i.e., **fileName**) — that will confuse moc, unfortunately. You just need the parameter types. In this case, our signal carries just one piece of data, a **QString**.

Finally, we need to emit our signal whenever we open or save a file! Emitting a signal works almost exactly like calling a function, but you prefix the call with the keyword **emit**. So, in both **open()** and **save()**, immediately after the **file.close()** line, we'll put the following line:

```
emit useFile(fileName);
```

Here we **emit** the signal named **useFile**; the data that **useFile** should carry along with it is the contents of the **fileName** variable. Now whenever we open a file or save a file, the filename will appear in the title!

By now, you know the basics of arranging widgets on screen and event-driven programming with signals and slots. There are a lot of different widgets out there! If you want to build more featureful GUI programs, you should definitely have a look through what Qt has to offer. Also, have a look through the documentation on the widgets we've used in this chapter. They sport a lot of features we don't have space to talk about here. Qt's documentation is a little daunting, but quite detailed!

## Questions

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

1. Briefly explain (in your own words) the relationship between the `moc`, `qmake`, and `make`. That is, what are they each used for, and how do they relate?
2. Briefly explain (in your own words) what signals and slots are.
3. Let's say we have an object called `zoidberg` that emits a signal called `powerful_stench()`. Whenever `zoidberg` emits `powerful_stench()`, we want another object (`everyone_nearby`) to execute a slot (`barf()`). Write a `connect()` call that connects the `powerful_stench()` signal emitted by `zoidberg` to the `barf()` slot of `everyone_nearby`.

## Quick Reference

### qmake

- **qmake** is a utility that manages Qt projects and generates Makefiles automatically.
- The **-project** flag tells Qt to generate a project file (ends in **.pro**) that configures the Makefile.
- **qmake** will generate a Makefile
- If you already have a **.pro** file, all you have to do to build a Qt project: run **qmake**, then **make**

### Signals and Slots

- If you don't connect a signal to a slot, the slot's not going to run.
- You can also call slots like regular old member functions.
- If you have a typo in your **connect()** call, the **moc** may not catch it.

## Further Reading

- [A list of all Qt classes](#), with links to documentation for each
- [Qt Examples and Tutorials](#)
- [qmake documentation](#)
- [Qt Development Tools](#), including an IDE!

## Chapter 13

# Typesetting with L<sup>A</sup>T<sub>E</sub>X

### Motivation

Close your eyes. You are [Donald Knuth](#).

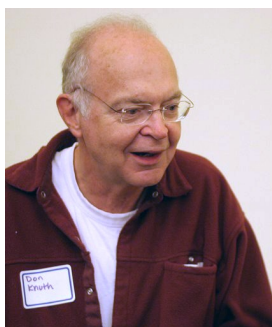


Figure 13.1: You, aka Donald Knuth

The year is 1977, and you have just finished hand-writing the final draft of the second edition of the second volume of *The Art of Computer Programming*. You send the draft off to the publisher for typesetting, and the proof that comes back is just horrendous. The letters are blurry, the myriad subscripts and superscripts unreadable smudges, and the spacing and justification is all out of whack. Your first edition proofs were nothing like this! You call them up to see what happened. “Sorry,” they say, “we switched to a modern photographic typesetting system; that’s what you get now.”

This will not do. “I’ve spent 15 years writing these books, but if they are going

to look awful I don't want to write any more.”<sup>1</sup> The first edition was typeset on a Monotype typesetter machine, which mechanically calculated the spacing required for justified lines and cast individual bits of type that were exactly the right size needed. What you need is a digital typesetting system that's equally as powerful! Writing one should only take a summer, so you set to work.

In 1989 the first “finished” version of T<sub>E</sub>X was completed.<sup>2</sup>

(Okay, you can stop being Don Knuth now.)

Since T<sub>E</sub>X is relatively low-level, Leslie Lamport (a fellow known for wearing silly hats and also verification of distributed systems) wrote a bunch of useful macros that took care of a lot of the day-to-day typesetting stuff. This collection came to be known as L<sup>A</sup>T<sub>E</sub>X — Lamport's T<sub>E</sub>X.

So, what is L<sup>A</sup>T<sub>E</sub>X good for?<sup>3</sup>

Do you think writing should come with more compiler errors? Do you ever wish HTML was more arcane and confusing? Are word processors too easy to use? Do you think PDF is the one true document format?

Do you want to make your research on the Area 51 coverup to look professional?

Then you should learn L<sup>A</sup>T<sub>E</sub>X!

## Takeaways

- Learn how to structure L<sup>A</sup>T<sub>E</sub>X documents
- Math is beautiful and easy to typeset
- T<sub>E</sub>X is definitely something designed in the 70's

## Walkthrough

L<sup>A</sup>T<sub>E</sub>X is a language for marking up text. You write plain ASCII text in a `.tex` file (say, `bob.tex`), then run it through the `pdflatex` command<sup>4</sup> (so, `pdflatex bob.tex`). `pdflatex`, as its name implies, spits out a PDF file that you can open with your favorite PDF viewer. Everyone has a favorite PDF viewer, right?

First, a bit about syntax. L<sup>A</sup>T<sub>E</sub>X commands begin with a `\`. So, to write the fancy L<sup>A</sup>T<sub>E</sub>X logo, you'd type `\LaTeX{}`. Required arguments are surrounded

---

<sup>1</sup>Paraphrased from *Digital Typography*, p. 5.

<sup>2</sup>See [this essay](#) for more about the history of mathematical typesetting.

<sup>3</sup>Besides having a nifty command for printing out its own logo, of course.

<sup>4</sup>Sometimes twice, and sometimes thrice! `pdflatex` is a one-pass parser and, well, some things just can't be done in one pass. Yes, this is poor design.

with `{}`s; optional arguments are listed between `[]`s. For example: `\command[option1, option2]{required argument 1}{required argument 2}`.

L<sup>A</sup>T<sub>E</sub>X also has ‘environments’, which are used to wrap larger chunks of text. These begin with `\begin{environment}` and end with `\end{environment}`.

## Document Classes

The first thing in a `.tex` file is a document class command: `\documentclass{classname}`. Several document classes come built-in, including the following:

- **article**: Used for conference and journal articles and typical classroom reports.
- **report**: Used for small books or longer reports that span several chapters.
- **book**: Used to make fritatta.
- **beamer**: Used to make slides for presentations.

The document class defines the look of the document, as well as what commands are available to structure your document.

## Document Structure

Between the `\documentclass` command and the rest of the document goes any package setup you desire. To include the ‘hyperref’ package, which lets you make clickable links, write `\usepackage{hyperref}`. Some packages may require additional configuration; consult the documentation to see how to use them.

Before starting the actual document, you should set the title, date, and author of the work by using the `\title`, `\date`, and `\author` commands.

The document itself is written between `\begin{document}` and `\end{document}`.

To insert the title and author, use the `\maketitle` command.

Here’s a short example:

```
1 \documentclass{article}
2
3 \title{Do Lizards Run The World?}
4 \author{Nathan Jarus}
5 \date{\today}
6
7 \begin{document}
```

```
8 \maketitle
9 \end{document}
```

Now for some actual content. Your document can be split into sections, subsections, subsubsections, paragraphs, and sub-paragraphs. (Some document classes, such as report and book, include a ‘chapter’ command that is above ‘section’ in the hierarchy.) The commands for each of these takes one argument: a title for the given part of the document. The document class controls the numbering and appearance of the titles for you.

Continuing our example:

```
1 \documentclass{article}
2
3 \title{Do Lizards Run The World?}
4 \author{Nathan Jarus}
5 \date{\today}
6
7 \begin{document}
8 \maketitle
9 \section{Introduction}
10
11 \section{Methodology}
12 \subsection{A Perpetual Energy Source}
13 \subsection{A Radio Beacon for the Pyramids of Giza}
14 \subsection{Plans for First Contact}
15
16 \section{Results}
17 \subsection{Physics Dislikes Me}
18 \subsubsection{Physicists don't want the truth}
19 \subsubsection{This foil hat is perfectly comfortable, thanks}
20
21 \section{Conclusion: Perhaps the real aliens
22         are the friends we made along the way}
23 \end{document}
```



# Do Lizards Run The World?

Nathan Jarus

November 15, 2017

## 1 Introduction

## 2 Methodology

### 2.1 A Perpetual Energy Source

### 2.2 A Radio Beacon for the Pyramids of Giza

### 2.3 Plans for First Contact

## 3 Results

### 3.1 Physics Dislikes Me

#### 3.1.1 Physicists dont want the truth

#### 3.1.2 This foil hat is perfectly comfortable, thanks

## 4 Conclusion: Perhaps the real aliens are the friends we made along the way

Figure 13.2: The document structure example, rendered

## Formatting Text

For the most part, you can write text as you normally would in a word processor. To make a new paragraph, put two newlines in a row:

1 This is a sentence.

2 This is a second.

3

4 And here is a brand new paragraph!

# \$ % ^ & \_ { } ` ~ and \ are reserved characters. (Of note: % starts a one-line comment, much as // does in C++.)

You can write them using the escape sequences `\#` `\$` `\%` `\^{}` `\&` `\_` `\{` `\}` `\`{}` `\~{}` and `\textbackslash{}`.<sup>5</sup>

Opening quotes are written with the ``` character and close quotes with the `'` character. So, ``text in double quotes'` renders like “text in double quotes”.

The age-old standbys of bold, italic, and underlined text are present in  $\text{\LaTeX}$  as well:

- `\textbf{bold face text here}`
- `\textit{italicized text here}`
- `\underline{this text is underlined}`

You can also put text in a monospaced font: `\texttt{I am a robot}` renders like **I am a robot**.

Last but not least, URLs and hyperlinks can be added. For this, you need the `hyperref` package, which provides several commands. The `\url` command prints a URL in monospaced font; you use it like so: `\url{http://www.funroll-loops.info/}`. The `\href` command lets you add hyperlinks: `\href{http://url.com}{displayed, underlined text}` makes the text clickable and provides a visual indication that there’s a link to click on.

Hyperref’s color scheme is not everyone’s favorite. You can configure this; for instance, to make hyperlinks black with an underline, put the following right after `\usepackage{hyperref}`:

```
\hypersetup{colorlinks=false,
  allbordercolors={0 0 0},
  pdfborderstyle={/S/U/W 1}
}
```

## Lists

You can make both bulleted and numbered lists in  $\text{\LaTeX}$ . The former are called ‘itemized lists’, while the latter are ‘enumerated lists’.

Here’s an example itemized list:

```
\begin{itemize}
  \item Itemize makes a bulleted list.
  \item Every item in the list starts with
    the item command.
  \item You can make multiline items\\
    by putting a linebreak in them.
\end{itemize}
```

---

<sup>5</sup>Usually `\^` and `\~` are used to write accents on letters; for instance, `\~n` renders like ñ.

And a numbered list:

```
\begin{enumerate}
  \item Enumerate numbers each item.
  \item Otherwise it's exactly the same as itemize.

  \item You can also nest lists!
  \item Just start a new itemize or enumerate in a list:
    \begin{enumerate}
      \item Enumerates will change numbering style.
      \item Itemizes will use a different glyph.
    \end{enumerate}
  \item Once you're done, you can keep adding new
    list items to the original list.
\end{enumerate}
```

## Math

Math typesetting is L<sup>A</sup>T<sub>E</sub>X's pride and joy. We could easily write a whole book chapter just on how to format various arcane equations. Rather than doing that, we'll just show you some examples of common usage.

Math can be placed in a sentence by putting math markup between \$ signs: \$f(x) = 2x\$ shows up like  $f(x) = 2x$ . For bigger, more important equations, you can put them in an 'equation' environment. (Your document class will probably number these equations.) For instance,

```
\begin{equation}
  f(x) = 2x + 4
\end{equation}
```

renders as

$$f(x) = 2x + 4 \tag{13.1}$$

Subscripts and superscripts can be stacked to your heart's content:

- `x_n` produces  $x_n$
- `x^2` produces  $x^2$
- `x_n^k` produces  $x_n^k$
- `x_{n^k}` produces  $x_{n^k}$

Set notation is a breeze: `\forall n \in \{1,2,3,4\}` appears as  $\forall n \in \{1,2,3,4\}$ .

Summations (as well as products and integrals) can be done using subscripts and superscripts: `\sum_{i=0}^{\infty} \frac{1}{3^i} = \frac{3}{2}` renders to

$$\sum_{i=0}^{\infty} \frac{1}{3^i} = \frac{3}{2} \quad (13.2)$$

Fractions can be done with the `\frac{ }{ }` command. You can adjust the size of parentheses, brackets, and such with the `\big`, `\Big`, `\large`, and `\Large` commands.

`\Big(\frac{1}{3}\Big)^k = \frac{1}{3^k}` renders as

$$\left(\frac{1}{3}\right)^k = \frac{1}{3^k} \quad (13.3)$$

For more math commands, consult [the wikibook on L<sup>A</sup>T<sub>E</sub>X's math mode](#).

## Figures

Figures go in the ‘figure’ environment, which positions them and lets you give them a caption. L<sup>A</sup>T<sub>E</sub>X will place the figure in a spot on the page that makes sense, usually at the top or the bottom (but you can tweak this manually if you like). The `\caption` command sets a caption for the image. You can center the image on the page with the `\centering` command.

The ‘graphicx’ package allows you to include pictures (`.png`, `.jpg`, `.eps`, or `.pdf`) with the `\includegraphics` command. Here is an example:

```

1 \documentclass{article}
2 \usepackage{graphicx}
3
4 \begin{document}
5
6 \begin{figure}[h] % Place 'here' instead of at top/bottom
7 \caption{4-corner simultaneous 4-day time cube}
8 \centering % Center the image
9
10 % width=\textwidth makes the image the width of the text
11 \includegraphics[width=\textwidth]{timecube}
12
13 \end{figure}
14 \end{document}
```

## Tables

Much like ‘figure’, the ‘table’ environment lets you caption and position tables. The actual table is made using the ‘tabular’ environment. Its syntax is a little

strange. Fortunately, there exist many websites, including <http://truben.no/table/>, which generate the tabular markup for you.

Here is an example table:

```
1 \begin{table}
2   \begin{tabular}{l|l|l}
3     ~           & Heading    & Another Heading \\ \hline
4     Sandwiches &  $x > 2$     & Very Tasty      \\
5     Ice Cream  &  $x = 5^5$     & Excellent       \\
6   \end{tabular}
7 \end{table}
```

We've barely scratched the surface of what  $\text{\LaTeX}$  can do — there's a reason it's the standard tool for writing papers in most scientific and engineering fields. It also has bibliography management tools, packages that can syntax highlight code, packages that you can use to draw gorgeous figures... whatever document feature your heart desires, there's probably at least one package out there for it. Go forth and make beautiful documents!

## Questions

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

1. How would you write the equation  $y^2 + x^2 = 1$  in the middle of a sentence?

2. What is the environment used for numbered lists?

3. What does the `\centering` command do?

## Quick Reference

## Further Reading

- The [L<sup>A</sup>T<sub>E</sub>X Wikibook](#) is a very handy reference.
- The [T<sub>E</sub>X StackExchange](#) has a lot of tips on doing various things and fixing various errors.
- [Detexify](#) lets you draw symbols and tells you various math commands that look similar!
- You can manage citations with [Bibtex](#).
- [CTAN](#) (the Comprehensive T<sub>E</sub>X Archive Network) has documentation on zillions of neat packages.
- [MiKTeX](#) is a Windows version of L<sup>A</sup>T<sub>E</sub>X.
- [Pandoc](#) can convert other document formats to and from L<sup>A</sup>T<sub>E</sub>X. This book is written using Pandoc!
- [TeXworks](#) is a nice cross-platform editor.
- [Gummi](#) is another good editor, but it is Linux-only.
- [LyX](#) is a WYSIWYG-ish editor based on L<sup>A</sup>T<sub>E</sub>X.





## Chapter 14

# Using C++11 and the Standard Template Library

### Motivation

Finally.

The powers that be have decided to build a STØR in your hometown. It's taken months to finish construction and stock the shelves, but now you're ready to check out their colorful housewares and famous, trendy-but-fragile furniture.

You chortle with excitement as you ride the escalator into the store. A bored sad man awaits you at the top.

"Hedge," he grunts.

After following the maze of clearly-marked paths through the furniture showcase, you marvel at your neatly compiled list of items to find in the warehouse:

- Bort
- Bort bort
- Vector
- Map
- Tuple
- Bort bort bort
- Pair

You march confidently downstairs to find your items in the housewares section. Luckily, the `Bort( Bort)+` are on display at the front. All that Bort is/are making it hard to hold the STØR map. As you struggle, you hear a friendly voice say,

“Hi-diddily-ho, customer-ino! Looks like you could use a hand!”

You stare.

“Let me guide ya around the STØR-a-roonie<sup>1</sup> here and we’ll find what you’re looking for!”

With an arm full of Bort you continue your mission in search of the other items on your list. However, now you have a mustachio’d lunatic to guide you.

## Takeaways

- Learn to use several language features offered by the C++11 Language Standard:
  - `auto` types
  - `for-each` loops
- Learn to use several data structures provided by the C++ Standard Template Library:
  - `std::vector`
  - `std::map`
  - `std::pair`
  - `std::tuple`

## Walkthrough

### Language Features

C++11 introduces a couple of language features that were not available in earlier versions of C++. This is great, but your compiler needs to know if you’re using C++11 features. Modern compilers assume you are, but if the features we’re going to see confuse the living daylights out of your version of `g++`, try passing it the `-std=c++11` flag.

```
# For example
$ g++ -std=c++11 main.cpp
```

### The `auto` keyword

The `auto` keyword asks the compiler to figure out the type of a variable for you.<sup>2</sup> `auto` is not magic: the compiler just looks at the type of the expression

---

<sup>1</sup>He’s not supposed to call it that. His managers call it “a violation of brand standards and common decency”.

<sup>2</sup>C++14 and C++17 add some more meanings for `auto`. They’re way cool, but they’re beyond the scope of this book.

on the other side of the equals sign and uses that for your variable type.<sup>3</sup> Again, **auto** is not magic: if you can't see what type the compiler is supposed to use, the compiler probably won't be able to figure it out either.

We could take this program...

```
1 int main()
2 {
3     char cstring[] = "asdf";
4     string str = string("asdf");
5
6     vector<int> thingers = vector<int>();
7     return 0;
8 }
```

... and make it a little more readable ...

```
1 int main()
2 {
3     auto cstring = "asdf";
4     auto str = string("asdf");
5
6     auto thingers = vector<int>();
7     return 0;
8 }
```

As you'll see later in this chapter, **auto** comes in handy when you work with template types in the standard template library. Those types are often pretty lengthy, and **auto** keeps your lines short. **auto** also comes in handy when you need to hold, say, the return value from a library function just long enough to pass it to another library function — you couldn't care less what the type of the value is, just as long as it gets to where it needs to go.

## The for-each loop

As the name implies, a for-each allows you to perform an action “for each item” in a container. These loops work with many types in the standard template library. Here's a quick list of things you can use a for-each loop with:

- arrays
- certain classes
  - `std::vector`
  - `std::map`
  - any other type with `begin()` and `end()` member functions

Refer to the further reading section to get an idea of what's required to get for-each loops to work with your own classes!

---

<sup>3</sup>Sorry if you were expecting Haskell or Rust levels of type deduction!

Let's have a look at an example:

```
1  int main()
2  {
3      int nums[] = {1,2,3,4,5,6};
4
5      for (auto i : nums)
6      {
7          cout << i * i << ", ";
8      }
9
10     return 0;
11 }
```

Which outputs

```
$ ./print-squares
1, 4, 9, 16, 25, 36,
```

In this example, we've created an array with six `ints` in it. We then use a for-each loop to iterate over all of those items. Upon each iteration, `i` is set to the **value** of the current element. It starts at 1, then 2, on and on until it reaches 6. For each `int`, it computes and prints the square of that value.

One caveat to be aware of is that changing the value of your loop variable (`i` in this case) won't change the thing we're iterating over (the array of `ints`). If we want to change the values in the array, we can use a reference variable instead. We could write `for(int& i : nums)`, or appending `&` to `auto` tells the compiler to infer a reference type instead:

```
1  int main()
2  {
3      int nums[] = {1,2,3,4,5,6};
4
5      // decrement every value by one
6      for (auto& i : nums)
7      {
8          i--;
9      }
10
11     for (auto i : nums)
12     {
13         cout << i * i << ", ";
14     }
15
16     return 0;
17 }
```

Output:

0, 1, 4, 9, 16, 25,

Just to be clear: for-each loops don't *have* to be used with **auto** — although they are quite a handy application for **auto**! Also, all of the containers we are about to look at in this chapter can be used with for-each loops.

## A Handful of Containers from the Standard Template Library

The Standard Template Library (STL) is vast. It has a lot of storage types for any need you can think of. More than STØR, possibly.

**std::vector** (**#include<vector>**)

“Here she is! The STØR::vector. Ain’t she a beaut’?”

You stare<sup>4</sup> at the accordion-looking contraption in his hands.

“Wanna see how it works?”

He doesn’t actually give you time to answer.

“You see, it’s empty now, so it’s totally flat. This indicator on the top says **size: 0** to let you know that. All we gotta do is **push** something into this compartment here, and whaddaya know? It says **size: 1** now. You can push as many items on the back here as ya please. It adjusts its space to accommodate whatever you put in here.”

“The only trouble is that everything in there has to be the same kind of thing. If you set it up to store Bort, that’s all it can store. It can’t store any Snell or Løjlig or anything. Only more Bort.”

He wedges it in your arms between the Bort and Bort-Bort.

“Now don’t lollygag! Let’s see what else is on your list.”

---

If that explanation didn’t sit quite right with you, a **std::vector** is like an array that resizes on the fly. Everything stored in it must be the same type, but it is templated so you can choose what the type is. You can push items on the back to grow the **vector**, and pop them off to shrink it.

```
1 vector<int> v; // An empty vector of ints
2 for (int i = 0; i < 10; i++)
3 {
4     v.push_back(i);
5 }
```

---

<sup>4</sup>This fella’s truly got you at a loss for word-iddly-ords.

```

6
7 for(int index = 0; index < v.size(); index++)
8 {
9     cout << v[index] << ' ';
10 }

```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

A word of caution: if you give the `[]` operator an index outside the bounds of the vector, it will (probably) segfault, just like a regular C++ array would. However, the `at` function is range-checked, and will throw an `out_of_range` exception if you pass it an index too large. We could rewrite line 11 above as `cout << v.at(index) << endl`; if we wanted to ensure we don't walk off the end of the vector.

STL vectors also feature *iterators*, which are objects that help iterate over the elements of a vector. They're used sort of like a pointer: you use `*` to 'dereference' the value the iterator is at, and you can use `++` and `--` to move forward or backward through the elements of the vector.

So, we could write the above loop using iterators like so:

```

13 for (vector<int>::iterator iter = v.begin(); iter != v.end(); ++iter)
14 {
15     cout << *iter << endl;
16 }
17
18 // For-each loops use iterators under the hood
19 for (int value : v)
20 {
21     cout << value << endl; // No need to "dereference"!
22 }

```

(`auto` is quite handy for replacing `vector<int>::iterator` on line 13 above.)

Iterators are also used to erase elements from or insert elements into the vector. To remove the third element from `v`:

```

23 // Adding 2 to begin() gives an iterator at the 3rd item
24 v.erase(v.begin() + 2);
25 // Now v = [0, 1, 3, 4, 5, 6, 7, 8, 9]

```

Items are inserted *before* the item the iterator points at. We can put 2 back into our vector like so:

```

26 v.insert(v.begin() + 2, 2);

```

Using `erase` and `insert` saves you the effort of writing a loop every time you want to slide something out of or into the middle of a vector.

**std::tuple (#include<tuple>)**

“The STØR::tuple isn’t for everyone.”

He holds what looks like a shoe box.

“You can take an put whatever you want in here, but whatever kind of thing that is...that’s all it can store. In fact, there are a bunch more rules about what it can and can’t store. It’s a pretty advanced little piece of container technology, really. I’ll leave you to read the manual about that.”

“Now what’s cool about this fella is you can attach a bunch of them together! We can snap three together and it’s a three-tuple. And each compartment can store *different kinds of things!* We could put a Bort here and a Sbibble here and even a Blagoonga here on the end!”

“Now, unlike that STØR::vector, you can’t change the size once it’s got stuff in it. This three-tuple has to have three items. We can’t add to it, and we can’t take away. It’s stuck storing this many items, and it’s stuck storing these kinds of items.”

He sets the tuple on top of your vast pile of items.

---

The **std::tuple** is kind of like a struct. It has a set number of items which can have different types, but the types of those items cannot change. Unlike a struct, you cannot name the members of a tuple — you just access them by their index. They come in handy primarily in situations where you want something like a struct, but don’t want to go to the effort of building a struct for a one-off use.

Here’s a tuple that stores a char, a float, and int, and a string!

```
1 tuple<char, float, int, string> thing('x', 2.5, 4, "ABC");
```

If you want to get the items out of a tuple, you need to use the **get<N>()** function. The template parameter to **get** (**N**) is the index of the item you want. If you want the first item in a tuple named **tup**, you’d use **get<0>(tup)** to get it.

We can print out that above tuple like so:

```
2 cout << get<0>(thing) << ' '
3     << get<1>(thing) << ' '
4     << get<2>(thing) << ' '
5     << get<3>(thing) << endl;
```

Alternatively, you can use the **tie()** function to “tie” variables to values.<sup>5</sup> This

---

<sup>5</sup>If you want to know how this works: **tie** returns a tuple of references to the variables it is passed. If we have an **int** **a** and a **char** **b**, the return type of **tie(a,b)** is **tuple<int&, char&>**.

lets you assign each element of the tuple to a variable in one line:

```
5 char c;
6 float f;
7 int i;
8 string s;
9
10 tie(c, f, i, s) = thing;
11
12 cout << c << ' ' << f << ' '
13      << i << ' ' << s << endl;
```

In the above example, we use `tie()` to assign the items in `thing` to `c`, `f`, `i`, and `s` respectively. After the last line, `c == 'x'`, `f == 2.5`, `i == 4`, and `s == "ABC"`.

If you want to use `tie`, but have some values in your tuple that you don't want to assign to any variable, you can use the `ignore` object. Consider the following example:

```
1 // A coordinate with a name
2 tuple<int,int,string> coord_name(2,4,"A");
3
4 // Prints (A, 2, 4)
5 cout << get<2>(coord_name) << ": ("
6      << get<0>(coord_name) << ","
7      << get<1>(coord_name) << ")\n";
8
9 int x, y;
10
11 // Unpacks the first two items into x and y, and ignores the last item.
12 tie(x, y, ignore) = coord_name;
13
14 // Prints (2, 4)
15 cout << "(" << x << "," << y << ")\n";
```

One cool use for the `tie` function is to “return multiple values”. See those quotes? You're really just returning one `tuple`, but with a call to `tie()`, it's kinda like you're returning more than one thing at a time.

```
1 tuple<int,int> divide(int divisor, int dividend)
2 {
3     // You can use make_tuple instead of the constructor if you want.
4     // If the template type is crazy (lots of items in the tuple)
5     // you might find make_tuple easier to read.
6     return make_tuple(divisor / dividend, divisor % dividend);
7 }
8
9 int main()
```



```

10 {
11     int quotient, remainder;
12
13     // Whoa! We set those two variables at once!
14     tie(quotient, remainder) = divide(13,5);
15
16     cout << "13 / 5 = " << quotient
17          << " with remainder " << remainder << endl;
18
19     return 0;
20 }

```

### **std::pair (#include<utility>)**

“Now don’t tell anybody, but the `STØR::pair` is just a `STØR::tuple` with two items. The rules are the same, but it adds a couple of convenience functions. I mean features.”

---

The `std::pair` type is a lot like the `std::tuple` type, but it holds exactly two items. You get to decide what types those two things have. You can access those items like a struct using `.first` and `.second` to get the first and second item, respectively.

```

1 // You can use the good ol' constructor of course
2 pair<int,string> origin = pair<int,string>(0,"bleep");
3 cout << "origin: (" << origin.first << ","
4      << origin.second << ")" << endl;
5
6 // There's also a handy function to make a pair
7 pair<int,int> coord = make_pair(3,5);
8 cout << "coord: (" << coord.first << ","
9      << coord.second << ")" << endl;

```

We’ll see `pair` used in the next section as a handy way to pass around two values at once.

### **std::map (#include<map>)**

The `STØR` guide leads you to a boxy contraption with a trap door on the top and some kind of laser scanner on the side.

“Now this is our `STØR::map`. It’s a bit more complex than the `STØR::vector`, but it sure is handy! Yes indeedly do!”

More staring.

“Lots of our customers struggle with keeping pairs of things together. Shoes, socks, you name it! They find one part of the pair, and they can’t find its buddy!”

“The `STØR::map` helps you keep track of your pairs. Watch this!”

The man takes his shoes off<sup>6</sup> and scans the left shoe. The trap door opens. He then drops his right shoe into the machine and closes the door again. The display on the machine shows `<LeftShoe,RightShoe>`.

“All ri-diddly-ight, my friend. Now, you see, the machine is configured to store pairs of shoes. You scan the left one, and it stores the corresponding right shoe. Now watch this! Give me your shoes!”

Nope.

“That’s alright. I got a demo pair here.”<sup>7</sup> He uses the same procedure to store the demo pair: scan, store, close. “Now, I’ve got two left shoes here.” He holds up his left shoe “If I want to get Lefty’s twin, I just place it under the scanner here.”

He does. The trap door opens, and he pulls out his right shoe.

“You see? And this works for any kind of thing. I can match left shoes to pepper shakers or apples to oranges! The only trick is that whatever you scan has to be the same kind of thing, and whatever you store has to be the same kind of thing. If you scan apples, that’s all it can scan. Right now this `STØR::map` scans `LeftShoes` and stores `RightShoes`. See the display?”

You nod.

He picks up the `STØR::map` and places it in the back of a golf cart sort of vehicle<sup>8</sup>, and gestures toward the passenger’s seat.

“Looks like we’re done with your list! Let’s head to the checkout!”

As you ride toward the checkout you marvel at the amazing, Scandinavian technologies you’ve seen. They’ve invented so many containers for so many purposes.

Your mind runs wild with C++ analogies.

---

As its name implies, a `std::map` maps keys to values. It’s sort of like a real-life dictionary. If you look up a word (key), you’ll find its definition (value). Another way of thinking about it is it’s like an array, but instead of having to use the integers 0, 1, ... for indices, you can use whatever type you like.

---

<sup>6</sup>Despite all objections.

<sup>7</sup>They’re gross.

<sup>8</sup>If golf carts were manufactured by Willy Wonka.

With a `std::map`, you get to decide on the type for the key and the type for the value. Like a vector, a `std::map` can change size to hold more items on the fly. It can hold as many key/value pairs as you'd like. The size of a `std::map` corresponds to the number of key/value **pairs** that have been set.

```
1 // Create a map that maps strings to floats
2 map<string, float> costs;
3
4 // Use the bracket and assignment operators to set values for keys
5 costs["beer"] = 5.5;
6 costs["soda"] = 6.0;
7
8 // oh wait. Beer is cheaper than that.
9 costs["beer"] = 4.85;
```

So, here we have a dictionary with the following definitions:

- "beer" maps to 4.85
- "soda" maps to 6.0

We can use the bracket operator to get the set values out, too. Be careful, though! If you try to access a value using a key that doesn't exist, the `std::map` will give you a **default value**.

```
10 // Prints out zero!!!
11 cout << costs["orange juice"] << endl;
```

You might expect the `std::map` to say "Hey man. I don't know what that is" and throw an exception at your face. That's not what happens. Instead, you're responsible for checking that a key exists before you try to access it. Here are a couple of ways to do that:

Use the `count()` member function to see if the key is there:

```
12 if(costs.count("beer") > 0)
13 {
14     // Use the bracket operator to actually get the value
15     cout << "beer costs " << costs["beer"] << endl;
16 }
```

Or we can use the `find()` member function. If the item is in the map, `find()` returns an iterator that points to the item; otherwise, it returns an iterator to after the end of the map.

```
17 map<string, float>::iterator iter = costs.find("beer");
18 if(iter != costs.end())
19 {
20     // We can just use the iterator to access the item
21     cout << "beer costs " << *iter << endl;
22 }
```

Like a `std::vector`, you can iterate over a `std::map`. Unlike a `std::vector`, there's no easy way to just loop through all the indices. So, we'll have to use an iterator instead. Let's look at an example to see how:

```
1 map<string, int> ages;
2 ages["rick"] = 70;
3 ages["morty"] = 14;
4
5 // That iterator type would be quite a doozy to write!
6 for (auto it = ages.begin(); it != ages.end(); it++)
7 {
8     // Dereferencing the iterator gives us key/value pairs
9     pair<const string, int> p = *it;
10    cout << p.first << " is "
11         << p.second << " years old." << endl;
12 }
13
14 for (auto it = ages.begin(); it != ages.end(); it++)
15 {
16     // Alternatively, we can dereference and access in one step.
17     cout << it->first << " is "
18         << it->second << " years old." << endl;
19 }
```

Alternatively, we can use a for-each loop! This handles all the iterator stuff for us; we just see the `pair<key,value>` objects.

```
20 for (auto kv : ages)
21 {
22     // No need to dereference!
23     cout << kv.first << " is "
24         << kv.second << " years old." << endl;
25 }
```

Whenever you iterate over a `std::map`, you iterate over its **key/value pairs**. It's super handy, but it can be a little tricky at first. In order to iterate over the key/value pairs, the `std::map` iterator points to instances of `std::pair`. The type of the `std::pair` corresponds to the type of the `std::map`. Iterating over a `std::map<string,int>` will give you `std::pair<string,int>`s. For each pair, you can access the key using `.first` and the value using `.second`.

## Questions

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

1. Describe each of the following items (in your own) terms using a single sentence (one sentence a piece):

- a. `std::vector`

- b. `std::map`

- c. `std::pair`

2. In a single line of valid C++, define and initialize a variable called `toad` of type `std::tuple` that stores 5 (an `int`), 2.3 (a `float`), and "boots" (a `string`) in that order. Use `make_tuple()`. You may use `auto` if you want to.

3. What's wrong with the following code snippet?

```
1  map<string, int> stuff;  
2  
3  stuff["bob"] = 10;  
4  stuff["linda"] = 12;  
5  
6  cout << stuff[10] << endl;
```

## Quick Reference

### Using iterators

- Use `*it` to get the value the iterator is pointing at
- Use `it->member` to access member variables and functions of the value the iterator is pointing at
- Increment to the next element with `++`
- Some iterators can move backwards with `--`
- Some iterators can skip forward or backward a number of elements if you add that number to them
- An iterator equal to `end()` means you have reached the end of elements in a container

### **std::vector**

- `push_back(element)` adds new elements to the end of a vector
- `pop_back()` removes elements from the end of a vector
- `operator[]` and `at()` access elements; `at()` throws an `out_of_range` exception if the index is out of range
- `size()` returns the number of elements in a vector
- `erase(iterator)` removes an element from a vector
- `insert(iterator, element)` inserts an element into the middle of a vector

### **std::tuple**

- `make_tuple(elem1, elem2, ...)` creates a tuple with the given elements: `auto two_things = make_tuple("Bender", "Fry");`
- `get<N>(tuple)` gets the Nth element from a tuple
- Use `tie(var1, var2, ...) = tuple` to assign each value from the tuple to a variable in one line

### **std::pair**

- Like a tuple, but has exactly two elements
- `make_pair(thing1, thing2)` creates a pair containing the two things
- Access the first element with `pair.first` and the second with `pair.second`

## **std::map**

- Add new elements with `operator[]`: `my_map["coffee"] = 1.99;`
- `count(key)` returns the number of entries for that key in the map (either 0 or 1)
- `find(key)` returns an iterator pointing at the key-value pair corresponding to that key
- A key is in the map if `my_map.count(key) > 0` or `my_map.find(key) != my_map.end()`
- Iterating over a map iterates over the pairs of keys and associated values in the map

## **Further Reading**

- [The `auto` keyword](#)
- [For-each loops](#)
- [std::vector](#)
- [std::tuple](#)
- [std::pair](#)
- [std::map](#)





## 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.

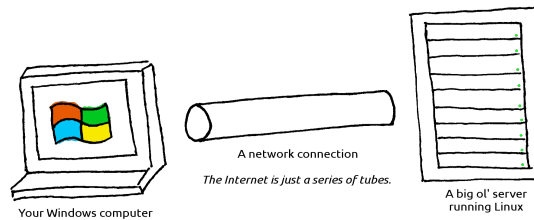


Figure A.1: PuTTY (running on your Windows computer) connects to a Linux computer over a network.

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.

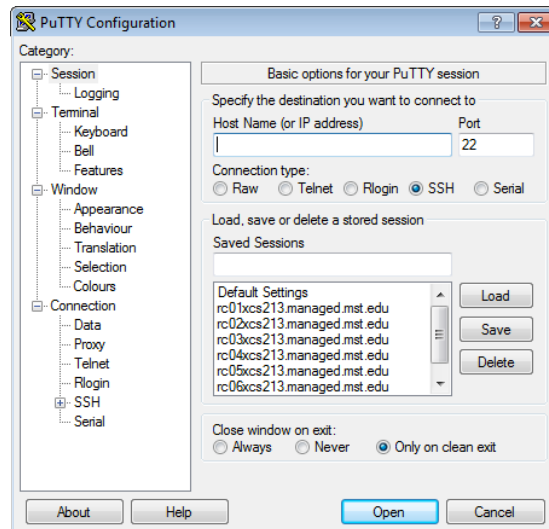


Figure A.2: The initial PuTTY screen

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.

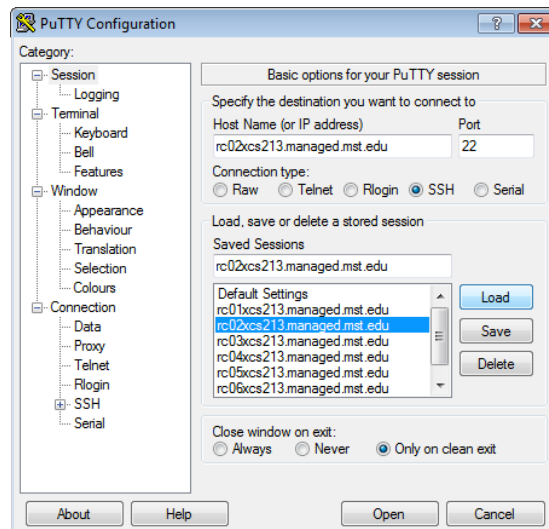


Figure A.3: Choose a configuration from the list and press the **Load** button to load it up.

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.

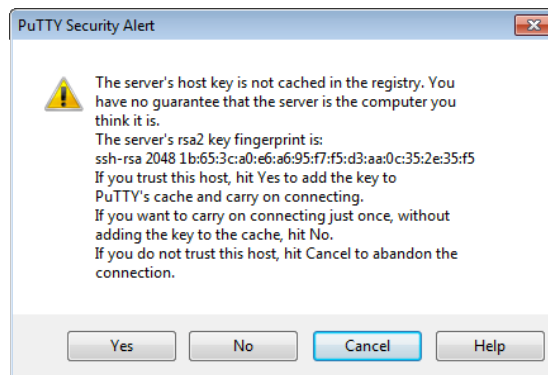


Figure A.4: If you've never connected to a particular Linux machine, PuTTY will ask if you're sure you want to connect.

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).

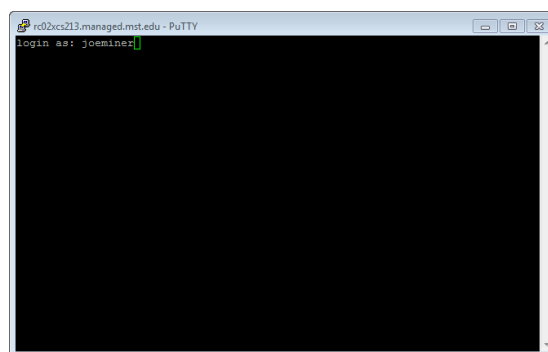


Figure A.5: If necessary, tell PuTTY your username.

---

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

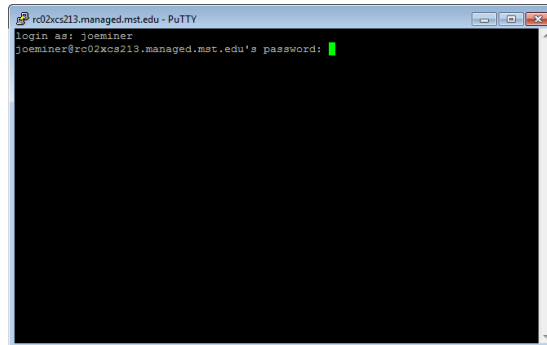


Figure A.6: Use your Single Sign-On password to sign in.

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.

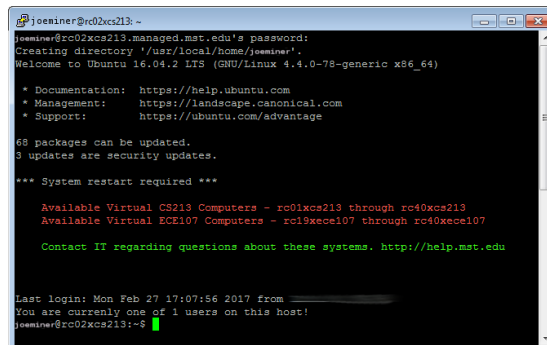


Figure A.7: Once you've connected, you'll be presented with a welcome message and a **bash** prompt.

## 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.

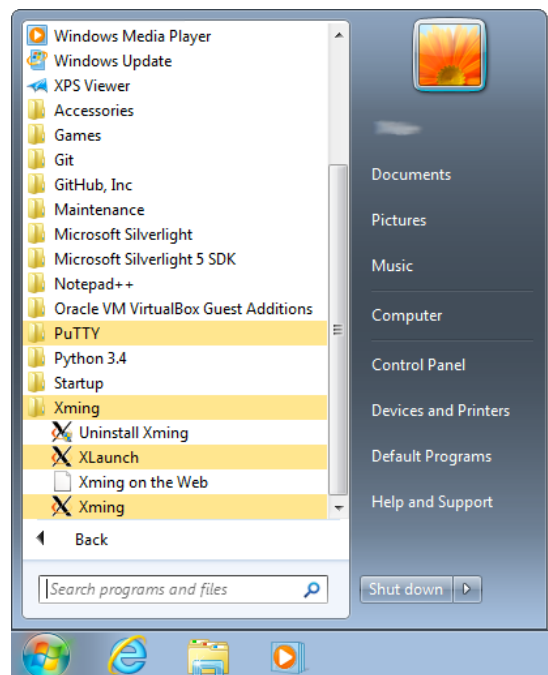


Figure B.1: We want to start Xming. **Not** XLaunch or anything else. Xming.

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.



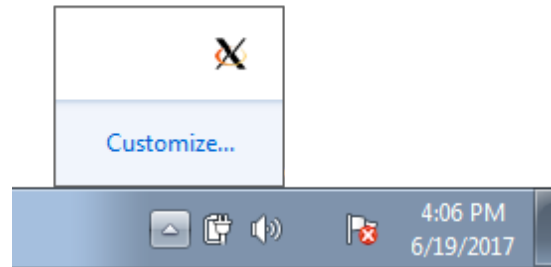


Figure B.2: You can check your task bar to see if Xming is running. ??

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.

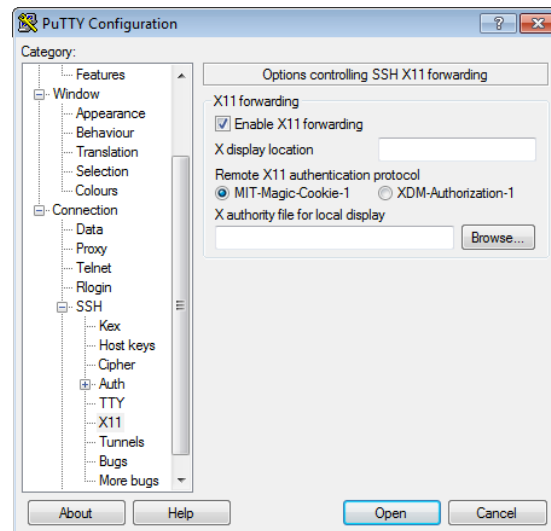


Figure B.3: Make **sure** that X11 Forwarding is enabled!

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

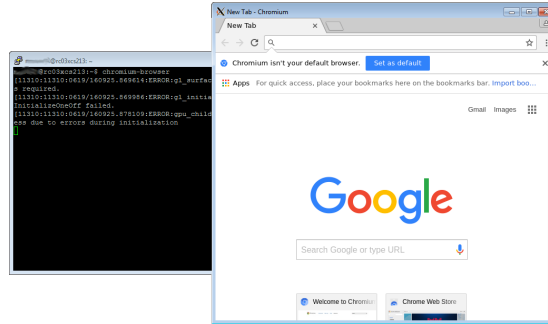


Figure B.4: The Chromium Browser forwarded to our Windows computer. Remember that Chromium is running *on the Linux computer*. PuTTY is forwarding the display data, and Xming is drawing the browser window for us.

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++

Throughout this book you'll find commands like `ls -l` or `grep needle haystack.txt`. Everything following the command name (`ls` or `grep`) is a command *argument*. You too can write programs that take arguments!

In C++, the arguments given to a command are passed as parameters to your `main` function. Instead of writing `int main()`, your `main` function will take two arguments:

- An `int` which gives the number of arguments passed on the command line.
- A `char**` which is an array of NTCA's<sup>1</sup> (also known as C-strings) which are the string values of the arguments passed.

These arguments are traditionally named `argc` and `argv`, for “argument count” and “argument values”, respectively. You can name them whatever you like, however; C++ doesn't care.

Let's do an example: here is a program that prints its arguments out to the screen, one per line:

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

int main(int argc, char** argv)
{
    cout << "Index\tArgument" << endl;
```

---

<sup>1</sup>Yep, it's a two-dimensional array. We use `char**` rather than something like `char[][100]` since we don't know how big the array or its subarrays will be; the operating system works this out when it runs your program.

```

    for(int i = 0; i < argc; i++)
    {
        cout << i << "\t" << argv[i] << endl;
    }

    return 0;
}

```

Let's run it and see what happens:

```

$ ./print a b c
Index  Argument
0      ./print
1      a
2      b
3      c

```

The “0th” argument is always passed; it is the name of the program that was run. So the first argument on the command line is at `argv[1]`, and so on.

For a practical example, let's write a program that counts the number of lines in a file (like `wc -l`, but less fancy). If the user doesn't pass us exactly one file, we want to print out a message telling them how to use the program; otherwise, we should open the file and count the number of newlines in it.

```

1  #include<iostream>
2  #include<fstream>
3  using namespace std;
4
5  int main(int argc, char** argv)
6  {
7      if(argc != 2)
8      {
9          cout << "Counts the number of newlines in the given filename"
10             << endl << "Usage: " << argv[0] << " [<filename>]" << endl;
11         return 1; // exit the program
12     }
13
14     ifstream fin(argv[1]);
15     char buffer;
16     int newline_count = 0;
17
18     while(fin.get(buffer))
19     {
20         if(buffer == '\n')
21         {
22             newline_count++;
23         }

```

```

24     }
25
26     fin.close();
27
28     cout << argv[1] << " has " << newline_count << " lines." << endl;
29
30     return 0;
31 }

```

In this example, you can see the use of `argv[0]` to display the program name. It is a common pattern to check the arguments passed before doing anything else and exit the program if they are incorrect. Finally, we can use `argv[1]` as we would any other NTCA. We could even access the individual characters of the filename by doing something like `argv[1][0]`.

Now, you may be wondering, “how do I get fancy-dancy options parsing, like `ls` and friends?” Well, there are a few options. The classic choice is [getopt](#), but the C standard library also has [a few other options](#). Boost also has a C++-style library for argument parsing, [program\\_options](#). Or, you can write your own!





## Appendix E

# Submitting homework with Git

Your instructor may have you submit your assignments using Git and GitLab. These tools (or tools like them) are commonly used in industry, so getting accustomed to them early will pay off in the long run.

Git is an open source, distributed version control system that allows programmers to track changes made to source files and share those changes with collaborators. It was created by Linus Torvalds (the Linux guy) and has become hugely popular in the last several years.

GitLab is an open source web application that makes it a bit easier for programmers to collaborate on programming projects. GitLab lets users create repositories (projects) on a central server. Users can then push their code to GitLab using git. From there, users can clone their code to other machines, push up new changes, etc. You could think of it (kind of) like Dropbox – GitLab is a central place to store your code.

In addition to storing code, GitLab has many features to help users collaborate with others. Users can share projects with other users, giving them the ability to push/pull code to/from the project. There are also bug tracking utilities, code review tools, and much more.

In this class, we'll be using just the git parts of GitLab, so don't worry if the fancier features sound difficult. All you need to be able to do is

- View the GitLab website.
- Retrieve (clone) the lab assignments.
- Record (commit) your solutions to those assignments.
- Upload (push) those solutions back to GitLab.

Once you are done with an assignment, you should **always** check GitLab to make sure that your solutions are properly uploaded.

## Warning

Do not blindly copy and paste git commands from the internet and run them. (This isn't specific to git — that's just plain ol' good life advice right there.) While all the common git commands are quite good at not losing your hard work, there *are* commands that can wreak havoc if used improperly. If you don't know what a command does, take a few minutes to read some documentation on it first.

If things are not working right, first check `git status` and `git log` to see what git thinks has happened. When that doesn't clear things up, have a look on your favorite search engine and/or ask a friend or your instructor for help. *Definitely* ask someone if the internet is recommending you run unfamiliar git commands!

## Configuring Git

Before you can use git, you need to configure a couple of things.

### Who Are You?

Don't worry, no need for an existential crisis. Git just wants to know your name and your email address.

Open a terminal (i.e., PuTTY) and run these commands (substituting your own name and email, of course):

```
$ git config --global user.name "Hank Chicken"
$ git config --global user.email hank@chickens.com
```

Git probably won't say anything when you run these commands; it's the strong silent type. Don't worry, you're fine.

## Editing

Git sometimes will open a text editor so you can type in certain things (mainly commit messages). The default editor it uses is joe, which almost no one knows how to use.<sup>1</sup> You will most likely want to set the editor to one you know and

---

<sup>1</sup>By the way, `Ctrl`+`k`+`x` will exit joe, should you forget to configure your editor before using git.

love.

To change the editor to `jpico`, run the following command (substitute a different editor if you like):

```
$ git config --global core.editor jpico
```

## Working on assignments

Here's the gist of how to do assignments:

1. Get the assignment
  - Clone the repository
  - Open the assignment directions
2. Do the assignment
  - Commit changes as you go
  - Push to GitLab if you like
3. Turn it in
  - Push all your work to GitLab
  - Check the GitLab website to make sure your submission looks right

## Getting the assignment (**git clone**)

For each lab assignment you will be granted access to a git repository on GitLab that contains starter code and examples. These repositories are visible only to you and the instructor and graders. You will be granted [developer](#) permissions to your repositories. This will allow you to view the project and push changes to it.

When you start an assignment, the first thing you must do is download a copy of the repository so you can make changes to it.

1. Log in to <https://git-classes.mst.edu>.
2. Navigate to the project that you want retrieve.
3. Copy your repository's URL from the text box in the middle of the page beneath the assignment name.
  - If you don't see it, try making your browser window wider. Hooray responsive web design!
  - Cloning over HTTPS (easy):
    1. Select 'HTTPS' from the dropdown next to the text box.
    2. Copy the URL from the text box. It should start with `https://git-classes.mst.edu`.
  - Cloning over SSH (if you have [ssh keys](#) configured):
    1. Select 'SSH' from the dropdown next to the text box.
    2. Copy the URL from the text box. It should start with `git@git-classes.mst.edu`.

4. Open up a terminal (i.e., PuTTY) and navigate to the directory (using `cd`) where you want to put your copy of the repository.
5. Run `git clone <URL you copied in step 3>`.

That's it! Now you've cloned your repository down, and you're ready to get started.

#### Resources:

- [Pro Git: Cloning an Existing Repository](#)
- [Atlassian Git Tutorials: git clone](#)

## Working on the assignment

As far as working on the files go, there's nothing special you have to do. Edit them, compile them, have fun with them. When you're ready to commit (take a snapshot of) your code, you'll need to interact with git.

#### Committing code (`git commit`)

1. Run `git status` to see which files are **staged**, **modified**, or **untracked**.
  - **staged** files will be included in the next commit.
  - **modified** files have been modified, but they haven't been staged for commit.
  - **untracked** files are unfamiliar to git. They exist in the folder, but git isn't paying any attention to changes made to them.
2. Stage all changes that you want to commit.
  - Stage files by using the `git add` command.
  - For example: `git add main.cpp`.
3. Run `git status` again to make sure that the correct files are staged for commit.
4. Run `git commit`. This will create a snapshot – recording the staged changes.
5. Enter a meaningful commit message in the text editor that opens. Then save and close it.

Now your changes have been committed!

#### Resources:

- [Pro Git: Recording Changes](#)
- [Atlassian Git Tutorials: Saving changes](#)

## Looking through your Repository (**git status**, **git log**)

There are a couple of commands that come in handy for viewing the state of your repository:

- **git status**: View the current state of files. What has changed? What's new in the repository?
- **git log**: Shows a timeline of commits. See a history of changes made to a repository.

### Resources:

- [Atlassian Git Tutorials: Inspecting a repository](#)

## Pushing your Commits (**git push**)

After you've committed something, you can push it to GitLab with `git push`. We recommend pushing frequently, just for peace of mind.

1. Run **git push**
  - If you cloned over HTTPS, Git will ask for your username and password. Those are the same as the ones you use to log into GitLab.
2. Check GitLab to ensure your code looks as expected.
  - This step isn't strictly required. However, if you're not used to working with Git, it is in your best interest to ensure your code looks right. We're going to grade what's in GitLab, after all.

### Resources:

- [Pro Git: Pushing to your Remotes](#)
- [Atlassian Git Tutorials: git push](#)

## Turning in your assignment (**git push**)

Once you've completed your assignment, make sure to commit and push all your work! If your work isn't on GitLab, we can't grade it. You should also check GitLab to make sure your submission isn't missing any files.

## Helpful Resources

These are some nice resources for learning more about Git:

- [In-browser Git Tutorial](#) (**Do this one.** It's a nice tutorial.)
- [Atlassian's Git Tutorials](#)
- [Pro Git](#)

- [GitLab's Help Pages](#) (You may need to log into git-classes.mst.edu for that to work.)
- [Introductory Videos from GitHub](#)

If you see mentions of BitBucket or GitHub around, they're just web applications like GitLab. Same basic idea, created different people.