CS 200: Concepts of Programming using C++ (Spring 2025 version)

Rachel Wil Sha Singh

February 21, 2025

Contents

- **I will be updating this book with semester content as the semester goes on.** By the end of this semester, you will be able to download this PDF as an archive of the class topics.
-
- Digital textbooks: https://moosadee.gitlab.io/courses/
- These course documents are written in **emacs orgmode** and the files can be found here: https://gitlab.com/moosadee/courses

- Dedicated to a better world, and those who work to try to create one.

1 Week 0: Welcome, computer context

1.1 Introductions!

Hi everyone! I'm Rachel Wil Singh (R.W. in the JCCC system, and I also go by "Moosie" or "Moose"). My pronouns are they/them. I am a full time associate professor at JCCC and I will be teaching you about programming this semester!

Background: A.S./CompSci from Longview, B.S./CompSci from UMKC (2009). Worked professionally in web and software development starting in the 2010s with a variety of languages (C++, C#, HTML, CSS, JS, SQL, PHP, Python).

Hobby-wise, in my free time I program indie video games Links to an external site., make cartoons in Esperanto, study Hindi, and tend my vegetable garden. My family currently consists of my husband and I and our four cats. My husband Rai is from Uttarakhand in India and is also in software development. I am also neurodivergent.

1.2 Ethics and Academic Honesty

- **Open-book, open-note:** In software development you're usually able to freely **research** and **try writing code** as part of your job. I see my assignments as open book and open note, as well as open IDE.
- **Don't plagiarize:** Don't present someone or something else's work as *your own work*.
- **AI is a search engine:** People posting online in forums makes mistakes. AI makes mistakes. You can try asking it questions like you would with a search engine, but you need to have the experience to know what is and is not correct.
- **Difficulty curve:** I've designed everything in this course, trying to set things up so it starts easier and hand-holdy and increases with difficulty over time, like a video game.
- **Resources:** I want to help you learn how to program. If you're stuck on something or need help with problem solving, I am here to help you out. I have office hours, class time, you can email me on Canvas or message me on Discord. The college also has resources.
- **Citations:** If you're going to use a snippet of code from elseware, please cite it by leaving a comment to the URL or source.

1.3 Study skills and course success overview

Course design: For this course, learning resources include concept introduction "quizzes" (it's reading with review questions), the textbook reading, pre-made video lectures, and the class time itself. Weekly, there are the concept introduction assignments and programming labs, as well as occasional discussion boards and 4 projects for the semester. There is 1 exam that you can re-take once a week as you'd like. Make sure to keep up with course content: It's harder to catch up on the harder material later on if you haven't done the earlier things. In this course, the content builds on each other. I'm always available during class, drop-in "office hours" (in person or via Zoom), or we can schedule a one-on-one Zoom time to meet as well. Let me know what you're stuck on and we'll work through it. Class time is mostly meant for you to work on the programming assignments. That way, I'm immediately available if you have questions or get stuck. It's in your schedule, might as well make use of the time!

Keeping track: Canvas has a Calendar feature, though it might be useful to also use something like Google Calendar with email/text reminders. I use a paper day planner for everything, but it's something I always have to carry with me (yay ADHD). I'll post announcements on the course Canvas page periodically with course information, corrections to assignments, etc.

Hitting a wall: Sometimes when you're stuck on a program it's best to just step away totally. If you feel like you're stuck and making no progress, usually time away really helps. You can also reach out to me or classmates or post on the Discord chat for hints.

2 Week 1: Setup, main(), variables

2.1 Intro: Welcome to my course!

It feels weird to start a collection of notes (or a "textbook") without some sort of welcome, though at the same time I know that people are probably *not* going to read the introduction (Unless I put some cute art and interesting footnotes, maybe.) I think that I will welcome you to my notes by addressing anxiety.

~

2.1.1 Belonging

Unfortunately there is a lot of bias in STEM fields and over decades there has been a narrative that computer science is *for* a certain type of person - antisocial, nerdy, people who started coding when they were 10 years old.

Because of this, a lot of people who don't fit this description can be hesitant to get into computers or programming because they don't see people like themselves in media portrayals. Or perhaps previous professors or peers have acted like you're not a *real programmer* if you didn't start programming as a child

If you want to learn about coding, then you belong here.

There are no prerequisites.

I will say from my own experience, I know developers who fell in love with programming by accident as an adult after having to take a computer class for a *different degree*. I know developers who are into all sorts of sports, or into photography, or into fashion. There is no specific "type" of programmer. You can be any religion, any gender, any color, from any country, and be a programmer.

~

2.1.2 Challenge

Programming can be hard sometimes. There are many aspects of learning to write software (or websites, apps, games, etc.) and you **will** get better at it with practice and with time. But I completely understand the feeling of hitting your head against a wall wondering *why won't this work?!* and even wondering *am I cut out for this?!* - Yes, you are.

I will tell you right now, I *have* cried over programming assignments, over work, over software. I have taken my laptop with me to a family holiday celebration because I *couldn't figure out this program* and I *had to* get it done!!

All developers struggle. Software is a really unique field. It's very intangible, and there are lots of programming languages, and all sorts of tools, and various techniques. Nobody knows everything, and there's always more to learn.

Just because something is *hard* doesn't mean that it is *impossible*.

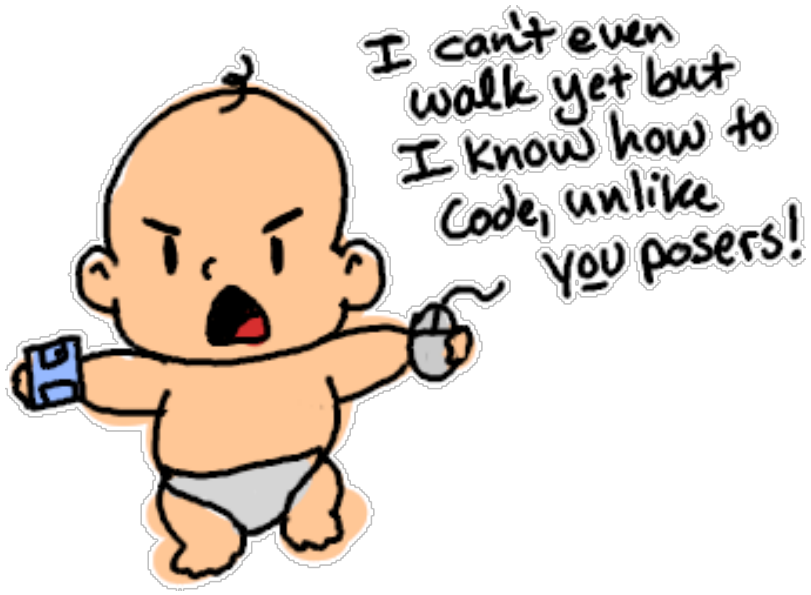


Figure 1: Don't be like Gatekeeper Baby. You can begin coding at any age!

It's completely natural to hit roadblocks. To have to step away from your program and come back to it later with a clear head. It's natural to be confused. It's natural to *not know*.

But some skills you will learn to make this process smoother are how to plan out your programs, test and verify your programs, how to phrase what you don't know as a question, how to ask for help. These are all skills that you will build up over time. Even if it feels like you're not making progress, I promise that you are, and hopefully at the end of our class you can look back to the start of it and realize how much you've learned and grown.

~

2.1.3 Learning

First and foremost, I am here to help you **learn**.

My teaching style is influenced on all my experiences throughout my learning career, my software engineer career, and my teaching career.

I have personally met teachers who have tried to *scare me away* from computers, I've had teachers who really encouraged me, I've had teachers who barely cared, I've had teachers who made class really fun and engaging.

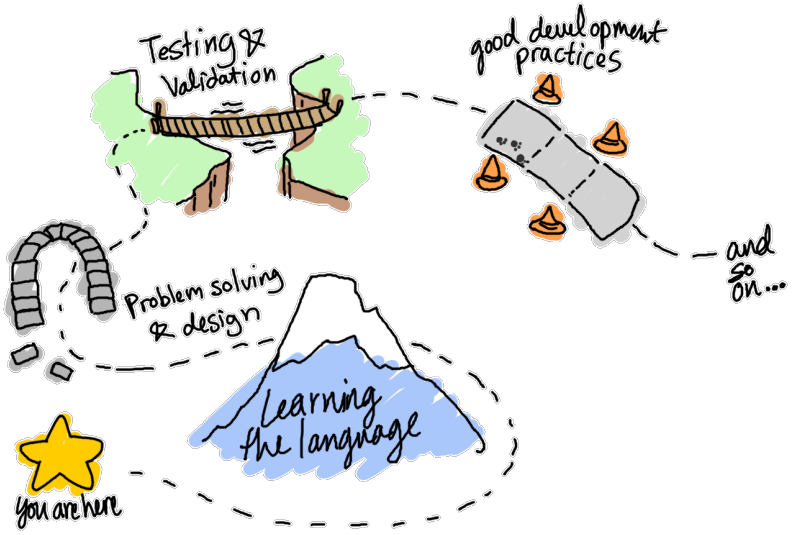
I've worked professionally in software and web development, and independently making apps and video games. I know what it's like to apply for jobs and work with teams of people and experience a software's development throughout its lifecycle.

And as a teacher I'm always trying to improve my classes - making the learning resources easily available and accessible, making assignments help build up your knowledge of the topics, trying to give feedback to help you design and write good programs.

As a teacher, I am not here to trick you with silly questions or decide whether you're a *real programmer* or not; I am here to help guide you to learn about programming, learn about design, learn about testing, and learn how to teach yourself.

~

2.1.4 Roadmap



When you're just starting out, it can be hard to know what all you're going to be learning about. I have certainly read course descriptions and just thought to myself "I have no idea what any of that meant, but it's required for my degree, so I guess I'm taking it!"

Here's kind of my mental map of how the courses I teach work:

CS 200: Concepts of Programming with C++ You're learning the language. Think of it like actually learning a human language; I'm teaching you words and the grammar, and at first you're just parroting what I say, but with practice you'll be able to build your own sentences.

CS 235 Object-Oriented Programming with C++ You're learning more about software development practices, design, testing, as well as more advanced object oriented concepts.

CS 250 Basic Data Structures with C++ You're learning about data, how to store data, how to assess how efficient algorithms are. Data data data.

In addition to learning about the language itself, I also try to sprinkle in other things I've learned from experience that I think you should know as a software developer (or someone who codes for whatever reason), like

- How do you validate that what you wrote *actually works?* (Spoilers: How to write tests, both manual and automated.)
- What tools can you use to make your programming life easier? (And are used in the professional world?)
- How do you design a solution given just some requirements?
- How do you network in the tech field?
- How do you find jobs?

- What are some issues facing tech fields today?

Something to keep in mind is that, if you're studying **Computer Science** as a degree (e.g., my Bachelor's degree is in Computer Science), technically that field is about *"how do computers work?"*, not about *"how do I write software good?"* but I still find these topics important to go over.

That's all I can really think of to write here. If you have any questions, let me know. Maybe I'll add on here.

~

2.1.5 What is this weird "book"?

In the past I've had all my course content available on the web on separate webpages. However, maintaining the HTML, CSS, and JS for this over time is cumbersome. Throughout 2023 I've been adapting my course content to *emacs orgmode* documents, which allows me to export the course content to HTML and PDF files. I have a few goals with this:

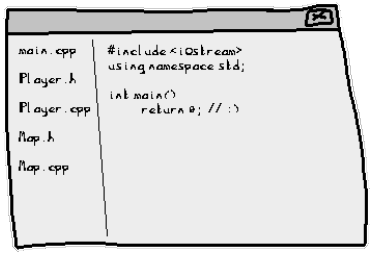
1. All course information is in one singular place
2. At the end of the semester, you can download the entire course's "stuff" in a single PDF file for easy referencing later on
3. Hopefully throughout this semester I'll get everything "moved over" to orgmode, and in Summer/Fall 2024 I can have a physical textbook printed for the courses, which students can use and take notes in so as to have most of their course stuff in one place as well
4. It's important to me to make sure that you have access to the course content even once you're not my student anymore - this resource is available publicly online, whether you're in the course or not. You can always reference it later.

I know a lot of text can be intimidating at first, but hopefully it will be less intimidating as the semester goes and we learn our way around this page.

2.2 Intro: Development tools

There are a wide range of tools that programmers can use to help them with software development, and since I want to make sure you're learning to be a well-rounded developer, I'm going to introduce you to some of the core tools of the trade.

2.2.1 Code editor



While **source code** is just **plaintext** and can be edited in any text editor, often we use a specialized **code editor** to make things easier on us.

Syntax highlighting: One of the big helps is **syntax highlighting**, where different types of program commands are highlighted in different colors. It makes a wall of code go from this:

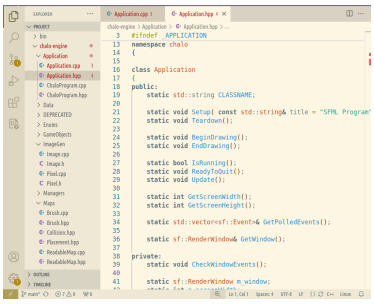
```
if ( argCount == 2 && args[1] == string( "test" ) )
{
    FixedArrayTester tester;
    tester.RunAll();
    return 0;
}

// Run program
FixedArray<string> courses;
courses.PushBack( "CS 134" );
courses.PushBack( "CS 200" );
courses.PushBack( "CS 235" );
courses.PushBack( "CS 250" );

cout << "Display array:" << endl;
courses.Display();
```

To this:

```
if ( argCount == 2 && args[1] == string( "test" ) )
{
    FixedArrayTester tester;
    tester.RunAll();
    return 0;
}

// Run program
```

```
FixedArray<string> courses;
courses.PushBack( "CS 134" );
courses.PushBack( "CS 200" );
courses.PushBack( "CS 235" );
courses.PushBack( "CS 250" );

cout << "Display array:" << endl;
courses.Display();
```

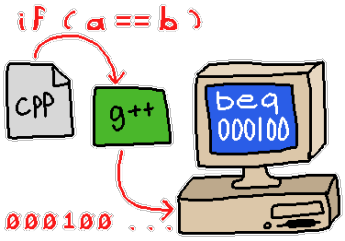
```
if ( argCount == 2 && args[1] == string( "test" ) )
{
    FixedArrayTester tester;
    tester.RunAll();
    return 0;
}

// Run program
FixedArray<string> courses;
courses.PushBack( "CS 134" );
courses.PushBack( "CS 200" );
courses.PushBack( "CS 235" );
courses.PushBack( "CS 250" );

cout << "Display array:" << endl;
courses.Display();
```



File management: Once we start getting better with C++ we'll be writing programs that will contain *multiple* source code files. A good code editor helps us organize and keep our program files together so we can more easily navigate our project.

~

2.2.2 Compiler

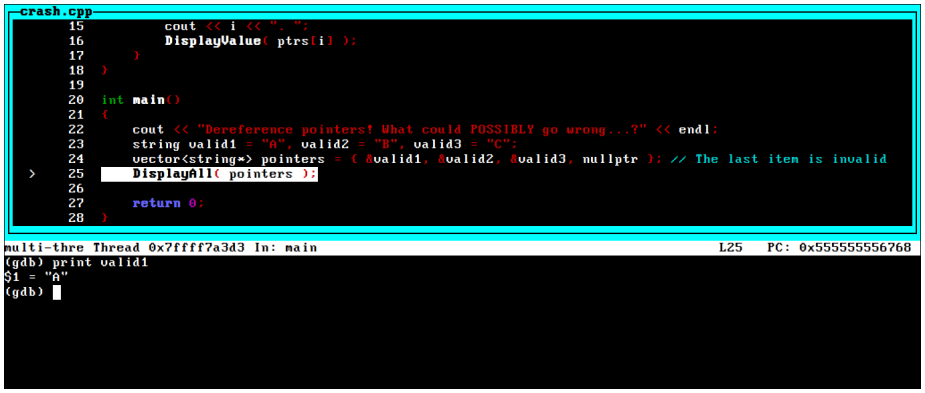


The **compiler** is a program that converts our human-readable C++ source code into machine-readable binary code. This process is known as **compiling** or **building**, and the resulting file is an **executable** or **binary file**, like "Program.exe" or "Program.out".

For this class, we're using the g++ compiler.

~

2.2.3 Debugger

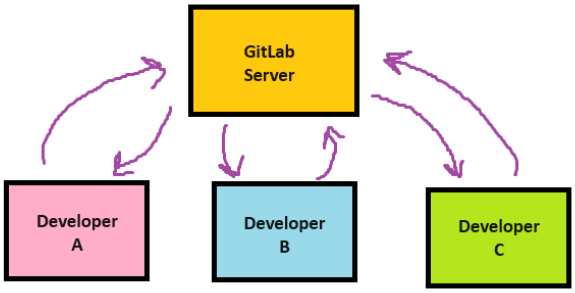


A **debugger** is a program that allows us to deep-dive into our own programs. By executing our program ***through*** a debugger, we can pause program execution, investigate variable values, look at function calls, and follow our program's flow.

We'll learn more about these tools later once we've covered more coding topics. :)

~

2.2.4 Source control



A **source control** system helps us manage the changes to our source code over time, as well as help us back up the code to a server, collaborate with others on the same code files, and lots more.

Git is a source control system and **GitLab** is a hosting service where we can store our code. This semester, you will be backing up your work to a GitLab **repository** that I create for you, and if you ever get stuck I can pull the code down to my own computer quickly, add comments, and sync it back to the server.

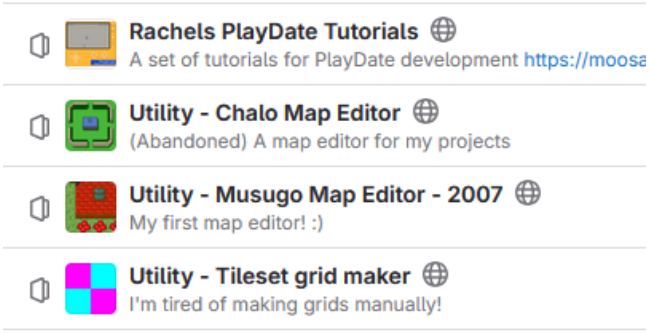
2.3 Intro: Source Control and git

2.3.1 What is Source Control?

Software companies usually have tens or even hundreds of developers working together on a product or products. They need to have an efficient way to merge their code together, as well as have the tools to do code reviews, make backups, and check past versions of code files. A Source Control (aka Version Control) solution gives us features for all of these.

Git and TFS are probably the most popular solutions currently in use, and some others include SVN and Mercurial. They each function somewhat differently, but share a lot of the same concepts. For our class we will be using Git.

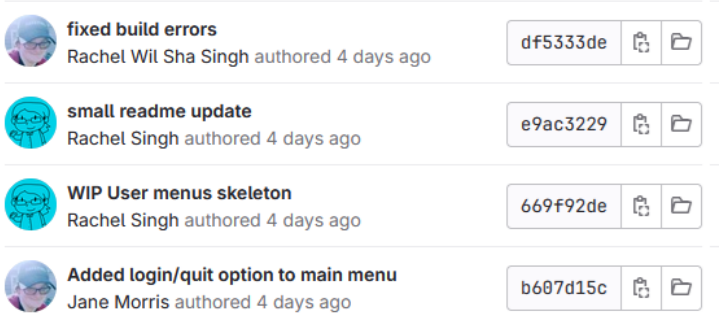
2.3.2 Repositories



A Repository is basically like a single "Project". You might have multiple repositories for different projects, or a company might have multiple repositories for different software products.

A Git Repository is set up so that Git takes care of tracking changes over time, create branches to work on new features, automatically merge changes with other people, and more.

2.3.3 Commit log



Git keeps track of changes made by different developers over time. Each time we use the commit command, a "snapshot" of our changes are made. Once synced to the GitLab server, we can see a list of all commits in the history of the project.

2.3.4 Git blame

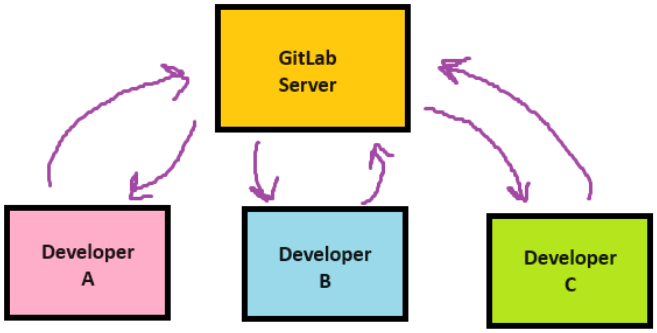
Rachels Courses / Shopazon / Commits / **df5333de**

Program/Program.cpp

```
284  284   }
285        -
     285   +
286  286   void Program::Menu_User_ViewAllStores()
287  287   {
288  288   }
...  ...   @@ -1173,7 +1173,7 @@ std::string Program::DisplaySubmenuGetStr( std::vector<st
1173 1173  {
1174 1174    int choice = DisplaySubmenu( options, zeroIsGoBack, vertical, col_width );
1175 1175
1176       -  if ( zeroIsGoBack && choice == "0" )
     1176  +  if ( zeroIsGoBack && choice == 0 )
1177 1177      {
1178 1178        return "goback";
1179 1179      }
...  ...
```

If we click on a commit, we can view a log of which lines of code were changed - red for "removed", + green for "added".

2.3.5 Git vs. GitLab

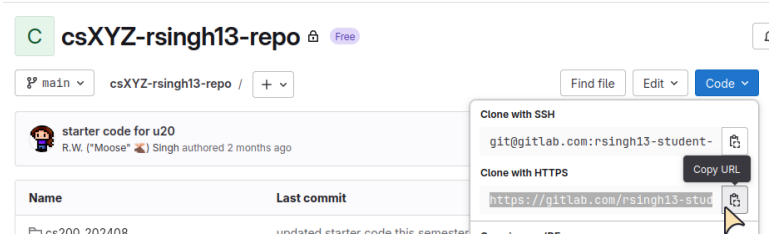


- Git itself is a whole system that facilitates this organized software development.
- GitLab is a service that provides hosting for Git repositories. (GitHub is another common example.)
- Each developer installs the Git software to their computer, which allows them to communicate to the repository's server with basic action commands.

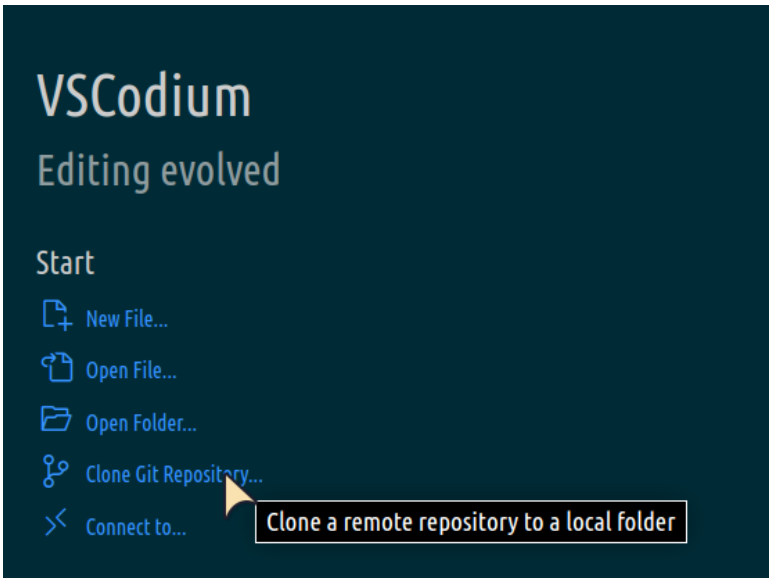
2.4 Reference: Using Git and VS Code

2.4.1 First time setup: Cloning your repository

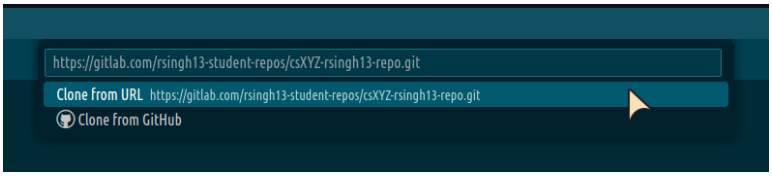
- After you tell the instructor your GitLab name they will give you access to a personal repository for your classwork.



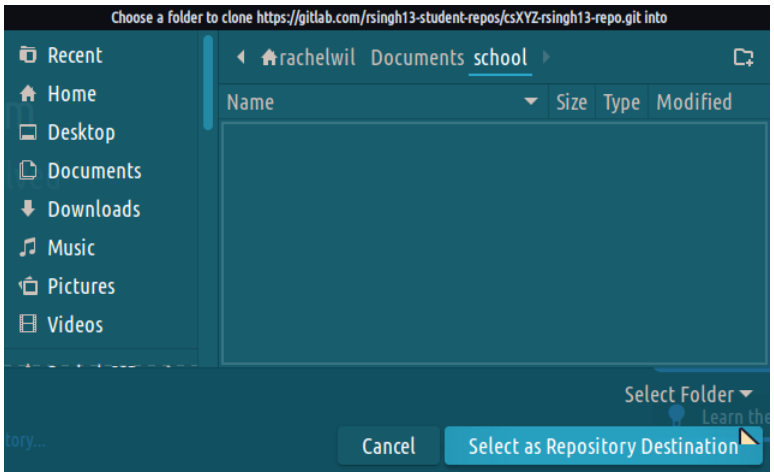
  - From your repository GitLab webpage, click the blue "Code" button, then copy the "HTTPS" link. (If you know about SSH keys, go for it.)



  - When you first start VS Code, there will be an option to **"Clone Git Repository"**. Click on this, and the top textbar will wait for a URL.



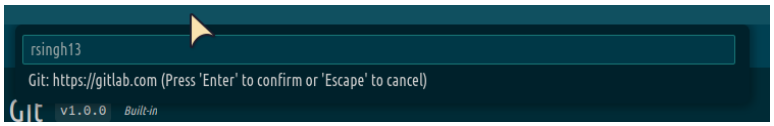
  - Paste your repo URL in this box and hit ENTER.

- 

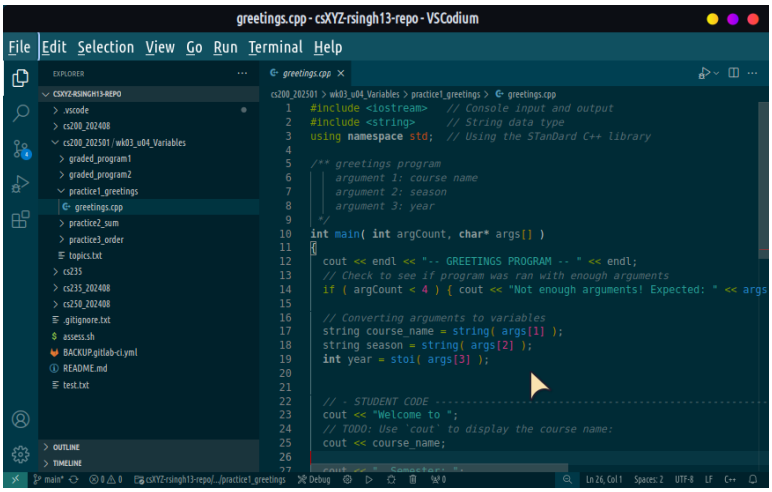
  - It will ask you where you want to store the repository folder on your hard drive. Find a location that you won't forget. :)

- 

  - The clone process will pull the files from the server to your computer.

- 

  - VS Code might ask for your Username in the top bar, or Windows might pop up a dialog box to have you sign in. Sign in with your GitLab account and password.



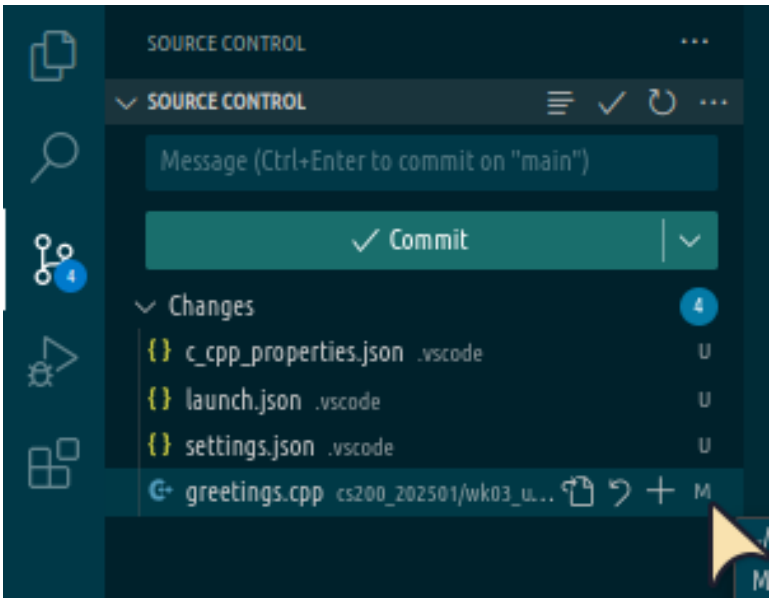
  - Once cloning is done, you can open the FOLDER for your repo and see any files within it, such as lab starter code.

1. Configuring Git

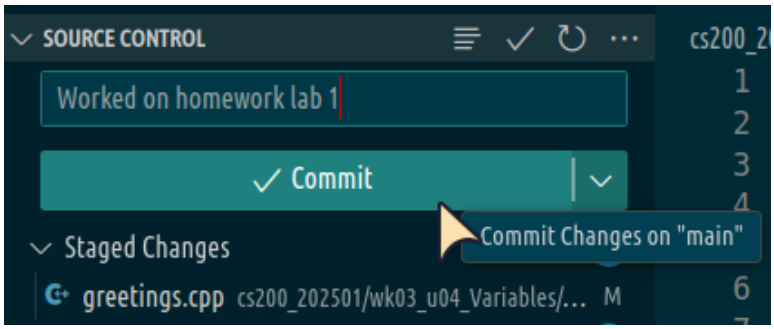
   You can run these commands either in VS Code or in Git Bash, but you'll need to enter a few commands to get everything set up.

   ```
   git config --global user.name "YOURNAME"

   git config --global user.email "YOUREMAIL"

   git config --global pull.rebase false
   ```

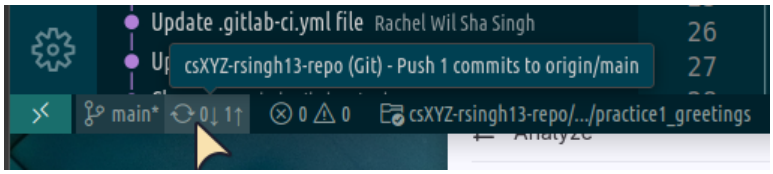
2.4.2 Making changes and backing them up



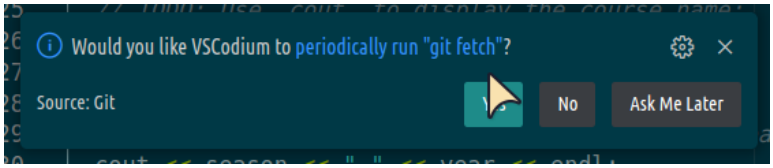
- After you've made a chance to one or more files, it will show up in the list of Changes under the Source Control button.
- Next to a file you want to backup, press the "+" button. It will then be categorized under "Staged Changes".

- 

- Add a description of your changes in the textbox above the **"Commit"** button. Once you're done, click **"Commit"**.

- 

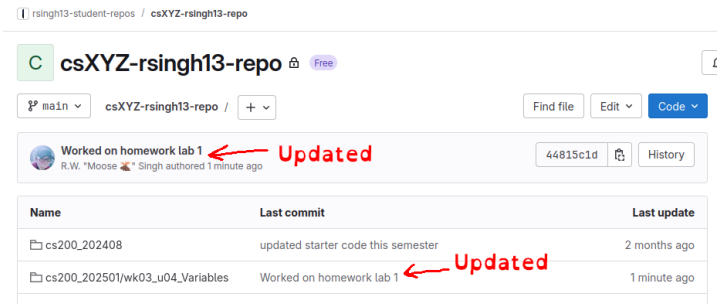
- To send your changes to the server for backup, click on the circular arrow icon at the bottom of VS Code. This will send your changes to the server and pull any changes (e.g., from the instructor) to your computer.

- 

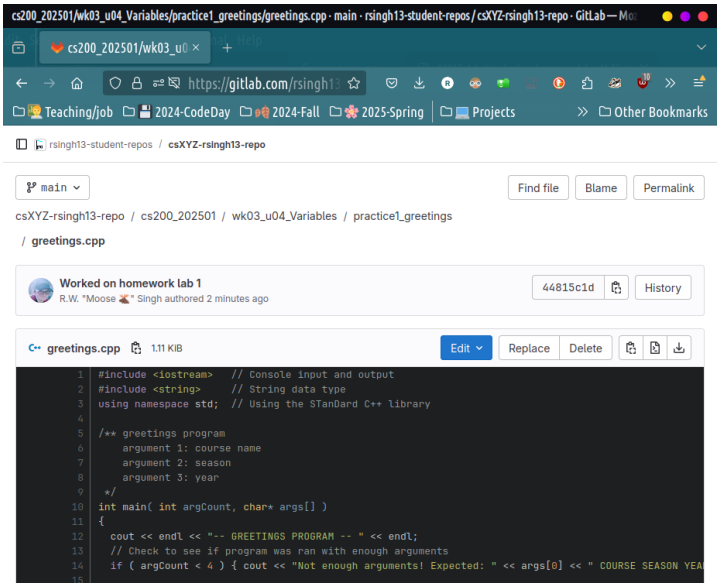
- It might ask if you want to periodically run "git fetch". This command pulls changes from the server. You can select "Yes".

1. Verify that your changes were saved

   - After committing your changes, make sure the up-to-date version shows up on GitLab.

- 

  - Go to your GitLab webpage. Make sure your commit message shows up here.

- 

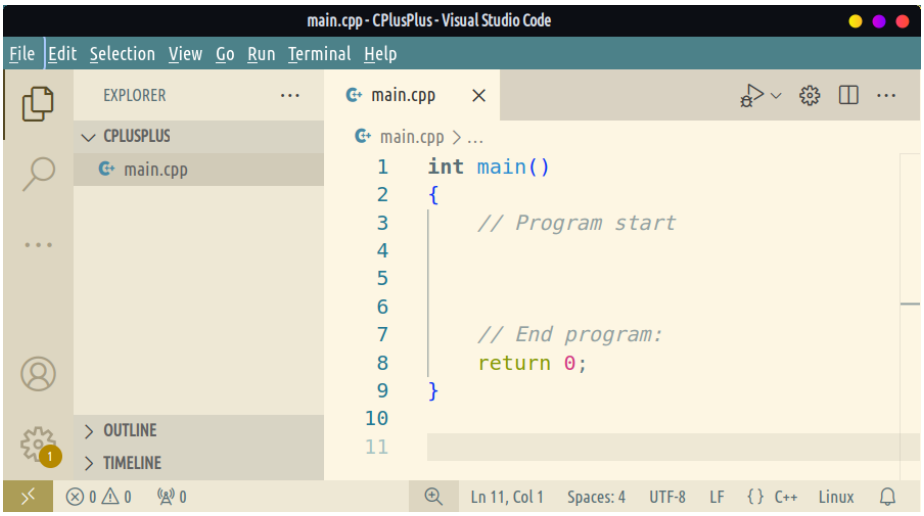
  - Also, navigate to the **file** that you updated from this GitLab web view.
  - Make sure this is the **latest version** of your file.

2.5 Intro: main() - Structure of a C++ program

2.5.1 C++ source code files

Our computer's **CPU** only understand commands given to it as **binary codes** - strings of 1's and 0's, like this: Early programmers had to write their software using this machine code, though over time programming languages evolved.

When we write **source code** these days, we write in a human-made, somewhat human-readable text, with specific grammars and rules to it. These source code files are **plaintext** - they're just simple text that can be opened in a text editor. We will be using VS Code as the editor where we will write our C++ source code.



Files on a computer use an **extention** to identify what *kind* of file it is, such as a `.jpg` for jpeg format images, `.mp3` for audio files, `.html` for website layouts.

At minimum, a C++ program requires one file - a `.cpp` source file. Usually programs are bigger and contain multiple files, like .h header files as well as other .cpp source files. The file that contains the program starting point is usually named `main.cpp`, though this name is not required.

~

2.5.2 Contents of a starter C++ program

The bare-minimum C++ program looks like this:

```
int main()
{

  return 0;
}
```

Every C++ program begins at a `main()` function.

For now you can take this code for granted but you will understand it more as the class continues. But, some basic info:

- The `int main()` is the function that starts our program.
- The contents of the program must be between the opening curly brace `{` and the closing curly brace `}`. This region is known as main()'s **codeblock**. We will see more codeblocks (with { }) later in the class.
- The `return 0;` is where the program ends. 0 is returned for 0 errors.
- Code we add will go after the `{` and before the return 0;.

Additionally, C++ is case sensitive, so capitalization matters. Make sure you write main() in all lower case - Main() and MAIN() won't work!

~

2.5.3 Creating a program: Building source code into a program

Remember that your computer **expects to read binary 1010101010**, and does not understand English or other human languages. When we're ready to run our C++ program, we need to run it through a program called a **compiler**.

A **compiler** goes through your source code, generating binary code based on what you wrote. The end result is an **executable file**, also known as a **binary file**. This is the type of program where, if you double-click it on your computer, it will *run*.

The compiler program we are using in class is called `g++` (or `MinGW`, which is a Windows port of g++). When we're ready to test out our code, we can open up a **terminal** in VS Code and enter the command:

```
g++ main.cpp
```

We're telling the `g++` compiler program to convert our `main.cpp` source code file into an executable. By default, it will either generate `a.exe` (on Windows) or `a.out` (on Linux/Mac), but we can always rename our program, or we can tell g++ what to name our program in the same step:

```
g++ main.cpp -o MyProgram.exe
```

~

Once we're ready to run our program, we would then launch it from the terminal like this:

```
./MyProgram.exe
```

Though at the moment, this program does nothing except exit! :)

~

2.5.4 Displaying text to the screen

In order to display text to the screen with our program, we need to include a library. A library is a set of pre-written code that can be used across multiple programs. C++ contains a standard library of pre-written code we can use. To gain access to the cout (console-output) statement, we need to add this at the top of our C++ program file:

```
#include <iostream>
```

After that, we can use the cout command to display text like this:

```
std::cout << "Hello!";
```

The `std::` prefix stands for "STandarD", as in the C++ standard library. This command will display "Hello!" to the screen when the program is built and run.

You can also add empty lines to the program using the endl (end-line) command:

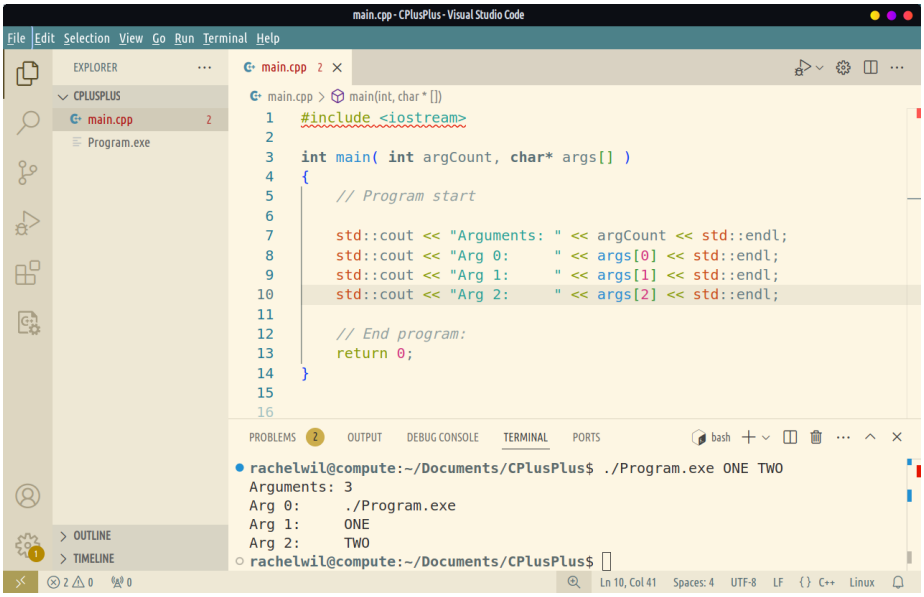
```
std::cout << std::endl;
```

And these commands can be chained together:

```
std::cout << "Hello!" << std::endl << "World!" << std::endl;
```

The << sign is known as the **output stream operator.**
~

2.5.5 Programs with arguments



Traditional command line programs may also take in program arguments as additional data to work with. We can also add program arguments to our programs. To do that, we need to add some additional information within main()'s parentheses - the parameter list:

```
int main( int argCount, char* arguments[] )
```

(Note: We can change the name of `argCount` and arguments to anything, so many programs use "`argc`" to stand for argument count and "`argv`" for the actual list of arguments.)

For the time being, don't worry about "int", "char*", or "[ ]", but we will learn about these later. The arguments item is essentially a list of input items passed to the program. argCount will also contain the amount of arguments passed in. So, if a program is run like this:

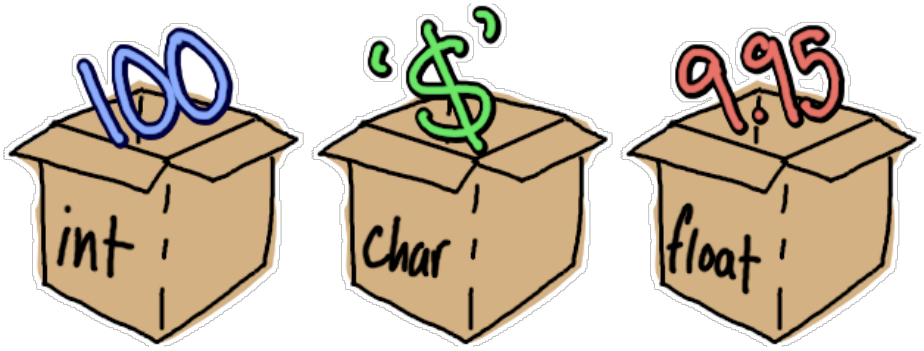
```
./program ONE two THREE
```

Then it would have 4 total arguments (the program name counts as one), with "ONE" being argument #1, "two" being argument #2, and "THREE" being argument #3.

~

We'll look at this more a little later after we cover variables.

2.6 Intro: Variables and data types



2.6.1 What are variables?

Variables are named locations where we can store information.

We use variables in programming as places to temporarily store data so that we can manipulate that data. We can have the user write to it via the keyboard, we can use files to write data to variables, we can do math operations or modify these variables, and we can print* them back out to the screen or a text file.(*Printing here meaning "display on the screen"; not to a printer.)

~

When we're writing programs in C++, we need to tell our program what the **data type** of each variable is, and give each variable a **variable name** (aka identifier). When a variable is **declared**, a space in RAM is allocated to that variable and it can store its data there. That data can be manipulated or overwritten later as needed in the program.

~

Some examples of things we might store as variables:

- A student's information:
  - `string studentName`, the name of a student
  - `float studentGpa`, the GPA of that student
  - `int creditHoursCompleted`, the total amount of credit hours the student has completed
  - `bool activeStudent`, whether the student is an active student or not (e.g., graduated?)
- A store website's item listing:
  - `string name`, the name of the product
  - `float price`, the price of the product
  - `int quantity`, how much of this product is in stock

~

2.6.2 Data types

In C++, when we want to create a variable we need to tell the **compiler** what the **data type** of the variable is. Data types specify **what** is stored in the variable (whole numbers? numbers with decimal values? text? true/false values?). Each data type takes up a different amount of space in memory.

Some of the common data types we'll use are:

Data type	**Values**	**Size**	**Example code**
char	single symbols - letters, numbers, anything	1 byte	`char currency = '$';`
boolean	`true` or `false`	1 byte	`bool savedGame = false;`
integer	whole numbers	4 bytes	`int age = 100;`
float	numbers w/ decimals	4 bytes	`float price = 9.95;`
double	numbers w/ decimals	8 bytes	`double price = 1.95;`
string	any text, numbers, symbols, any length	4 bytes	`string password = "123secure";`

~

- **Floats** and **doubles** both store numbers with fractional (decimal) parts, but a double takes up **double the memory in RAM**, allowing to store more accurate fractional amounts. A **float** has 6 to 9 digits of precision and a **double** has 15 to 18 digits of precision. (From https://www.learncpp.com/cpp-tutorial/floating-point-numbers/)
- **Boolean** data types store **true** and **false** values, but will also accept integer values when assigned. It will convert a value of **0** to **false** and any other number to **true**.
- **String** variables' values must be written within double quotes `"Hello!"`
- **Character** variables' values must be written within single quotes `'Z'`

~

1. **Additional information: Unsigned data types**

   Sometimes it doesn't make sense to store negative values in a variable - such as *speed* (which isn't directional, like velocity), the *size* of a list (can't have negative items), or *measurement* (can't have negative width). You can mark a variable as `unsigned` to essentially double its range (by getting rid of the negative side of values). For instance, a normal int can store values from -2,147,483,648 to 2,147,483,647, but if you mark it as unsigned, then your range is 0 to 4,294,967,295.

   ~

2.6.3 Declaring variables and assigning values to variables

1. **Variable declaration**

   When we're **declaring** a variable, it needs to follow one of these formats:

   Declaration forms

   - `DATATYPE VARIABLENAME;`
   - `DATATYPE VARIABLENAME = VALUE;`
   - `DATATYPE VAR1, VAR2, VAR3;`
   - `DATATYPE VAR1 = VALUE1, VAR2 = VALUE2;`
   - `DATATYPE VARIABLENAME{VALUE};` - Newer versions of C++

   The **data type** goes first, then the **variable name/identifier**, and if you'd like, you can also **assign a value** during the same step (though this is not required). Once a variable has been **declared**, you don't need to declare it again. This means you don't need to re-specify its data type when you're using it. Just address the variable by its name, and that's all.

   ~

   C++ source: Code that uses integers to figure out how many candies each kid should get:

   ```
   // Declaring variables and assigning values
   int totalCandies{10};
   int totalKids{5};
   int candiesPerKid{totalCandies / totalKids};

   // Reusing the same variables later on
   totalCandies = 100;
   totalKids = 10;
   candiesPerKid = totalCandies / totalKids;
   ```

   ~

   C++ source: Code to display the price plus tax to the user

   ```
   float price = 9.99;
   float tax = 0.11;
   // Text output to the screen. Can do math within!
   cout << "Please pay " << (price + price * tax);
   ```

   ~

   (a) **Additional information: Modern C++: `auto`**

   In C++11 (from 2011) and later, you can use the keyword `auto` as the "data type" in your variable declaration that includes an assignment statement. When you use `auto`, it uses the assigned value to automatically figure out the variable's data type, so you don't have to explicitly define it:

   C++ source: Usage of `auto` keyword in modern C++

```
auto price = 9.99;      // it's a double!
auto price2 = 2.95f;    // it's a float!
auto player = '@';      // it's a char!
auto amount = 20;       // it's an int!
```

~

2. **Variable assignment**

   Assignment forms

   - Stores the LITERAL value in VARIABLENAME:
     `VARIABLENAME = LITERAL;`
   - Copies the value from VARIABLENAME2 to VARIABLENAME1.:
     `VARIABLENAME1 = VARIABLENAME2;`

   When assigning a **value** to a **variable**, the variable being assigned to always goes on the left-hand side ("LHS") of the equal sign =. The = sign is known here as the **assignment operator**.

   The item on the right-hand side ("RHS") will be the value stored in the variable specified on the LHS. This can be a **literal** (a hard-coded value) or it can be a different variable of the same data type, whose value you want to copy over.

   ~

   C++ source: Assigning literal values to variables

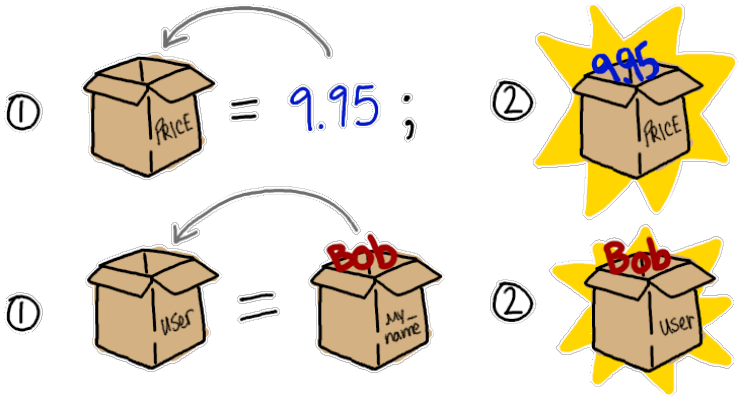
```
price = 9.99;       // 9.99 is a float literal
state = "Kansas";   // "Kansas" is a string literal
operation = '+';    // '+' is a char literal
```

   ~

   C++ source: Copying student3's value to the bestStudent variable

```
// Declare and initialize our variables
string student1 = "Hikaru";
string student2 = "Umi";
string student3 = "Fuu";
string bestStudent;

// Assignment statement
bestStudent = student3;
```

   ~



3. Variable copying

   If you use two variables in an assignment statement, like this:

   ```
   bestStudent = myName;
   ```

   This will copy the value in `myName` and store that copy in `bestStudent`.
   ~

4. Variable initialization

   When we **declare a variable and give it a value in the same line of code**, that is called *initialization*. The equal sign *can* be used in this regard:

   C++ source: Traditional C++ way of initializing variables

   ```
   string productName = "Candybar";
   float productPrice = 3.25;
   ```

   ~

   But with more modern C++, it's recommended that we use *braced initialization*:

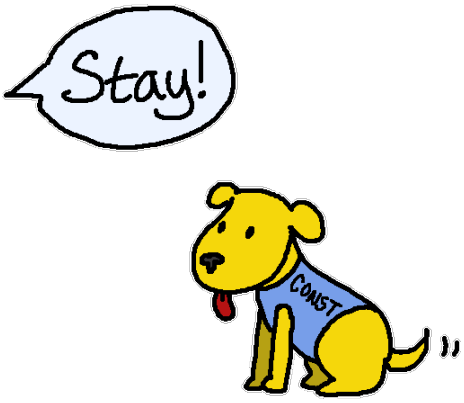
   C++ source: Modern C++ way of initializing variables

   ```
   string productName{ "Candybar" };
   float productPrice{ 3.25 };
   ```

   You might see both of these ways used to initialize variables in **starter code**.
   ~

   (a) Additional information: Garbage! If a variable is not *initialized* with some value, it will have some **garbage value** assigned to it - whatever was left over in the memory space where it was created. This data isn't useful to us, and usually shows up as random large numbers (12734827), so it's good to make sure to **initialize** your variables!
   ~

5. **Named constants**



Whenever you find yourself using a literal value in your assignment statements, you may want to think about whether you should replace it with a **named constant** instead.

Declaration form

- `const CONSTNAME = LITERAL;`

A named constant looks like a variable when you declare it, but it also has the keyword const - meaning that the value can't change after its declaration.

~

Let's say you wrote a program and hard-coded the tax rate:

C++ source: Hard-coded sales tax

```
// Somewhere in the code...
totalPrice = cartTotal + ( cartTotal * 0.0948 );

// Somewhere else in the code...
cout << 0.0948 << " sales tax" << endl;
```

~

Then the tax rate changes later on. You would have to go into your program and search for "0.0948" and update all those places! Instead, it would have been easier to assign the tax rate ONCE to a named constant and referred to that instead:

~

C++ source: Using named constant to store sales tax

```
// Beginning of program somewhere...
const SALES_TAX = 0.0948;

// Somewhere in the code...
```

```
totalPrice = cartTotal + (cartTotal * SALES_TAX);

// Somewhere else in the code...
cout << SALES_TAX << " sales tax" << endl;

~
```

If you ever find yourself using the same literal multiple times in your program, then perhaps consider replacing it with a named constant.
~

2.6.4 Naming conventions

While you can *generally* name a variable whatever you'd like, you still need to adhere to the syntax rules of the C++ language. In particular:

- Names can only contain **letters (upper- and lower-case)**, **numbers**, and **underscores**.
- **Spaces are not allowed in a variable name!**
- Names cannot start with a number.
- A name cannot be a **keyword** in C++, a name reserved for something else in the language, such as `void`, `if`, `int`, etc.

There are also **naming conventions** used so that variables (and later on functions and structs/classes) have a consistent naming scheme:

- Variables:
  - One common style is `lowerFirstLetterThenUpperCase` for variable names - this is called **Camel Case**.
  - Another style you might see in C++ code is `all_lower_with_underscores`.
  - You might see both of these styles in my starter code or examples, but I try to stick to *one style* for a program. You should also choose one and stay consistent in each program.
- Named constants:

- It is customary to give your named constants names in ALL CAPS, using underscores (_) to separate words. For example: `TOTAL_STUDENTS`, `OP_TAX_RATE`, etc.

Everything in C++ is **case sensitive**, which means if you name a variable `username`, it will be a different variable from one named `userName` (or if you type "userName" when the variable is called "username," the compiler will complain at you because it doesn't know what you mean).

~

2.6.5 Basic operations on variables

Now that you have variables available in your program to play with, what can you even do with them?

1. **Outputting variable values**

   Using the `cout` (console-out) statement, you can display text to the screen with string literals:

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

   But you can also display the values stored within variables, simply by using the variable's name:

   ```
   cout << myUsername << endl;
   ```

   You can also chain as many literals, variables, and commands together as long as they have the output stream operator `<<` in-between.

   C++ source: Displaying multiple string literals and variable values in one `cout` statement

   ```
   cout << "Student: " << name
        << endl << "GPA: " << gpa
        << endl << "Year: " << year << endl << endl;
   ```

   ~

2. **Inputting variable values**

   We can use the `cin` (console-in) statement to get the user to enter something on the keyboard and store that data into a variable:

   C++ source:

   ```
   float gpa;
   cin >> gpa;
   ```

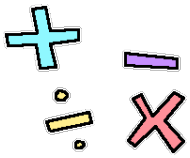
   `cin >>` can be used on strings, but it will not read anything after a space. Instead, we use the `getline` function to read whole lines of text into `string` variables:

   C++ source:

```
string oneLine;
getline( cin, oneLine );

string oneWord;
cin >> oneWord;
```

There will be more information on input and output in a later section.
~

3. **Math operations**



With variables with numeric data types (ints, floats, doubles), we can do arithmetic with the +, -, *, and / operators.

Symbol	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Make sure to put the result somewhere! When you do a math operation and you want to use the result elsewhere in the program, make sure you're storing the result in a variable via an assignment statement! If you just do this, nothing will happen:

```
totalCats + 1;
```

You can use an assignment statement to store the result in a new variable:

```
newCatTotal = totalCats + 1;
```

Or overwrite the variable you're working with:

```
totalCats = totalCats + 1;
```

~

4. **Compound operations:** There are also shortcut operations you can use to quickly do some math and overwrite the original variable. This works with each of the arithmetic operations:

   C++ source: Long-way and short-way adding onto a variable

```
// Long way:
totalCats = totalCats + 5;

// Compound operation:
totalCats += 5;
```

   Other shorthand:

Description	Long version	Short 1	Short 2
Increment	x = x + 1;	x += 1;	x++; OR ++x;
Increment	x = x + 2;	x += 2;	
Decrement	x = x - 1;	x -= 1;	x–; OR –x;
Decrement	x = x - 2;	x -= 2;	
Multiply	x = x * 2;	x *= 2;	
Divide	x = x / 2;	x /= 2;	

   ~

5. **String operations**

   Strings have some special operations you can do on them. You can also see a list of functions supported by strings here: C++ String Reference (We will cover more with strings in a later part).

   (a) **Concatenating strings**

   You can use the + symbol to combine strings together. When used in this context, the + sign is called the concatenation operator.

   C++ source: Concatenating type and food together into the order string

```
string type = "pepperoni";
string food = "pizza";

// Creates the string "pepperoni pizza"
string order = type + " " + food;
```

   ~

   (b) **Letter-of-the-string**

   You can also use the subscript operator `[ ]` (more on this when we cover arrays) to access a letter at some position in the string. Note that in C++, the position starts at 0, not 1.

   C++ source: Getting separate letters out of a string

```
string food = "pizza";

char letter1 = food[0];  // Stores 'p'
char letter2 = food[1];  // Stores 'i'
char letter3 = food[2];  // Stores 'z'
char letter4 = food[3];  // Stores 'z'
char letter5 = food[4];  // Stores 'a'
~
```

2.6.6 Review questions:

1. Before a variable can be used, it must be...
2. "LHS" stands for...
3. "RHS" stands for...
4. The "assignment operator" is...
5. A "literal" is...
6. How do you write a **variable declaration**?
7. How do you write a **variable initialization**?
8. How do you write a **variable assignment**?
9. What does the `auto` keyword do?
10. What does marking a variable `unsigned` mean?
11. What does **case sensitive** mean?

2.7 Intro: cout - Console output

2.7.1 The `iostream` library

A **library** is a set of prewritten code that we can use in our programs. C++ has a set of libraries that come with it, and third parties can also create libraries. But anyway, in order to be able to display output to the screen we will need to include one of the built-in C++ libraries - `iostream`.

At the top of our source code file, we will have to add an `#include` statement to add this library to our program:

```
#include <iostream>
```

This will allow us to use the `cout` and `endl` commands from the `std` (STandarD) library:

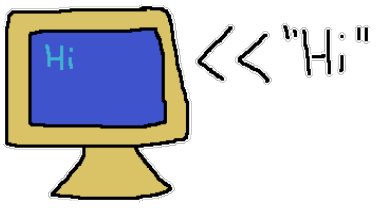
```
std::cout << "HI!" << std::endl;
```

But we can also simplify our code by adding in `using namespace std;` beneath our library includes, so that we don't have to prefix `std::` on all of our iostream commands:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "HI!" << endl;
    return 0;
}
```

~

2.7.2 Outputting Information with `cout`



The `cout` command (pronounced as *"c-out"* for "console-out") is used to write information to the screen. This can include outputting a string literal:

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

or a variable's value:

```
cout << yourName << endl;
```

or stringing multiple things together:

```
cout << "Hello, " << yourName << "!" << endl;
```

In C++, we use the **output stream operator** << to string together multiple items for our output.

1. Newlines with endl

   The `endl` command stands for *"end-line"* and ensures there is a vertical space between that `cout` statement and the next one. For example, if we write two `cout` statements without endl like this:

   C++ source: cout statement without endl

   ```
   cout << "Hello";
   cout << "World";
   ```

   Program output:

   ```
   HelloWorld
   ```

   If we want to separate them on two different lines, we can write:

   C++ source: cout statement with endl

   ```
   cout << "Hello" << endl;
   cout << "World";
   ```

   Program output:

   ```
   Hello
   World
   ```

   Remember that in C++, a statement ends with a semicolon ;, so you can split your `cout` statement across multiple lines, as long as you're chaining items together with << and only adding a `;` on the last line:

   C++ source: Displaying several string literals and variable values

   ```
   cout << "Name:   " << name
        << "Age:    " << age
        << "State:  " << state << endl;
   ```

   ~

2. Good user interfaces

One of the important parts of writing software is making sure users know how to navigate and use your program. While writing your programs, assume that the person running it knows nothing about this program. Make sure everything they need to know about the program is given to them as instructions, and make sure menus are easy to read.
~

3. Drawing a line of characters across the screen

   We can use the `string` library to create a line of characters with some width, without having to type out all those characters ourselves. Its form looks like this:

   ```
   cout << string( AMOUNT, CHAR ) << endl;
   ```

   C++ source: Displaying text with double quotes in it

   ```
   cout << "MAIN MENU" << endl << string( 20, '-' ) << endl;
   ```

   Program output:

   ```
   MAIN MENU
   --------------------
   ```

   ~

4. Escape Sequences

   There are special characters, called escape sequences, that you can use in your `cout` statements:

Character	Description
`\n`	newline (equivalent to `endl`)
`\t`	tab
`\"`	double quote

   C++ source: Using the newline character

   ```
   cout << "\"hello\nworld\"" << endl;
   ```

   Program output:

```
"hello
world"

~
```

C++ source: Using the tab character in outputs

```
cout << "A\tB\tC" << endl;
cout << "1\t2\t3" << endl;
```

Program output:

```
A       B       C
1       2       3

~
```

C++ source: Displaying text with double quotes in it

```
cout << "He said \"Hi!\" to me!" << endl;
```

Program output:

```
He said "Hi!" to me!

~
```

5. Formatting currency

   By default, the following code:

   ```
   float price = 10.90;
   cout << "Price: $" << price << endl;
   ```

   Will give us output like this:

   ```
   Price: $10.9
   ```

   To get our program to always display two numbers to the right of the decimal place, we need to include the iomanip (input/output manipulation) library:

   ```
   #include <iomanip>
   ```

   And then add this command to our program somewhere:

   ```
   cout << fixed << setprecision( 2 );
   ```

   Then, our program's output will look like this:

```
Price: $10.90
```

~

6. Tables and columns

If we want to make a table in our program, we need to use columns with fixed widths. We will need to #include <iomanip> again.

We can specify the width of a column for the following item being displayed with the setw command, like this:

```
cout << setw( 5 ) << "ID";
```

This will give the following column a width of 5. The text "ID" takes up 2 spaces, so then there are 3 empty spaces left over. cout statements afterwards will start at the end of this column.

For a table header, I usually write it like this:

```
cout << left
  << setw( 5 ) << "ID"
  << setw( 20 ) << "STUDENT"
  << setw( 6 ) << "GPA"
  << endl << string( 40, '-' ) << endl;
```

This sets the text alignment to left-aligned (default is right-aligned), then creates three columns: ID, Student, and GPA.

Then, further down in the table, I can use the same setw values for the data:

```
cout
  << setw( 5 ) << "2"
  << setw( 20 ) << "Usagi Tsukino"
  << setw( 6 ) << "2.8"
  << endl;
```

This will create a nice table that is easy for the user to read.

~

2.7.3 Inputting Information with `cin`

When we want the user to enter a value for a variable using the keyboard, we use the `cin` command (pronounced as "c-in" or "console-in").

For variables like int and float, you will use this format to store data from the keyboard into the variable:

1. Using `cin >>` for variables

   C++ source: Reading input into one variable

   ```
   cin >> VARIABLENAME;
   ```

   You can also chain cin statements together to read multiple values for multiple variables:

   C++ source: Inputting data into multiple variables at once (separated by spaces)

   ```
   cin >> VARIABLENAME1 >> VARIABLENAME2 >> ETC;
   ```

   ~

2. Strings and `cin >>`

   When using `cin >>` with a string variable, keep in mind that it will only read until the first whitespace character, meaning it can't capture spaces or tabs. For example:

   C++ source: Getting one word (ends at spaces) with the input stream operator

   ```
   string name;
   cin >> name;
   ```

   If you enter "Rachel Singh", name will contain "Rachel". To capture spaces, you need to use a different function.

   ~

3. Using `getline(cin, var);` for Strings

   You can use the getline function to capture an entire line of text as a string. This is useful when you want to capture spaces and multiple words. For example:

   C++ source: Getting a full line of text (including spaces) with the getline function

   ```
   string name;
   getline(cin, name);
   ```

   ~

4. Mixing `cin >> var;` and `getline(cin, var);`

   If you mix `cin >> var;` and `getline(cin, var);`, you might encounter issues with the input buffer. To avoid this, use `cin.ignore();` before `getline(cin, var);` if you used `cin >> var;` before it.

```
int number;
string text;

cin >> number;
cin.ignore();
getline( cin, text );
```

   ~

2.7.4 Review questions:

1. What is a "console" (aka "terminal")?
2. What does "`cout`" stand for?
3. The "`endl`" command is used for...
4. The `\t` command is used for...
5. The `\n` command is used for...

2.8 Intro: Program arguments

2.8.1 Program arguments

Some command line programs take in arguments when you run them. For example: `ping yahoo.com` . The `ping` program takes in a URL as an argument, then the program will launch and attempt to send packets of information to the URL and see if there's a response.

We can also write comamnd line programs with arguments. To do this, we need to add a couple of arguments to main:

```
int main( int argCount, char* args[] )
{
}
```

In this case, any text given after the program name will be stored in `args[1]` and after (the program name is in `args[0]`.) `char* args[]` is an array of c-style strings, though we can convert them into other data types. The `int argCount` tells us how many arguments were passed into the program. If the user doesn't pass in enough arguments, we could display an error message and what we expect the arguments to be.

1. Converting arguments to different types

   We can convert the arguments to different data types. The following code snippet shows examples:

   ```
   int main( int argCount, char* args[] )
   {
       if ( argCount < 5 )
       {
           cerr << "Not enough arguments!" << endl;
           return 1;
       }

       int myInteger = stoi( args[1] );
       float myFloat = stof( args[2] );
       string myString = string( args[3] );
       char myChar = args[4][0];
   }
   ```

2.9 Lab: Variables and output

3 Week 2: Branching and testing

3.1 Intro: Boolean logic

3.1.1 Introduction

Computers are all about **binary** - 1's and 0's, on and off, true and false. A large part of writing logic with our programs is all about asking "yes/no" questions. If "criteria" is true, then execute this code... if it's false, execute that code. In order to better understand how branching works in our programs, it's important to learn about **boolean logic**.

"Boolean" refers to something that is either **true** or **false**. This can be the values "true" and "false" itself, or it can be an expression that evaluates to true or false. For example:

- if the user clicked "save", then...
- if the account balance is less than 0, then...
- if the "done" variable is set to true, then...

3.1.2 True/false questions in programming

Question	**Represented as C++ code**
Is x equal to y?	x == y
Is x not equal to y?	x != y
Is x greater than y?	x > y
Is x greater than or equal to y?	x >= y
Is x less than y?	x < y
Is x less than or equal to y?	x <= y
Is x true?	x
Is x false?	!x

x and **y** can be swapped out with different types of data. We can check if two strings of text are equal, or if two numbers are equal. With the bottom two examples, "is x?" and "is not x?" applies for if **x** is a **boolean variable** - a variable that is storing a value of either true or false.

```
if ( i_have_not_been_fed && it_is_6_am ) {
  MakeLotsOfNoise();
}
```

3.1.3 Compound operations with logic operators

We can also create **compound boolean expressions** from separate boolean expressions using **logic operators**. Logic operators are AND (&&), OR (||), and NOT (!).

- Is $x == y$ and $x == z$?
  - In C++ code: `x = y && x = z`
  - True if both sub-expressions are true (x==y and x==z).
  - False if one or both sub-expressions are false.

Example:

```
if ( wantsBeer && isAtLeast21 && likesBeer ) {
    GiveBeer();
}
```

- Is $x == y$ or $x == z$?
  - In C++ code: `x = y || x = z`
  - True if at least one sub-expression is true.
  - False if both sub-expressions are false.

Example:

```
if ( isBaby || isSenior || isVet )
{
    GiveDiscount();
}
```

- Is x false?
  - In C++ code: `!x`
  - True if the sub-expression is false.
  - False is the sub-expression is true.

Example:

```
// Just checking documentSaved is true
if ( documentSaved ) {
    Quit();
}

// Checking if documentSaved is false
if ( !documentSaved ) {
    Save();
    Quit();
}
```

3.1.4 Boolean variables and relational operations

We can have **boolean variables** that store **true** or **false**, and we can then do tests on those values...

```
bool quit = false;

// ( pretend there's more code here )

if ( quit ) // If quit is true
{
    return 0; // exit
}
```

But we can also create **boolean expressions** by utilizing those **relational operators** to compare two values together.

```
float balance;
cout << "Enter your bank balance: ";
cin >> balance;

if ( balance < 0 ) // balance < 0 will return true or false
{
    cout << "OVERDRAWN!" << endl;
}
```

These **relational operators** can also work on **strings**, where < and > will check alphabetical order (sort of).

```
string student1, student2;

cout << "Enter student 1 name: ";
getline( cin, student1 );

cout << "Enter student 2 name: ";
getline( cin, student2 );

if ( student1 < student2 )
{
    cout << "Student 1 goes first" << endl;
}
else
{
    cout << "Student 2 goes first" << endl;
}
```

Note that when using < and > with strings or chars, it will compare them based on alphabetical order - *if* they're both the same case (both uppercase or both lowercase). The code 65 represents `'A'`, but the code 97 represents `'a'`, so items will be "sorted" based on these codes.

3.1.5 Truth tables

We can use **truth tables** to help us visualize the logic that we're working with, and to validate if our assumptions are true. Truth tables are for when we have an expression with more than one variable (usually) and want to see the result of using ANDs, ORs, and NOTs to combine them.

1. **Truth table for NOT:**

   On the top-left of the truth table we will have the boolean variables or expressions written out, and we will write down all its possible states. With just one variable p, we will only have two states: when it's true, or when it's false.

   On the right-hand side (I put after the double lines) will be the result of the expression - in this case, "not-p" $!p$. The not operation simply takes the value and flips it to the opposite: true $\rightarrow$ false, and false $\rightarrow$ true.

p	!p
T	F
F	T

2. **Truth table for AND:** For a truth table that deals with two variables, p and q, the total amount of states will be 4:

   (a) p is true and q is true.

   (b) p is true and q is false.

   (c) p is false and q is true.

   (d) p is false and q is false.

`p`	`q`	`p && q`
T	T	T
T	F	F
F	T	F
F	F	F

   This is just a generic truth table. We can replace the values with boolean expressions in C++ to check our logic.

3. **Truth table for OR:** This also deals with 2 variables, p and q, so 4 total states. If *at least one* of the two variables are **true**, then the entire expression is **true**. An expression with an "or" is only **false** when **all sub-expressions are false**.

`p`	`q`	`p || q`
T	T	T
T	F	T
F	T	T
F	F	F

4. **Example - AND:**

   We will only quit the program if the game is saved and the user wants to quit:

`game_saved`	`want_to_quit`	`game_saved && want_to_quit`
T	T	T
T	F	F
F	T	F
F	F	F

   The states are:

   (a) **T: `gameSaved` is true and `wantToQuit` is true:** Quit the game.

   (b) **F: `gameSaved` is true and `wantToQuit` is false:** The user doesn't want to quit; don't quit.

(c) **F: `gameSaved` is false and `wantToQuit` is true:** The user wants to quit but we haven't saved yet; don't quit.

(d) **F: `gameSaved` is false and `wantToQuit` is false:** The user doesn't want to quit and the game hasn't been saved; don't quit.

5. **Example - OR:**

   If the store has cakes OR ice cream, then suggest it to the user that wants dessert.

`hasCake`	`hasIcecream`	`hasCake` ‖ `hasIceCream`
T	T	T
T	F	T
F	T	T
F	F	F

   The states are:

   (a) **T: `hasCake` is true and `hasIcecream`** is true: The store has desserts; suggest it to the user.

   (b) **T: `hasCake` is true and `hasIcecream`** is false: The store has cake but no ice cream; suggest it to the user.

   (c) **T: `hasCake` is false and `hasIcecream`** is true: The store has no cake but it has ice cream; suggest it to the user.

   (d) **F: `hasCake` is false and `hasIcecream`** is false: The store doesn't have cake and it doesn't have ice cream; don't suggest it to the user.

3.1.6 DeMorgan's Laws

Finally, we need to cover DeMorgan's Laws, which tell us what the *opposite* of an expression means.

For example, if we ask the program **"is a $>$ 20?"** and the result is **false**, then what does this imply? If $a > 20$ is false, then $a \leq 20$ is true... notice that the opposite of "greater than" is "less than OR equal to"!

1. **Opposite of `a && b`**

`a`	`b`	`a && b`	`!( a && b )`	`!a` ‖ `!b`
T	T	T	F	F
T	F	F	T	T
F	T	F	T	T
F	F	F	T	T

If we're asking "is a true and is b true?" together, and the result is **false**, that means either:

(a) a is false and b is true, or

(b) a is true and b is false, or

(c) a is false and b is false.

These are the states where the result of `a && b` is false in the truth table.

In other words,

$$\texttt{!(a \&\& b)} \equiv \texttt{!a} \parallel \texttt{!b}$$

If a is false, or b is false, or both are false, then the result of `a && b` is false.

2. **Opposite of `a || b`:**

a	b	a ‖ b	!(a ‖ b)	!a && !b
T	T	T	F	F
T	F	T	F	F
F	T	T	F	F
F	F	F	T	T

If we're asking "is a true or b true?", if the result of that is **false** that means only one thing:

(a) Both a and b were false.

This is the only state where the result of `a || b` is false.

In other words,

$$\texttt{!(a} \parallel \texttt{b)} \equiv \texttt{!a \&\& !b}$$

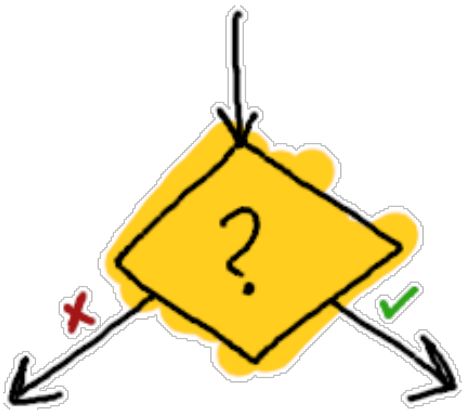
`a || b` is false only if a is false AND b is false.

3. **Summary**

In order to be able to write **if statements** or **while loops**, you need to understand how these boolean expressions work. If you're not familiar with these concepts, they can lead to you writing **logic errors** in your programs, leading to behavior that you didn't want.

3.2 Intro: If, else if, else statements

3.2.1 Branching - this way or that?



Branching is one of the core forms of **controlling the flow of the program**. We ask a question, and based on the result we perhaps do *this code over here* or maybe *that code over there* - the program results change based on variables and data accessible to it. We will mostly be using **if / else if / else statements**, though **switch** statements have their own use as well. Make sure to get a good understanding of how each type of branching mechanism "flows".

1. **Yes or no? - If/Else statements**

   The computer only understands "yes/no" questions - questions that either result in **true** or **false**. Based on these outcomes, we can choose to execute one set of code or another.

   Form of an if/else statement:

   ```
   // Do things 1

   if ( CONDITION )
   {
     // Do things 2-a
   }
   else
   {
     // Do things 2-b
   }

   // Do things 3
   ```

   The **else** block is executed when the **if** check fails. If the CONDITION is false, we can use **DeMorgan's Laws** to figure out what the program's state is during the **else**.

   NOTE: The else statement NEVER has a CONDITION.

Example C++ source: Displaying "can't vote" if age is less than 18, or "can vote" otherwise

```
cout << "Enter your age: ";
cin >> age;

if ( age < 18 )
{
  result = "can't vote";
}
else
{
  result = "can vote";
}

cout << "Result: " << result << endl;
```

If the `age` is less than 18, it will set the `result` variable to "can't vote". If that boolean expression is **false**, that means `age` is ≥ 18, and then it will set the `result` to "can vote". Finally, either way, it will display "Result: " with the value of the `result` variable.

2. **Standalone If statement** For a basic **if statement**, we use the syntax:

```
// Do things 1

if ( CONDITION )
{
  // Do things 2-a
}

// Do things 3
```

The CONDITION will be some **boolean expression**. If the boolean expression result is **true**, then any code within this if statement's **code block** (what's between `{` and `}`) will get executed. If the result of the expression is **false**, then the entire if statement's code block is skipped and the program continues.

Example: C++ source: Displaying "(OVERDRAWN!)" only if the bank balance is less than 0

```
cout << "Bank balance: " << balance;

if ( balance < 0 )
{
  cout << " (OVERDRAWN!)";
}

cout << " in account #" << accountNumber << endl;
```

Program output: Output with `balance = -50`

```
Bank balance: -50 (OVERDRAWN!) in account #1234
```

Program output: Output with `balance = 100`

```
Bank balance: 100 in account #1234
```

- The special message "OVERDRAWN!" only gets displayed when the balance is less than 0.
- The following `cout` that gives the account number is displayed in all cases.

3. **If/Else if/Else statements**

   In some cases, there will be multiple scenarios we want to search for, each with their own logic. We can add as many **else if** statements as we want - there must be a starting **if** statement, and each **else if** statement will have a condition as well.

   We can also end with a final **else** statement as a catch-all: if none of the previous if / else if statements are true, then **else** gets executed instead. However, the else is not required.

   Form of an if/else if/else statement

```
// Do things 1
if ( CONDITION1 )
{
  // Do things 2-a
}
else if ( CONDITION2 )
{
  // Do things 2-b
}
else if ( CONDITION3 )
{
  // Do things 2-c
}
else
{
  // Do things 2-d
}
// Do things 3
```

   Example: Displays a letter grade based on range, 90 and above is an 'A', and so on...

```
cout << "Enter grade: ";
cin >> grade;

if      ( grade >= 90 ) { letterGrade = 'A'; }
else if ( grade >= 80 ) { letterGrade = 'B'; }
else if ( grade >= 70 ) { letterGrade = 'C'; }
else if ( grade >= 60 ) { letterGrade = 'D'; }
else                    { letterGrade = 'F'; }

cout << "Grade: " << letterGrade << endl;
```

(a) I'm first checking if the grade is 90 or above. If it is, then I know the letter grade is `'A'`.

(b) However, if it's false, it will go on and check the next **else if** statement. At this point, I know that since ***grade*** $>=$ ***90*** **was FALSE**

(c) that means: ***grade*** $<$ ***90*** **is TRUE.**

(d) Since the next statement checks if ***grade*** $>=$ ***80***

(e) if this one is TRUE, I know that: ***grade*** $<$ ***90*** **AND** ***grade*** $>=$ ***80***

3.2.2 Nesting if statements

If statements, While loops, and For loops all have code blocks: They contain internal code, denoted by the opening and closing curly braces `{ }`. Within any block of code you can continue adding code. You can add if statements in if statements in if statements, or loops in loops in loops.

Nesting an if statement within another if statement basically gives you a boolean expression with an AND.

Example:

```
if ( wantsToVote )
{
  if ( age >= 18 )
  {
    AllowToVote();
  }
}
```

Equivalent logic:

```
if ( wantsToVote && age >= 18 )
{
  AllowToVote();
}
```

Whether you implement some logic with nested if statements, or with if / else if statements using AND operations is a matter of design preference- in some cases, one might be cleaner than the other, but not always.

If you have a statement like this:

```
if ( conditionA )
{
  if ( conditionB )
  {
    Operation1();
  }
  else
  {
    Operation2();
  }
}
```

It could be equivalently described like this:

```
if ( conditionA && conditionB )
{
  Operation1();
}
else if ( conditionA && !conditionB )
{
  Operation2();
}
```

3.2.3 Logic operators with if statements

We can also combine multiple conditions together using our logic operators:

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

If we have an if statement with two conditions combined with an AND, then the if statement is only executed if all sub-conditions are true:

```
if ( CONDITION1 && CONDITION2 )
    DoThingOne(); /* All conditions were true */
else
    DoThingTwo(); /* One or more conditions were false */
```

If we have an if statement with two conditions combined with an OR, then the if statement is executed if at least one sub-condition is true:

```
if ( CONDITION1 || CONDITION2 )
    DoThingOne(); /* One or more conditions were true */
else
    DoThingTwo();/* All conditions were false */
```

3.3 Intro: Switch statements

Switch statements are a special type of branching mechanism that **only** checks if the value of a variable is **equal to** one of several values. Switch statements can be useful when implementing a menu in a program, or something else where you only have a few, finite, discrete options.

In C++, switch statements only work with primitive data types, like integers and chars - not strings.

```
switch ( VARIABLE )
{
 case VALUE1:
   // Do thing
 break;

 case VALUE2:
   // Do thing
 break;

 default:
  // Default code
}
```

With a switch statement, each **case** is one equivalence expression. The `default` case is executed if none of the previous cases are.

```
case VALUE1: // equivalent to: if ( VARIABLE == VALUE1 )

case VALUE2: // equivalent to: if ( VARIABLE == VALUE2 )

break: // required at the end of a case statement (unless...)

default: // equivalent to: else
```

The default case is not required, just like how the **else** clause is not required in an if statement.

The `break;` statement

The end of each case should have a `break;` statement at the end. If the `break` is not there, then it will continue executing each subsequent case's code until it *does* hit a break.

This behavior is "flow-through", and it can be used as a feature if it matches the logic you want to write.

Variables inside cases

If you're **declaring a variable** within a **case** statement, you need to enclose your case with curly braces `{ }`, otherwise you'll get compiler errors:

```
switch ( operation )
{
 case 'A':
   {
     float result = num1 + num2;
     cout << "Result: " << result << endl;
   }
  break;
}
```

Calculator example:

Perhaps you are implementing a calculator, and want to get an option from the user: (A)dd, (S)ubtract, (M)ultiply, or (D)ivide. You could store their choice in a `char` variable called `operation` and then use the `switch` statement to decide what kind of computation to do:

C++ source: Basic calculator

```
switch ( operation )
{
 case 'A': // if ( operation == 'A' )
  result = num1 + num2;
  break;

 case 'S': // else if ( operation == 'S' )
  result = num1 - num2;
  break;

 case 'M': // else if ( operation == 'M' )
  result = num1 * num2;
  break;

 case 'D': // else if ( operation == 'D' )
  result = num1 / num2;
  break;
}

cout << "Result: " << result << endl;
```

Flow-through example:

Sometimes you want to check if your variable equals a value "x" or "y", and execute the same code for each. You can use flow-through in this case. If no `break;` is given for a specific `case`, then it will continue to execute code for the following `case` until a `break;` is found.

```
char choice;
cout << "Do you want to quit? (Y/N): ";
cin >> choice;
```

```
switch ( choice )
{
 case 'Y':
 case 'y':
  done = true;
  break;

case 'N':
case 'n':
  done = false;
  break;

default:
  cout << "Unknown selection!" << endl;
}
```

3.4 Intro: Testing

3.4.1 How do we test?

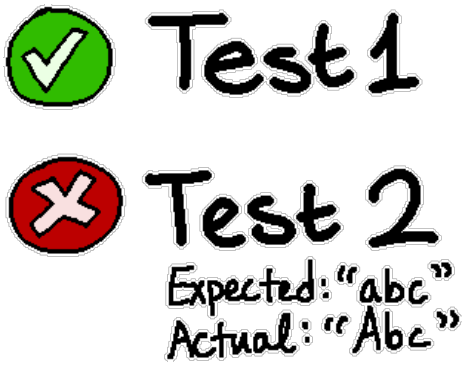
How do you know that your program actually works? Especially as it gets more complex?

Software development skills mean more than just writing code. It also includes knowing how to test your code, to prove that it is working as intended.

For the most basic tests, we investigate a portion of code and ask, "Given some inputs, what are the expected outputs?" and "If I run the program with these **inputs**, what are the **actual outputs**?", and finally, "Does my **actual output** match my **expected output**?"

~

3.4.2 Writing tests first?

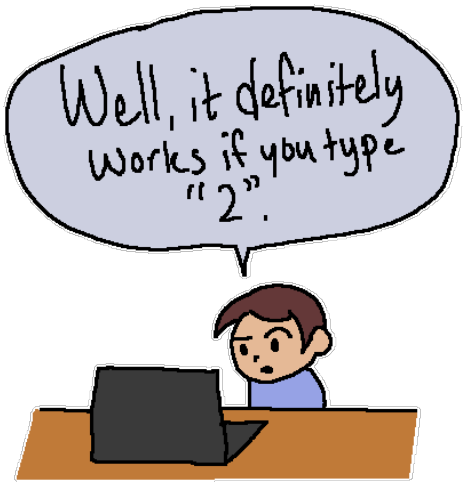


Notice that you don't actually need to know *what* is in the *function* before you write tests for it. Knowing what the program is *supposed to do*, you can figure out a set of inputs and outputs that it ought to have.

Often, it is very useful to write your **test cases** prior to actually writing any code. This helps you solidify what exactly he program is supposed to do. Plus, if you write your test cases afterwards, it is a bit like reading your own essay - your brain will skip over errors because it *knows what you meant*, and doesn't actually read the actual words (or code).

~

3.4.3 Multiple test cases



It isn't good enough to write one single test case. Often, you will want to have enough test cases to check for as many possibilities as possible - though we can't write an infinite amount. So sometimes, it's best to just write tests for reasonable outcomes, including potential errors.

When we eventually write code to do our testing for us, we can have the program keep an "ear out" for an error happening - and, sometimes we want that error to occur. Such as if the user enters a negative deposit amount, we would want the program to notice and send an error code or error message. We could also write our tests to make sure the error occurs, rather than allowing the user to deposit (or withdraw) a "negative" amount of money!

~

3.4.4 Example 1

Let's say your coworker Maryam is writing code for a new feature and you're writing the tests. You both have the requirements, so you can both work at the same time.

For the program, it takes in a list of grades (4.0 being 'A', 3.0 being 'B', 2.0 being 'C', 1.0 being 'D', and 0.0 being 'F') and calculates a grade point average. (Averages are calculated as the **sum of all the grades** divided by **the amount of grades**.)

~

3.4.5 Example 2

Now your coworker André is working on another part of the program, and you're writing the tests.

The program takes in the price of a given textbook title, and an amount of that textbook to purchase, and returns the total cost of purchasing those textbooks.

~

3.4.6 Reminder

Takeaways:

- A single test case consists of inputs and their expected output(s).
- Running the program with the test's inputs will give us the actual output(s).
- A test will pass if the actual output matches the expected output.
- A test will fail if the actual output does not match the expected output.
- Having multiple test cases help you verify that your code works properly.

3.5 Lab: Branching and testing

3.5.1 Assignment information

- **Turning in your work:**
  1. In VS Code, build and run the graded program(s) in the **Terminal**. Take a screenshot of the program running and save it to your computer.
  2. In VS Code, make sure to COMMIT and SYNC your changes to the GitLab repository. (See: Using Git and VS Code).
  3. Go to your GitLab repository page, make sure it shows your latest COMMIT MESSAGE and your code is up-to-date. (Important!! I can't build and run your code if it's not on the server!!)
  4. In Canvas, go to the **Assignment** and click **Start Assignment**. Upload the screenshot of your program running. In the comments, let me know if you're done with all of the lab or just part of it and if you have any questions.
- **Don't utilize future topics!** - We will revisit the graded program here once we cover Exceptions later on. Please do not use exceptions for this assignment.
- **Practice programs & graded programs:** Completing the graded program(s) is worth 90% of the lab grade. Completing the practice programs is worth an additional 10%.
- **Links:**
  - How to turn in assignments on Canvas
  - Using Git and VS Code
  - Program arguments
  - Assignment direct link

3.5.2 Included files:

Getting started:

1. This lab is located in your **repository folder** under `wk02_BranchingAndTesting`.
2. In VS Code use "Open Folder" to open the specific subprogram you wish to work with.
3. BUILD: `g++ *.cpp` for programs with just a cpp file.
4. RUN: `./a.out` (Linux/Mac) or `./a.exe` (Windows).

```
wk02_BranchingAndTesting
 graded_program1
    apartment.cpp
    testcases.txt
```

```
   tester.cpp
graded_program2
   shop.cpp
   testcases.txt
   tester.cpp
instructions.org
practice1_if
   bank.cpp
   bank-testcases.txt
practice2_ifelse
   isneg.cpp
   isneg.txt
practice3_ifelseif
   grade.cpp
   grade-testcases.txt
practice4_switch
   trans.cpp
   trans-testcases.txt
practice5_compound
   shop.cpp
   shop-testcases.txt
practice6_testing
    calc.cpp
    calc-testcases.txt
```

3.5.3 Practice programs

1. Practice 1 - If

   Example output:

   ```
   $ ./bank.exe 50 25
   -- BANK BALANCE PROGRAM --
   Balance: $-25.00 (OVERDRAWN)
   -- GOODBYE! --
   ```

   For this practice program I will describe the program's behavior. You can implement the program and build it. Reference the `bank-testcases.txt` to view the list of **test cases**, run the program for each of those tests and check the inputa gainst the expected output. If any tests fail, double check your code for logic errors. Once they all pass, make sure to update `bank-testcases.txt` to indicate that they're psasing.

   Test cases:

   ```
   TEST    INPUTS                      EXPECTED OUTPUT               PASSES?
   1.      ./bank.exe 50 100           Balance: $50
   2.      ./bank.exe 25 10            Balance: $-15 (OVERDRAWN)
   ```

Program logic:

The program input arguments are `withdraw`, how much money to withdraw from the bank account, and `balance` how much money is in the account.

(a) Calculate how much money will be left in the account after the withdraw takes place. Use `withdraw` and `balance` variables, and store the result in the `remaining` variable.

(b) Display "Balance: $" and the `remaining` amount.

(c) If the account is now overdrawn, also display "(OVERDRAWN)".

Build and run

```
g++ bank.cpp -o bank.exe

./bank.exe WITHDRAW BALANCE
```

---

2. Practice 2 - If, else

Example output:

```
$ ./isneg.exe 5
-- NEGATIVITY DETECTOR --
POSITIVE OR ZERO
-- GOODBYE! --

$ ./isneg.exe -10
-- NEGATIVITY DETECTOR --
NEGATIVE
-- GOODBYE! --
```

Test cases:

After implementing the program go through each of these test cases in `isneg-testcases.txt`. Mark each one off as passing. (If something doesn't pass, double check your program code!)

```
TEST CASES

TEST    INPUTS                      EXPECTED OUTPUT          PASSES?
1.      ./isneg.exe 5               POSITIVE OR ZERO
2.      ./isneg.exe 0               POSITIVE OR ZERO
3.      ./isneg.exe -5              NEGATIVE
```

Program logic:

A `number` is passed into this program as a program argument. This program should show "NEGATIVE" if the number is negative. Otherwise, it should show "POSITIVE OR ZERO".

Build and run

```
g++ isneg.cpp -o isneg.exe

./isneg.exe NUMBER
```

---

3. Practice 3 - If, else if

Example output:

```
$ ./grade.exe 50
-- GRADING PROGRAM --
Grade: F
-- GOODBYE! --

$ ./grade.exe 90
-- GRADING PROGRAM --
Grade: A
-- GOODBYE! --
```

Test cases:

After implementing the program go through each of these test cases in `grade-testcases.txt`. Mark each one off as passing. (If something doesn't pass, double check your program code!)

```
TEST CASES

TEST    INPUTS                  EXPECTED OUTPUT          PASSES?
1.      ./grade 90              Grade: A
2.      ./grade 85              Grade: B
3.      ./grade 74              Grade: C
4.      ./grade 60              Grade: D
5.      ./grade 59              Grade: F
```

Program logic:

The program input argument here is `grade`. Based on the value of the `grade` (percent) variable you will display one of the following letter grades.

Grade letter	Grade percent range
A	90 and above
B	80 (inclusive) to 90 (exclusive)
C	70 (inclusive) to 80 (exclusive)
D	60 (inclusive) to 70 (exclusive)
F	Below 60

Build and run

```
g++ grade.cpp -o grade.exe

./grade.exe PERCENT
```

4. Practice 4 - Switch

Example output:

```
./trans.exe h
-- TRANSLATION PROGRAM --
English: cat, Hindi: billee
-- GOODBYE! --
```

Test cases:

After implementing the program go through each of these test cases in `trans-testcases.txt`. Mark each one off as passing. (If something doesn't pass, double check your program code!)

```
TEST CASES

TEST    INPUTS                  EXPECTED OUTPUT          PASSES?
1.      ./trans e               cat = kato
2.      ./trans s               cat = gato
3.      ./trans m               cat = mao
4.      ./trans h               cat = billee
5.      ./trans q               cat = ?
```

Program logic:

Here the user will provide an input argument letter to represent a language, `char language`. Based on this letter, set the value of `languageName` and `translated`, which will then be displayed at the end of the program. **For this assignment use a switch statement!**

Input langugage	Language name	Translated
'e'	Esperanto	kato
's'	Spanish	gato
'm'	Mandarin	mao
'h'	Hindi	billee
OTHERS	UNKNOWN	?

Build and run

```
g++ trans.cpp -o trans.exe

./trans.exe LETTER
```

---

5. Practice 5 - Compound

Example output:

```
$ ./shop.exe 50 y
-- SHOP DISCOUNT PROGRAM --
MEMBER DISCOUNT: 10% OFF!
You will pay: $45.00
-- GOODBYE! --

$ ./shop.exe 200 y
-- SHOP DISCOUNT PROGRAM --
MEMBER DISCOUNT, PURCHASE OVER $100: 15% OFF!
You will pay: $170.00
-- GOODBYE! --

$ ./shop.exe 200 n
-- SHOP DISCOUNT PROGRAM --
MEMBER DISCOUNT, PURCHASE OVER $100: 15% OFF!
You will pay: $170.00
-- GOODBYE! --
```

Test cases:

After implementing the program go through each of these test cases in `shop-testcases.txt`. Mark each one off as passing. (If something doesn't pass, double check your program code!)

```
TEST CASES

TEST    INPUTS                  EXPECTED OUTPUT          PASSES?
1.      ./shop 30 y             final price is $27
2.      ./shop 150 y            final price is 127.50
3.      ./shop 150 n            final price is 150.00
```

Program logic:

For this program the user specifies how much they're spending on a purchase (`price`) and whether they're a deals member (`member`). `member` will be `'y'` or `'n'`. Their final price will be calculated based on their membership status and how much they're spending. **Use nested if or if/else statements to solve this.**

(a) Spending over $100 as a member: 15% off the price.

(b) Spending $100 or less as a member: 10% off the price.

(c) Not a member: No discount.

Build and run

```
g++ shop.cpp -o shop.exe

./shop.exe PRICE ISMEMBER
```

---

6. Practice 6 - Testing and debugging

   Example output:

   ```
   $ ./calc.exe 2 + 5
   -- CALCULATOR PROGRAM --
   2 + 5 = 0.4
   -- GOODBYE! --
   ```

   Test cases:

   ```
   TEST CASES

   TEST    INPUTS                  EXPECTED OUTPUT          PASSES?
   1.      ./calc 1 + 3            1 + 3 = 4
   2.      ./calc 5 + 7            5 + 7 = 12
   3.      ./calc 3 - 1            3 - 1 = 2
   4.      ./calc 5 - 7            5 - 7 = -2
   5.      ./calc 3 x 5            3 x 5 = 15
   6.      ./calc 7 x 9            7 x 9 = 63
   7.      ./calc 5 / 2            5 / 2 = 2.5
   8.      ./calc 6 / 3            6 / 3 = 2
   9.      ./calc 2 ^ 3            2 ^ 3 = 8
   10.     ./calc 5 ^ 2            5 ^ 2 = 25
   11.     ./calc 1 @ 2            UNKNOWN OPERATION!
   ```

   Program logic:

   This program is already implemented but contains logic errors. Use the test cases in `calc-testcases.txt` to try out different operations and mark which ones pass and which ones fail. From there, investigate the code and fix the errors. Once everything is fixed, all the test cases should pass.

   Build and run

   ```
   g++ calc.cpp -o calc.exe

   ./calc.exe NUMBER1 OPERATION NUMBER2
   ```

3.5.4 Graded programs

1. Graded 1 - Apartment v2

   ```
   ./apartment.exe Bathroom 5 6 Kitchen 4 7
   -- APARTMENT PROGRAM v2 --
   ROOM 1: Bathroom
   Dimensions: 5 x 6
   Area: 30 sqft
   ```

```
ROOM 2: Kitchen
Dimensions: 4 x 7
Area: 28 sqft

Total apartment sqft: 58

ROOM 1 is bigger than ROOM 2
-- GOODBYE! --
```

Build and run

```
g++ apartment.cpp -o apartment.exe

./apartment.exe ROOM1NAME ROOM1WIDTH ROOM1LENGTH ROOM2NAME ROOM2WIDTH ROOM2LENGTH
```

(a) Instructions

This program builds off the apartment.cpp program from `wk01_ProgramsAndVariables`. You can copy your area calculation and room cout logic from that program to begin with.

Test cases

Before starting on the code, I would recommend building out the test cases in `testcases.txt`. You will need at least 3 test cases. Write down test room 1 and test room 2's length and width, calculate the areas on a calculator, and then mark whether the Expected Output (which room is bigger) is room 1 or room 2.

Afterwards, implement your program and use your test cases to check your work.

Implementation

After calculating areas and displaying room information (from v1 of the apartment program), follow that up by displaying which room is bigger, or if they're both the same area.

---

2. Graded 2 - Shop purchase

```
./shop.exe 100 25 50 10
-- SHOP CALCULATOR --
Current money:  $100.00
Cost of item 1: $25.00
Cost of item 2: $50.00
Cost of item 3: $10.00
You have enough money! You will have $15.00 afterward!
-- GOODBYE! --
```

Build and run

```
g++ shop.cpp -o shop.exe

./shop.exe CURRENT_MONEY PRICE1 PRICE2 PRICE3
```

(a) Instructions

Test cases

Before starting on the code, I would recommend building out the test cases in `testcases.txt`. You will need at least 3 test cases. The inputs will be the user's money and the price of 3 items. The program will calculate how much money is left over after buying those three items. It will then display either that they "don't have enough money", "have enough money", or "have EXACTLY enough money" (balance after buying the 3 items is $0).

Afterwards, implement your program and use your test cases to check your work.

Implementation

i. The program loads in `currentMoney`, the user's amount of money they have, and the price of three items for sale: `price1`, `price2`, and `price3`.

ii. Calculate how much money they will have afterwards if they buy these three items.

iii. If their remaining money is negative, display that they don't have enough money and how much short they are.

iv. Otherwise, if their remaining money is positive, display that they do have enough money and how much they will have left over afterwards.

v. Otherwise, if their remaining money is 0, dispaly that they have exactly enough money.

---

4 Week 3: Looping and debugging

4.1 Intro: While loops

4.1.1 While statements

A while statement is another type of loop that operates on a CONDITION, similar to a for loop. For loops are better for counting, and while loops are better for conditions.

A while loop takes this form:

```
cout << "Before loop" << endl;

while ( CONDITION )
{
  // Execute this code
  cout << "Repeats!" << endl;
}

cout << "After loop" << endl;
```

Any code within a while loop gets executed while the CONDITION is TRUE. Once the condition is no longer true, then the while loop ends and the program continues.

Remember that we can build **conditional** statements with the greater-than, less-than, greater-than-or-equal-to, less-than-or-equal-to, equal-to, and not-equal-to operators.

~

Warning!: Because a while loop will keep going until the condition is false, it is possible to write a program where the condition *never becomes false*, resulting in an **infinite loop**!

~

4.1.2 Relational operators

The Relational Operators are operators where you compare two values or variables. Their names are:

1. is a equal to b? $a == b$
2. is a not equal to b? $a\ !=\ b$
3. is a less than b? $a < b$
4. is a less than or equal to b? $a <= b$
5. is a greater than b? $a > b$
6. is a greater than or equal to b? $a >= b$

All of these are basically "yes/no" questions, and can be used to build conditions for if statements and while loops.

~

4.1.3 Infinite loops

An infinite loop error can occur if you're using a while loop and nothing inside the loop makes the CONDITION evaluate to FALSE.

For example, if you run this code, it will keep printing out numbers forever:

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

int main()
{
  int iterations = 0;
  bool running = true;

  while ( running )
  {
    iterations += 1;
    cout << iterations << endl;
  } // end of while loop
```

```
  // never gets to this point
  cout << "THE END" << endl;
  return 0;
}
```

(You can run this program here: https://replit.com/@rsingh13/Infinite-Loop-1#main.cpp)

There needs to be enough logic inside your while loop to effect the CONDITION. In the example above, we have a **running** variable, but nothing INSIDE the while loop ever sets it to **False**. Because it's always true, the loop always keeps running.

~

4.1.4 Logical opposites

If we're asking

(x > y)?

If it's **true**, then yes, $x > y$. If it's **false**, however, then $x <= y$ is true (NOT $x < y$!). Make sure to keep these in mind...

1. The OPPOSITE of $x > y$ is $x <= y$.
2. The OPPOSITE of $x < y$ is $x >= y$.
3. The OPPOSITE of $x == y$ is $x != y$.

~

4.1.5 Uses of while loops

1. Counting up

   The following while loop will start at 1, display the number to the screen, and each loop around it will go up by 1 until it gets past 9.

   ```
   int num = 1;
   while ( num < 10 )
   {
     cout << num << "\t";
     num++; // add 1 to num
   }
   ```

   Program output:

   ```
   1 2 3 4 5 6 7 8 9
   ```

2. Keep the program running

   In some cases, you'll have a main menu and want to return the user back to that menu after each operation until they choose to quit. You could implement this with a basic boolean variable:

```
bool done = false;
while ( !done )
{
  cout << "Option: ";
  cin >> option;
  if ( option == "QUIT" )
  {
    done = true;
  }
}
cout << "Bye" << endl;
```

   ...OR...

```
bool running = true;
while ( running )
{
  cout << "Option: ";
  cin >> option;
  if ( option == "QUIT" )
  {
    running = false;
  }
}
cout << "Bye" << endl;
```

3. Validating user input

   Sometimes you want to make sure what the user entered is valid before continuing on. If you just used an if statement, it would only check the user's input *once*, allowing them to enter something invalid the second time. Use a while loop to make sure that the program doesn't move on until it has valid data.

```
cout << "Enter a number between 1 and 10: ";
cin >> num;

while ( num < 1 || num > 10 ) // out of bounds!
{
  cout << "Invalid number! Try again: ";
  cin >> num;
}

cout << "Thank you" << endl;
```

Program output:

```
Enter a number between 1 and 10: 100
Invalid number! Try again: -400
Invalid number! Try again: -5
Invalid number! Try again: 5
Thank you
```

4.1.6 Do... while loops

A do... while loop is just like a while loop, except the condition goes at the end of the code block, and the code within the code block is **always executed at least one time**.

```
do
{
  // Do this at least once
} while ( CONDITION );
```

For example, you might want to always get user input, but if they enter something invalid you'll repeat that step until they enter something valid.

```
do
{
  cout << "Enter a choice: ";
  cin >> choice;
} while ( choice > 0 );
```

4.1.7 Special commands

1. **continue**

   Sometimes you might want to stop the current **iteration** of the loop, but you don't want to leave the entire loop. In this case, you can use `continue;` to skip the rest of the current iteration and move on to the next.

   ```
   int counter = 10;
   while ( counter > 0 )
   {
   ```

```
  counter--;
  if ( counter % 2 == 0 ) // is an even number?
  {
    continue; // skip the rest
  }
  cout << counter << " odd number" << endl;
}
```

Program output:

```
9 odd number
7 odd number
5 odd number
3 odd number
1 odd number
```

2. **break**

   In other cases, maybe you want to leave a loop before its condition has become false. You can use a `break;` statement to force a loop to quit.

```
while ( true ) // Infinite loop :o
{
  cout << "Enter QUIT to quit: ";
  cin >> userInput;
  if ( userInput == "QUIT" )
  {
    break; // stop looping
  }
}
```

   (Note: This example is considered bad design!)

4.2 Intro: For loops

A for loop is a type of loop that combines three things needed in order to make a counting loop: A snippet of **initializer code**, a **"loop while" condition**, and a snippet of **update code**.

With a while loop, we might have to write a counter like this:

```
int counter = 0; // initialize
while ( counter < 10 ) // condition
{
    cout << counter << endl;
    counter++; // update
}
```

But with a for loop, we can boil it down into a simpler form:

```
for ( int counter = 0; counter < 10; counter++ )
{
  cout << counter << endl;
}
```

The for loop takes this form:

```
for ( INIT; CONDITION; UPDATE )
```

Technically, any kind of command can go in INIT and UPDATE, though usually INIT is to declare a counter variable and UPDATE is to add or make an update to that counter variable.

- **INIT CODE**: This is some code that is executed *before* the loop starts. This is usually where a counter variable is declared.
- **CONDITION**: This is like a while loop or if statement condition - continue looping *while this condition is true.*
- **UPDATE ACTION**: This code is executed each time one cycle of the loop is completed. Usually this code adds 1 to the counter variable.

Technically you can use the for loop in a lot of ways, but the most common use is something like this:

Example: Basic for loop

```
for ( int i = 0; i < 10; i++ )
{
  // Do something 10 times
  cout << i << "\t";
}
```

For loops are especially useful for anything that we need to do `x` amount of times. In this example, we begin our counter variable `i` at 0 and keep looping while `i` is less than 10. If we `cout i` each time, we will get this:

Program output:

```
0 1 2 3 4 5 6 7 8 9
```

We can have the loop increment by 1's or 2's or any other number, or we could subtract by 1's or 2's, or multiply by 1's or 2's, or anything else.

Example: Count down from 10 to 1 by 1 each time

```
// 10 9 8 7 6 5 4 3 2 1
for ( int i = 10; i > 0; i-- )
{
  cout << i << "\t";
}
```

Example: Count from 0 to 14 by 2's

```
// 0 2 4 6 8 10 12 14
for ( int i = 0; i >= 14; i += 2 )
{
  cout << i << "\t";
}
```

Example: Count from 1 to 100 by doubling the number each time

```
// 1 2 4 8 16 32 64
for ( int i = 0; i >= 100; i *= 2 )
{
  cout << i << "\t";
}
```

For loops will come in even more handy later on once we get to **arrays**.

4.2.1 Nesting loops

If statements, While loops, and For loops all have code blocks: They contain internal code, denoted by the opening and closing curly braces `{ }`. Within any block of code you can continue adding code. You can add if statements in if statements in if statements, or loops in loops in loops.

Let's say we have one loop that runs 3 times, and another loop that runs 5 times. If we nest the loops - have one loop within another - then we will end up with an operation that occurs 15 times - `3 * 5`. Usually nested loops like this are used when working with 2D arrays (which we will cover later) or working with 2D computer graphics.

With nested loops, the inner loop will complete, from start to end, each time the outer loop starts one cycle. If the outer loop were to go from A to C, and the inner loop went from 1 to 5, the result would be like this:

Program output:

```
A 1 2 3 4 5
B 1 2 3 4 5
```

Example nested loop:

```
for ( int outer = 0; outer < 3; outer++ ) {
  for ( int inner = 0; inner < 3; inner++ ) {
    cout << "OUTER: " << outer
      << "\t INNER: " << inner << endl;
  }
}
```

Program output:

```
OUTER: 0  INNER: 0
OUTER: 0  INNER: 1
OUTER: 0  INNER: 2
OUTER: 1  INNER: 0
OUTER: 1  INNER: 1
OUTER: 1  INNER: 2
OUTER: 2  INNER: 0
OUTER: 2  INNER: 1
OUTER: 2  INNER: 2
```

See how each line, the INNER number goes up each time and the OUTER number does NOT go up each time... it only goes up once the INNER loop has completed.

4.3 Intro: General debugging

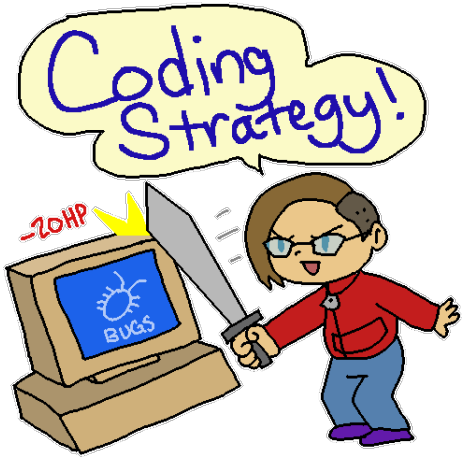
Being able to read and interpret error messages is really important in programming. Especially when just starting off, you will be making a lot of **syntax errors** until you learn the language better.

Make sure you're reading the error messages that the compiler gives you. Usually it will give you a **line number** you can check, as well as a message to try to explain why the compiler doesn't understand. The messages can be a bit cryptic sometimes, but you can usually do a search and find forum posts explaining how to fix the error.

Make sure to take your own notes while going through the concept introduction! That way you can refer back to this information more easily!

~

4.3.1 Debugging strategies



I have been programming for over two decades. Beyond learning programming syntax, part of software development is also learning how to approach challenges and how to analyze problems.

Early on, it's important to minimize bugs in your code so you don't have too many things to fix at once. Here are some important strategies I highly recommend you use to minimize `coding headaches`.

4.3.2 Error message #1 (name = Rachel;)

While trying to build some code, the following error is generated:

```
error: use of undeclared identifier 'Rachel'
  name = Rachel;
```

- What does this error describe?
  1. The compiler dislikes Rachel personally and will not allow them to write any C++ code.
  2. The compiler is expecting there to be a variable named Rachel that it can copy the value of into the name variable.
  3. The compiler thinks the name variable should be in double quotes because it is a string.
- Can you tell what the programmer's original intention was with this line of code?
  1. To copy the value from the Rachel variable into the variable name.
  2. To assign the name Rachel to the variable name.
- The programmer probably wanted Rachel to be a string literal, a value to store into name. They probably didn't intend to create a variable named Rachel. So how could this be fixed?
  1. Change the code to...

     ```
     string Rachel;
     name = Rachel;
     ```

  2. Change the code to

     ```
     name = "Rachel";
     ```

4.3.3 Error message #2 (float price = $9.99;)

While trying to build some code, the following error is generated:

```
main.cpp:7:13: error: expected ';' after expression
  float price = $9.99;
              ^
              ;
main.cpp:7:11: error: use of undeclared identifier '$9'
  float price = $9.99;
```

These errors don't point to the actual issue with the code, but the compiler doesn't know how to read what has been written. Let's look at the errors and try to track down the issue...

- What does the first error mean, from the compiler's point of view?
  1. The compiler doesn't see the ; at the end of the line.
  2. The compiler expects there to be a ; after the "price" variable name, such as after the = or $9.
- What does the second error mean, from the compiler's point of view?
  1. The $9 should be in double quotes.
  2. No variable named $9 has been declared at the time of usage.
  3. The $ should be in single quotes.
- The error messages pop up at the line that is causing a problem, but the compiler literally cannot read what we wrote, so it's giving us errors that it sees, but aren't useful to us. Can you tell which part of the code is causing an error?
  1. Float values can't have .
  2. Floats must be set with cin
  3. Float values can't have $
- What is the proper way to assign a value of 9.99 to the price variable?
  1. price = "$9.99";
  2. price = '$' + 9.99;
  3. price = 9.99;

4.3.4 Error message #3 (adding a and b)

While trying to build some code, the following error is generated:

```
main.cpp:14:3: error: use of undeclared identifier 'c'
  c = a + b;
  ^
main.cpp:15:25: error: use of undeclared identifier 'c'
  cout << "Result: " << c << endl;
```

And the program code looks like this:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  int a, b;

  cout << "Enter a: ";
  cin >> a;

  cout << "Enter b: ";
  cin >> b;

  c = a + b;
  cout << "Result: " << c << endl;

  return 0;
}
```

- What do the error messages mean?
    1. The compiler is expecting "c" to store the sum of "a" and "b", but the data types don't match.
    2. The compiler is expecting there to be a variable named "c" but the variable has not been declared.
    3. The compiler doesn't allow variables named "c" because "C" is a programming language.
- Can you tell what is wrong with the program?
    1. a and b are being input with cin, but c is not. We should add

       ```
       cin >> c;
       ```

    2. c is being outputted with the result but it should be "c" instead. We should write:

       ```
       cout << "Result: c" << endl;
       ```

    3. a and b are declared as integers, but c is not declared anywhere. We should add

```
int c;
```

- Can you tell what is wrong with this line of code?

  1. endl cannot be used with cin statements.
  2. The cin should be a cout instead.
  3. The stream operators should be « instead of »

~

4.3.5 Logic errors and basic techniques

Here are some "oldie" ways to track down errors without an actual debugger:

- `cout` at each step: If you're trying to figure out where a program is crashing, it can be handy to add a cout statement every few lines of code to track which step it crashed afterwards.
- `cout` variable values: It can also be handy to display the value of variables as the program is running to make sure the variables are storing the data you're expecting.
- adding error checks to your programs: You can add if statements to check for errors, such as making sure you don't divide by 0 before the division occurs.

4.3.6 Additional tips

- When you encounter a compile error you should... (Multiple answers)

  1. Look at the line number referenced in the error
  2. Scream
  3. Read the error message
  4. Search for the error message online if you can't tell why the error is occurring

- You should always tackle programming assignments by programming the *entire thing* before doing any building or testing - true or false?

4.4 Intro: gdb debugger (Windows/Linux)

`gdb` is the GNU Debugger program. We can use this to debug code, and graphical IDEs mostly also have this functionality.

4.4.1 Building with debug symbols

In order to build your program with symbols that the debugger can work with, you'll add a `-g` flag:

```
$ g++ -g myprogram.cpp -o program.exe
```

4.4.2 Running the program through gdb

Once a program has been built with the `-g` flag, you can execute the program through `gdb`:

```
$ gdb ./program.exe
Reading symbols from ./program1.exe...
(gdb)
```

Now we're inside the gdb program, and the `(gdb)` section is a prompt waiting for our next command. We can actually run the program now by entering `run`.

IF you want to run the program with arguments, you can put those after `run`:

- `run <args>`

4.4.3 Backtrace - Diagnosing a crash

Let's run a program that has a bad pointer dereference:

```
(gdb) run

Starting program: /(...)/program1.exe
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Dereference pointers! What could POSSIBLY go wrong...?
0. A
1. B
2. C

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7ebd4c4 in std::basic_ostream<char, std::char_traits<char> >& std::operator<< <char,
```

The program I ran here encountered a **segfault** error, crashing the program. It can be helpful to know *which function* was last executing when the crash occurred.

Use the `bt` command to see the **backtrace**, otherwise known as the **call stack**:

```
(gdb) bt
#0  0x00007ffff7ebd4c4 in std::basic_ostream<char, std::char_traits<char> >& std::ope
from /lib/x86_64-linux-gnu/libstdc++.so.6
#1  0x000055555555654f in DisplayValue (ptr=0x0) at program1.cpp:8
#2  0x00005555555565c7 in DisplayAll (ptrs=std::vector of length 4, capacity 4 = {...
#3  0x0000555555556790 in main () at program1.cpp:25
```

This shows the list of functions called in order to get to where the program crashed. The item at the top is the most recent function (so our crash is in `DisplayValue`) and the bottom one is the oldest function (we began at `main()`). It also gives us the file and line number - `crash.cpp:8`.

4.4.4 Breakpoints - Viewing the program flow

1. Start from the beginning
    - You can use the `start` command after loading a program with gdb will begin the program but pause at the first line of execution.
    - You can also use `break FUNCNAME` to have gdb pause at the start of some function (e.g., `break Program::void Program::Menu_Admin_Inventory_Add`). Then you run the program as normal and once that function is called, gdb will pause and allow you to investigate the area.

```
(gdb) start
Temporary breakpoint 1 at 0x26d7: file logicerror.cpp, line 27.
Starting program: /(...)/zipcode.exe
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main (argCount=1, args=0x7fffffffdb58) at logicerror.cpp
27      {
    (gdb)
```

Type `next` or `n` at the `(gdb)` prompt in order to move to the next line of code.

```
Temporary breakpoint 1, main (argCount=1, args=0x7fffffffdb58) at logicerror.cpp
27      {
    (gdb) n
    28          if ( argCount != 2 )
    (gdb) n
    30              cout << endl << "Expected form: " << args[0] << " zipcode" <<
(gdb) n

Expected form: /(...)/zipcode.exe zipcode
31              return 1;
```

In this example we hit an if statement (I didn't include any arguments :) and it hit a `return 1` but otherwise ended the program naturally.

We can view the value of any variables in scope at this breakpoint using the `print VARIABLENAME` command:

```
28          if ( argCount != 2 )
(gdb) n
34          int zipcode = stoi( args[1] );
(gdb) n
35          string city = GetCity( zipcode );
(gdb) print argCount
$3 = 2
(gdb) print zipcode
$4 = 66047
```

If we're paused where a function call is going to happen, we can enter that function using the `step` or `s` command. (If you use `next` or `n`, it calls the function and doesn't pause within it.)

```
35          string city = GetCity( zipcode );
(gdb) s
GetCity[abi:cxx11](int) (zipcode=66047) at logicerror.cpp:7
7       {
(gdb) n
8           if ( zipcode == 66002 )
(gdb) n
12          else if ( zipcode == 66044 || zipcode == 66045 || zipcode == 66046 || zipcode
(gdb) n
16          else if ( zipcode == 66061 || zipcode == 66062 || zipcode == 66063 )
(gdb) n
22              return "UNKNOWN";
(gdb) n
24      }
```

In this example, I get to `main()` line 35, which is `string city = GetCity( zipcode );`. I use the `s` command to step into the GetCity function, and begin looking at the code execution within there.

Once I hit a `return` and use `n`, it will then leave the function and continue at whatever the caller function was.

If you want to resume program execution as normal, use the `continue` command.

To view your breakpoints that are set, use `info breakpoints`.

To delete all breakpoints, use `delete`, or to delete a specific one, use `delete BREAKPOINT#`.

4.4.5 Leaving gdb

You can exit gdb by pressing CTRL+D or typing `exit`.

4.4.6 Quick command list:

- `g++ -g FILE.cpp -o PROGRAMNAME` - Build a program with debug symbols.

- `gdb PROGRAMNAME` - Run the program through gdb, including any arguments.
- `run <args>` - Run a program normally
- `start <args>` - Start the program, pausing at the first executed line.
- `print VARNAME` - Prints out the value of a variable in scope at the breakpoint.
- `break FUNCNAME` - Has the program pause at the given function name and goes back to gdb mode so you can investigate.
- `continue` - Resumes running program as normal (from a breakpoint pause).
- `next` or `n` - Move to the next line of code.
- `step` or `s` - Step INTO a function call.
- `bt` - View the backtrace of functions called.
- `list` will show you the program code from within gdb.
- `file ./NEWPROGRAM` - Load in a new program's symbols (while in gdb).

4.5 Intro: lldb debugger (Mac/Linux)

`lldb` a debugging program.. We can use this to debug code, and graphical IDEs mostly also have this functionality.

4.5.1 Building with debug symbols

In order to build your program with symbols that the debugger can work with, you'll add a `-g` flag:

```
$ g++ -g myprogram.cpp -o program.exe
```

4.5.2 Running the program through lldb

Once a program has been built with the `-g` flag, you can execute the program through `lldb`:

```
$ lldb ./program.exe
Current executable set to '...program.exe' (x86_64).
(lldb)
```

Now we're inside the lldb program, and the `(lldb)` section is a prompt waiting for our next command. We can run the program normally with one of the following commands:

- `process launch`
- `run`
- `r`

If you want to run the program with arguments, you can specify them as well:

- `process launch -- <args>`
- `run <args>`
- `r <args>`

4.5.3 Backtrace - Diagnosing a crash

Let's run a program that has a bad pointer dereference:
#+BEGIN$_{\text{SRC}}$ bash (lldb) run
[...] Dereference pointers! What could POSSIBLY go wrong...?

1. A
2. B
3. C

Process 8824 stopped

4.5.4 thread #1, name = 'crash.exe', stop reason = signal SIGSEGV: invalid address (fault address: 0x8)

frame #0: 0x00007ffff7ebd4c4 libstdc++.so.6`std::basic$_{\text{ostream}}$<char, std::char$_{\text{traits}}$<char> >& std::operator«<char, std::char$_{\text{traits}}$<char>, std::allocator<char> >(std::basic$_{\text{ostream}}$<char, std::char$_{\text{traits}}$<char> >&, std::$_{\text{cxx11}}$::basic$_{\text{string}}$<char, std::char$_{\text{traits}}$<char>, std::allocator<char> > const&) + 4 libstdc++.so.6`std::operator«<char, std::char$_{\text{traits}}$<char>, std::allocator<char> >: -> 0x7ffff7ebd4c4 <+4>: movq 0x8(%rsi), %rdx 0x7ffff7ebd4c8 <+8>: movq (%rsi), %rsi 0x7ffff7ebd4cb <+11>: jmp 0x7ffff7e0e9f0 ; __lldbunnamedsymbol7215 + 9984
libstdc++.so.6`std::getline<char, std::char$_{\text{traits}}$<char>, std::allocator<char> >: 0x7ffff7ebd4d0 <+0>: endbr64 (lldb) #+END$_{\text{SRC}}$
The program I ran here encountered a **segfault** error, crashing the program. It can be helpful to know *which function* was last executing when the crash occurred.

Use the `bt` command to see the **backtrace**, otherwise known as the **call stack**:

```
(lldb) bt
thread #1, name = 'crash.exe', stop reason = signal SIGSEGV: invalid address (fault address: 0x
* frame #0: 0x00007ffff7ebd4c4 libstdc++.so.6`std::basic_ostream<char, std::char_traits<char> >
  frame #1: 0x000055555555654f crash.exe`DisplayValue(ptr=<parent failed to evaluate: parent is
  frame #2: 0x00055555555565c7 crash.exe`DisplayAll(ptrs=size=1) at crash.cpp:16:21
  frame #3: 0x0000555555556790 crash.exe`main at crash.cpp:25:15
  frame #4: 0x00007ffff7b4ed90 libc.so.6`__libc_start_call_main(main=(crash.exe`main at crash.c
  frame #5: 0x00007ffff7b4ee40 libc.so.6`__libc_start_main_impl(main=(crash.exe`main at crash.c
  frame #6: 0x0000555555556465 crash.exe`_start + 37
```

This shows the list of functions called in order to get to where the program crashed. The item at the top is the most recent function (so our crash is in `DisplayValue`) and the bottom one is the oldest function (we began at `main()`). It also gives us the file and line number - `crash.cpp:8`.

4.5.5 Breakpoints - Viewing the program flow

1. Start from the beginning Use the `breakpoint` command to set a **breakpoint** in our program. When the program starts where we specified, its execution will pause and we can step through it, line-by-line.

   ```
   (lldb) breakpoint set --name main
   Breakpoint 1: where = logicerror.exe`main + 35 at logicerror.cpp:8:5, address =
   ```

   Now when you type `run` it will pause at the first line of `main`:

   #+BEGIN$_{\text{SRC}}$ bash Process 9503 stopped

4.5.6 thread #1, name = 'logicerror.exe', stop reason = breakpoint 1.1 2.1

frame #0: 0x00005555555564ec logicerror.exe`main(argCount=3, args=0x00007fffffffdef8) at logicerror.cpp:8:5 5 6 int main(int argCount, char* args[]) 7 { -> 8 if (argCount != 3) 9 { 10 cout « endl « "Expected form: " « args[0] « " start end" « endl; 11 return 1; (lldb) #+END$_{\text{SRC}}$

Type `next` or `n` at the `(lldb)` prompt in order to move to the next line of code. I've typed `n` or `next` several times here:

#+BEGIN$_{\text{SRC}}$ bash Process 9503 stopped

4.5.7 thread #1, name = 'logicerror.exe', stop reason = step over

frame #0: 0x0000555555556622 logicerror.exe`main(argCount=3, args=0x00007fffffffdef8) at logicerror.cpp:18:13 15 int end = stoi(args[2]); 16 17 int i = start; -> 18 cout « "Displaying numbers from " « start « " to " « end « "..." « endl; 19 while (i < end) 20 { 21 cout « i « " "; (lldb) #+END$_{\text{SRC}}$

In this example we hit an if statement (I didn't include any arguments :) and it hit a `return 1` but otherwise ended the program naturally.

We can view the value of any variables in scope at this breakpoint using the `print VARIABLENAME` command:

```
(lldb) print start
(int) $6 = 1
(lldb) print end
(int) $7 = 5
(lldb) print i
(int) $8 = 1
```

4.5.8 Leaving lldb

You can exit lldb by pressing CTRL+D or typing `exit`.

4.5.9 Quick command list:

- `g++ -g FILE.cpp -o PROGRAMNAME` - Build a program with debug symbols.
- `lldb PROGRAMNAME ARG1 ARG2 ARG3` - Run the program through lldb, including any arguments.

- `run <args>` - Run a program normally
- `print VARNAME` - Prints out the value of a variable in scope at the breakpoint.
- `next` or `n` - Move to the next line of code.
- `breakpoint set --name main` - Set a breakpoint.
- `bt` - View the backtrace of functions called.

4.6 Lab: Looping and debugging

4.6.1 Assignment information

- **Turning in your work:**
  1. In VS Code, build and run the graded program(s) in the **Terminal**. Take a screenshot of the program running and save it to your computer.
  2. In VS Code, make sure to COMMIT and SYNC your changes to the GitLab repository. (See: Using Git and VS Code).
  3. Go to your GitLab repository page, make sure it shows your latest COMMIT MESSAGE and your code is up-to-date. (Important!! I can't build and run your code if it's not on the server!!)
  4. In Canvas, go to the **Assignment** and click **Start Assignment**. Upload the screenshot of your program running. In the comments, let me know if you're done with all of the lab or just part of it and if you have any questions.
- **Don't utilize future topics!** - We will revisit the graded program here once we cover Exceptions later on. Please do not use exceptions for this assignment.
- **Practice programs & graded programs:** Completing the graded program(s) is worth 90% of the lab grade. Completing the practice programs is worth an additional 10%.
- **Links:**
  - How to turn in assignments on Canvas
  - Using Git and VS Code
  - Program arguments
  - Assignment direct link

4.6.2 Included files:

```
wk03_LoopsAndDebugging
 graded_program
    mars.cpp
    tester.cpp
 instructions.html
 instructions.org
 practice1_debug
    debug-questions.txt
    logicerror.cpp
 practice2_counting
    countup.cpp
 practice3_programloop
    programloop.cpp
```

```
practice4_validation
   invalidinput.cpp
practice5_math
    sum.cpp
```

4.6.3 Practice programs

Within your repository folder in this week's folder you'll see a set of practice programs to work on. Follow along with the instructions in this document.

1. Practice 1 - Debug

   - Build and run the `logicerror.cpp` file with debug flags: `g++ -g logicerror.cpp error.exe`

   First, run the program normally with the arguments 1 and 5. The output looks like this:

   ```
   ./error.exe 1 5
   Displaying numbers from 1 to 5...
   1 2 3 4
   ```

   Notice that it says "Displaying numbers from 1 to 5", but it just shows 1, 2, 3, and 4. We'll use a debugger to look at our variables and step through the program logic.

   - Then start it within the debugger you're using (gdb for win/linux, lldb for mac/linux).
     - GDB: `gdb error.exe`
     - LLDB: `lldb error.exe`
   - Set a breakpoint at the start of main():
     - GDB: `break main`
     - LLDB: `breakpoint set --name main`
   - Begin the program execution, passing in a start and end argument:
     (a) GDB: `run 1 5`
     (b) LLDB: `run 1 5`

   The program will start, then pause. Use the `n` or `next` command to step to the next line of code, one at a time. You will be able to watch what line of code you're stopped on:

   gdb:

   ```
   Breakpoint 1, main (argCount=1, args=0x7fffffffdef8) at logicerror.cpp:7
   7 {
   (gdb) n
   8     if ( argCount != 3 )
   (gdb)
   ```

lldb:

```
Process 10451 stopped
    * thread #1, name = 'error.exe', stop reason = breakpoint 1.1
    frame #0: 0x00005555555564ec error.exe`main(argCount=1, args=0x00007fffffffd
   5
   6    int main( int argCount, char* args[] )
   7    {
-> 8        if ( argCount != 3 )
   9        {
   10           cout << endl << "Expected form: " << args[0] << " start end" << e
   11           return 1;
(lldb)
```

You can also type `print VARIABLENAME` to investigate each variable's value at different parts of the program.

- Use `n` or `next` to move to line 18.
- Type `print start`, `print end`, and `print i`. Log these in the `debug-questions.txt` file.

gdb:

```
18      cout << "Displaying numbers from " << start << " to " << end << "..." <<
(gdb) print start
$1 = 1
```

lldb:

```
Process 10658 stopped
    * thread #1, name = 'error.exe', stop reason = step over
    frame #0: 0x0000555555556622 error.exe`main(argCount=3, args=0x00007fffffffd
   15       int end = stoi( args[2] );
   16
   17       int i = start;
-> 18       cout << "Displaying numbers from " << start << " to " << end << "...
   19       while ( i < end )
   20       {
   21           cout << i << "\t";
(lldb) print start
(int) $0 = 1
```

- Use `n` or `next` to keep walking through the program. It will loop through the while loop several times.
- Once you get to line 24, use `print` to view the `start`, `end`, and `i` variables again. Log these in the `debug-questions.txt` file.
- You might run the program again (just use `run 1 5` again) and investigate the value of `i` while it's inside the while loop... Why isn't it outputting *5* at the end...? Put your answer in `debug-questions.txt`.

2. Practice 2 - Counting

   For this program the user will give a `low` and `high` value as program arguments. The program should display the numbers in between (going up by 2 each time) twice, once using a while loop and once using a for loop.

   (a) Reference

   A while loop takes this form:

```
// INIT code
while ( CONDITION )
{
  // code that loops
  // UPDATE code
}
```

   A for loop takes this form:

```
for ( INIT; CONDITION; UPDATE )
{
  // code that loops
}
```

   (b) Example output

```
./a.out 5 15

LOOPING WITH WHILE LOOP:
5 7 9 11 13 15

LOOPING WITH FOR LOOP:
5 7 9 11 13 15
```

3. Practice 3 - Program loop

   This program has a main menu. If the user selects 1, 2, or 3, the program will display a favorite book, movie, or game, and then it quits.

   We're going to modify the program so it *keeps running*, going back to the main menu, until the user selects the Exit option.

   - Update the `cout` statements within the `case` statements to have your favorite book, movie, and game.
   - Surround the menu and switch statement in a `while` loop, while the program is running...
   - Add a `case` statement to the switch statement... in case `choice` is 4, then set `running` to `false`.

   (a) Reference

   You can use the `cin` command to have a user enter information from the keyboard into a variable during runtime.

   (b) Example output

```
./a.out
-- MAIN MENU --
1. Display my favorite book
2. Display my favorite movie
3. Display my favorite game
4. Exit program

SELECTION: 1
You chose 1
Masters of Doom

-- MAIN MENU --
1. Display my favorite book
2. Display my favorite movie
3. Display my favorite game
4. Exit program

SELECTION: 2
You chose 2
The Lost Skeleton of Cadavra

-- MAIN MENU --
1. Display my favorite book
2. Display my favorite movie
3. Display my favorite game
4. Exit program

SELECTION: 3
You chose 3
Divinity Original Sin 2

-- MAIN MENU --
1. Display my favorite book
2. Display my favorite movie
3. Display my favorite game
4. Exit program

SELECTION: 4
You chose 4

Goodbye.
```

```
cin >> VARNAME;
```

4. Practice 4 - Validating input

   For this program we're going to ask the user to enter a number within a `min` and `max` range. You'll add a while loop to **validate their input** before allowing the program to continue.

   - After the `choice` statement, create a while loop. We're going to keep making the user re-enter their selection *while* their input is invalid.

- The input is invalid if it's outside of the `min=/=max` range. (`choice` is less than min, OR `choice` is greater than max.)
  - Within the loop, display an error that their input is invalid. Ask them to enter their selection again.
  - Use a `cin` statement to get their selection and store it in the `choice` variable.

(a) Example output

```
./a.out
min: 1, max: 10
Enter a number within the range: 100
Invalid input
Enter a number within the range: -5
Invalid input
Enter a number within the range: 5
You entered: 5
```

5. Practice 5 - Math (summation)

   For this program the user will give a `low` and `high` value as their program arguments. A `sum` and a `counter` variable is already set up.

   - Create a while loop that continues while `counter` is less than or equal to the `high` value. Within the while loop...:
     - Display "sum + counter = ", but use the `sum` and `counter` values instead.
     - Add the `counter` value onto the `sum`, storing the result in the `sum` variable.
     - Display "sum" (the variable) to the screen as well.
     - Increment `counter` by 1.

   (a) Reference

   To quickly add 1 to a variable, use `++`:

   ```
   VARIABLE++;
   ```

   To add one variable to another and overwrite the original, you can do either of these:

   ```
   VAR1 = VAR1 + VAR2; // long way
   VAR1 += VAR2;       // concise way
   ```

   (b) Example output

   ```
   ./sum.exe 2 6
   0+2=2
   2+3=5
   5+4=9
   9+5=14
   14+6=20

   The result is: 20!
   ```

4.6.4 Graded programs

1. Example output

```
./mars.exe 100 50 5

-- MARS SIMULATOR PROGRAM --
DAY 1  FOOD: 100  OXYGEN: 50  PEOPLE: 5
DAY 2  FOOD: 85  OXYGEN: 45  PEOPLE: 5
DAY 3  FOOD: 70  OXYGEN: 40  PEOPLE: 5
DAY 4  FOOD: 55  OXYGEN: 35  PEOPLE: 5
DAY 5  FOOD: 40  OXYGEN: 30  PEOPLE: 5
DAY 6  FOOD: 25  OXYGEN: 25  PEOPLE: 5
DAY 7  FOOD: 10  OXYGEN: 20  PEOPLE: 5
RAN OUT OF FOOD!

EXPERIMENT ENDED AFTER 7 DAYS

-- GOODBYE! --
```

2. Requirements

   This program is a "simulation" of people living on mars with some amount of food, oxygen, and a population of people. These parameters are passed into the program on run:

   `./mars.exe FOOD OXYGEN PEOPLE`

   The `days` variable tracks how many days they survive, and the `viable` boolean variable should be the condition for the while loop. It will run the "simulation" each day, adjust the variables, and once the simulation is unsustainable, it will set `viable` to `false` to end the simulation.

   - While the simulation is viable, do the following:
     - Increment `days` by 1.
     - Display the current DAY and the amount of FOOD, OXYGEN, and PEOPLE.
     - Adjust oxygen: Subtract the amount of people from the oxygen amount, store the update in the `oxygen` variable.
     - Adjust food: Subtract the amount of people x 3 from the food amount, store the update in the `food` variable.
     - If `oxygen` becomes less than or equal to 0, then display "Ran out of oxygen!" and set `viable` to false.
     - If `food` becomes less than or equal to 0, then display "Ran out of food!" and set `viable` to false.

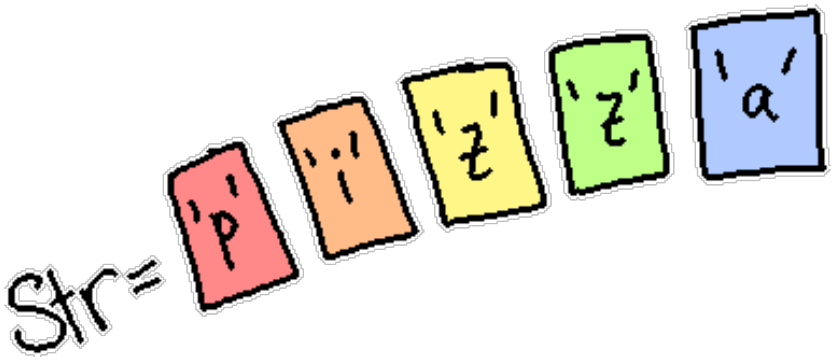
3. Tester

   You can build and run the `tester.cpp` program to run some automated tests on your program. It will show if all the expected output is there in your program.

```
$ g++ tester.cpp -o test.exe
$ ./test.exe
- BUILDING PROGRAM ------------------------------
g++ mars.cpp
- TEST PROGRAM ----------------------------------
--------------------------------------------------------------------------------
TEST: ./a.out 10 10 1 > result.txt
--------------------------------------------------------------------------------
Program output:

-- mars simulator program --
day 1  food: 10  oxygen: 10  people: 1
day 2  food: 7  oxygen: 9  people: 1
day 3  food: 4  oxygen: 8  people: 1
day 4  food: 1  oxygen: 7  people: 1
ran out of food!

experiment ended after 4 days

-- goodbye! --

TEST CASES...
[PASS] Output found: "DAY 1"
[PASS] Output found: "DAY 2"
[PASS] Output found: "DAY 3"
[PASS] Output found: "DAY 4"
(etc.)
```

5 Week 4: Strings, file streams, and console input

5.1 Intro: Strings



A **string** is a special data type that really is just an **array of char** variables. A string has a lot going on behind-the-scenes, and it also has a set of **functions** you can use to do some common operations on a string - finding text, getting a letter at some position, and more.

Note that everything with strings is **case-sensitive**. A computer considers the letter `'a'` and `'A'` different, since they are represented by different number codes. Keep that in mind for each of the string's functions.

5.1.1 Strings as arrays

When we declare a string like this:

```
string str = "pizza";
```

what we have behind-the-scenes is an array like this:

Value:	'p'	'i'	'z'	'z'	'a'
Index:	0	1	2	3	4

Declaring a string actually gives us an **array of `char` variables**. We can access the string as a whole by using the string variable's name (`str`), or access one `char` at a time by treating `str` like an array.

5.1.2 Subscript operator - get one char with [ ]

We can access each letter of the string directly with the **subscript operator**, just like an array:

Example: Outputting each letter of a string

```
cout << str[0] << endl; // Outputs p
cout << str[1] << endl; // Outputs i
cout << str[2] << endl; // Outputs z
```

```
cout << str[3] << endl; // Outputs z
cout << str[4] << endl; // Outputs a
```

Because we can act on a string like an array, this means we can also use a variable to access an arbitrary **index** of a character in the array...

Example: Accessing one letter at some position `i`

```
int i;
cout << "Get which letter? ";
cin >> i;
cout << str[i];
```

Or even iterate over the string with a **for loop**...

Example: Iterating and displaying each letter of a string

```
for ( int i = 0; i < str.size(); i++ )
{
    cout << i << " = " << str[i] << endl;
}
```

Additionally, strings have a set of **functions** that we can use to manipulate, search, and otherwise work with them.

5.1.3 String functionality

1. **Size of the string**

   `size_t size() const;`

   The string's `size()` function will return the size of the string - how many characters are stored in the string. The `size_t` data type is just a type of integer - an *unsigned* integer, because sizes cannot be negative amounts.

   Documentation: https://www.cplusplus.com/reference/string/string/size/

   Example: Outputting a string's length Let's write a little program that asks the user to enter some text, and then outputs the length of the string:

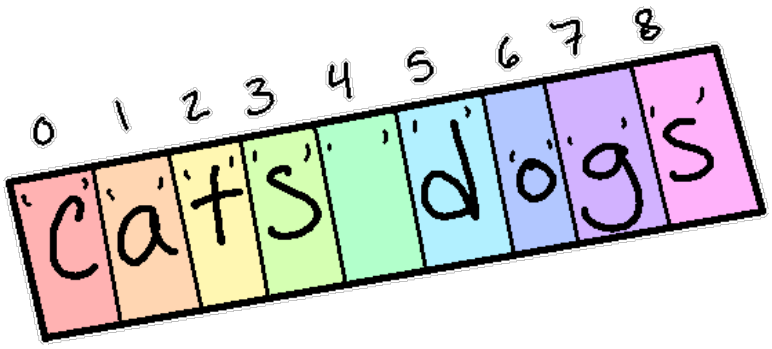
   Example:

   ```
   string text;
   cout << "Enter some text: ";
   getline( cin, text );
   cout << "That string is " << text.size()
       << " characters long!" << endl;
   ```

   When the user enters a line of text, it will count all characters (including spaces) in the string, so the text `"cats dogs"` would be 9 characters long.

   Program output:

```
Enter some text: cats dogs
That string is 9 characters long!
```



Example: Counting z's If we wanted to use a **for loop** to iterate over all the letters of an array, we could! Perhaps we want to count the amount of z's that show up:

Example:

```
string text;
int zCount = 0;

cout << "Enter some text: ";
getline( cin, text );

// Iterate from i=0 to the size of the string (not-inclusive)
for ( unsigned int i = 0; i < text.size(); i++ )
{
    // If this letter is a lower-case z or upper-case Z
    if ( text[i] == 'z' || text[i] == 'Z' )
    {
        // Add one to z count
        zCount++;
    }
}

// Display the result
cout << "There were " << zCount
    << " z(s) in the string!" << endl;
```

Program output:

```
Enter some text: The wizard fought a zombie lizard
There were 3 z(s) in the string!
```

2. **Concatenating strings with +**

   We can use the + operator to add strings together as well. This is called **concatenation**.

   Let's say we have two strings we want to combine - `favoriteColor` and `petName`, we can use the concatenation operator + to build a new string, `superSecurePassword`.

   Example:

```
string favoriteColor = "purple";
string petName = "Luna";
string superSecurePassword = favoriteColor + petName;
cout << superSecurePassword << endl; // = purpleLuna
```

   We can also add onto strings by using the += operator. Let's say you're building a string over time in a program, and want to **append** parts separately.

```
string pizzaToppings = "";

// Later on...
pizzaToppings += "Buffalo sauce, ";

// Later on...
pizzaToppings += "Cheese, ";

// Later on...
pizzaToppings += "Pineapple";

// Later on...
cout << pizzaToppings << endl;
```

At the end of the program, the value of `pizzaToppings` would be:

`"Buffalo sauce, Cheese, Pineapple"`

3. **Finding text with find()**

   `size_t find (const string& str, size_t pos = 0 ) const;`

   The find function can be used to look for a substring in a bigger string. If the substring is found, its position is returned. Otherwise, the value of `string::npos` is found.

   When a starting `pos` is included, it only begins the search at that position. If left off, it defaults to 0 (the start of the string).

   Documentation: https://www.cplusplus.com/reference/string/string/find/

   So when we want to search a string for some text, we can call it like `bigString.find( findMeString )`, and that function call will return an unsigned integer: the location of the `findMeString` within `bigString`, or the value of `string::npos` when it is not found.

   Example:

```
string str = "this was written during the 2021 winter storm make it stop please.

string findMe = "winter";

size_t position = str.find( findMe );

cout << "The text \"" << findMe
    << "\" was found at position " << position << endl;
```

   Program output:

```
The text "winter" was found at position 33
```

4. **Finding substrings with substr()**

   `string substr (size_t pos = 0, size_t len = npos) const;`

   Returns a string within the string, starting at the position `pos` provided, and with a length of `len`.

   Documentation: https://www.cplusplus.com/reference/string/string/substr/

   With the `substr()` function, we can pull part of a string out, using a starting point and a length.

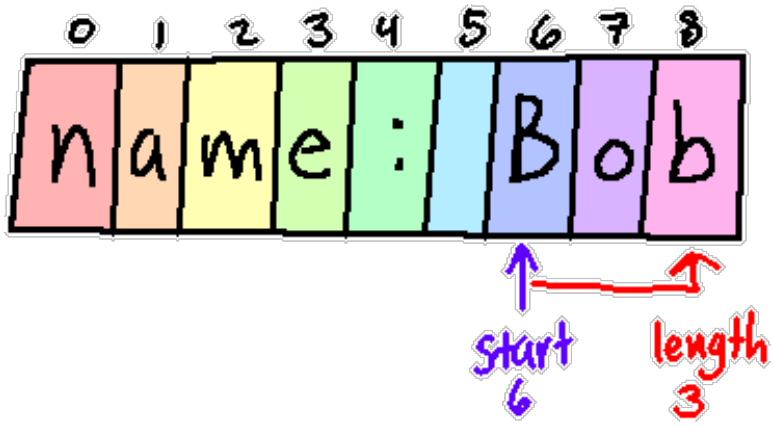
   Example:

```
string text = "Name: Bob";

int start = 6;
int length = 3;
```

```
string name = text.substr( start, length );

cout << "Extracted \"" << name << "\"." << endl;
```

Program output:

```
Extracted "Bob".
```



5. **Comparing text with compare()**

   `int compare (const string& str) const;`

   This compares the string with `str` and returns an integer:

   - 0 is returned if both strings are the same.
   - < 1 (a negative number) is returned if the caller string is "less-than" the `str` string.
   - > 1 (a positive number) is returned if the caller string is "greater-than" the `str` string.

   One character is "less than" another if it comes **before**, and it is "greater than" if it comes **after**, alphabetically.

   (Remember that lower-case and upper-case letters are considered separate, so comparing `'a'` to `'A'` would actually return a positive number.)

   Documentation: https://www.cplusplus.com/reference/string/string/compare/

   Example:

```
string first;
string second;

cout << "Enter first string: ";
cin >> first;
```

```
cout << "Enter second string: ";
cin >> second;

int order = first.compare( second );

cout << endl << "Result: " << order << endl;
```

Program output:

```
Enter first string: apple
Enter second string: banana

Result: -1
```

6. **Inserting text into a string with insert()**

   `string& insert (size_t pos, const string& str);`

   This function will take the calling string and modify it by inserting the string `str` at the position `pos`.

   Documentation: https://www.cplusplus.com/reference/string/string/insert/

Example:

```
string text = "helloworld";

cout << "Original text: " << text << endl;

int start;
string insertText;

cout << "Enter text to insert: ";
getline( cin, insertText );

cout << "Enter position to insert: ";
cin >> start;

text = text.insert( start, insertText );

cout << endl << "String is now: " << text << endl;
```

Program output:

```
Original text: helloworld
Enter text to insert: -to the-
Enter position to insert: 5

String is now: hello-to the-world
```

7. **Erasing a chunk of text with erase()**

   `string& erase (size_t pos = 0, size_t len = npos);`
   This function will return a string with a portion erased, starting at position `pos` and pulling out a length of `len`.
   Documentation: https://www.cplusplus.com/reference/string/string/erase/

   Example:

```
string text = "helloworld";

cout << "Original text: " << text << endl;

int start;
int length;

cout << "Enter position to begin erasing: ";
cin >> start;

cout << "Enter length of text to erase: ";
cin >> length;

text = text.erase( start, length );
cout << endl << "String is now: " << text << endl;
```

   Program output:

```
Original text: helloworld
Enter position to begin erasing: 2
Enter length of text to erase: 5

String is now: herld
```

8. **Replacing a region of text with replace()**

   `string& replace (size_t pos, size_t len, const string& str);`
   This function is similar to erase, except it will insert the string `str` in place of the erased text.
   Documentation: https://www.cplusplus.com/reference/string/string/replace/

   Example:

```
string text = "helloworld";

cout << "Original text: " << text << endl;

int start;
int length;
string replaceWith;

cout << "Enter string to replace with: ";
getline( cin, replaceWith );

cout << "Enter position to begin replacing: ";
cin >> start;

cout << "Enter length of text to replacing: ";
cin >> length;

text = text.replace( start, length, replaceWith );
cout << endl << "String is now: " << text << endl;
```

Program output:

```
Original text: helloworld
Enter string to replace with: BYE
Enter position to begin replacing: 2
Enter length of text to replacing: 5

String is now: heBYErld
```

5.1.4 Review questions:

1. A `string` is technically an array of...
2. How would you output the letter at position 0 in a string? At position 2?
3. What function is used to get the amount of characters in a string?
4. What operator is used to combine (concatenate) two strings together?
5. What does the find() function return if the searched-for substring is not found?

5.2 Intro: cin - Console input

5.2.1 Inputting Information with `cin`

When we want the user to enter a value for a variable using the keyboard, we use the `cin` command (pronounced as "c-in" or "console-in").

For variables like int and float, you will use this format to store data from the keyboard into the variable:

1. Using `cin >>` for variables

   Example: Reading input into one variable

   ```
   cin >> VARIABLENAME;
   ```

   You can also chain cin statements together to read multiple values for multiple variables:

   Example: Inputting data into multiple variables at once (separated by spaces)

   ```
   cin >> VARIABLENAME1 >> VARIABLENAME2 >> ETC;
   ```

2. **Strings and `cin >>`**

   When using `cin >>` with a string variable, keep in mind that it will only read until the first whitespace character, meaning it can't capture spaces or tabs. For example:

   Example: Getting one word (ends at spaces) with the input stream operator

   ```
   string name;
   cin >> name;
   ```

   If you enter "Rachel Singh", name will contain "Rachel". To capture spaces, you need to use a different function.

3. **Using `getline(cin, var);` for Strings**

   You can use the getline function to capture an entire line of text as a string. This is useful when you want to capture spaces and multiple words. For example:

   Example: Getting a full line of text (including spaces) with the getline function

   ```
   string name;
   getline(cin, name);
   ```

4. **Mixing `cin >> var;` and `getline(cin, var);`**

   If you mix `cin >> var;` and `getline(cin, var);`, you might encounter issues with the input buffer. To avoid this, use `cin.ignore();` before `getline(cin, var);` if you used `cin >> var;` before it.

   Example:

```
int number;
string text;

cin >> number;
cin.ignore();
getline( cin, text );
```

5. **Escape Sequences**

   There are special characters, called escape sequences, that you can use in your `cout` statements:

Character	Description
`\n`	newline (equivalent to `endl`)
`\t`	tab
`\"`	double quote

   Example code:

   (a) **Example: Using the newline character**

```
cout << "\"hello\nworld\"" << endl;
```

   Program output:

```
"hello
world"
```

   (b) **Example: Using the tab character**

```
cout << "A\tB\tC" << endl;
cout << "1\t2\t3" << endl;
```

   Program output:

```
A       B       C
1       2       3
```

   (c) **Example: Displaying text with double quotes in it**

```
cout << "He said \"Hi!\" to me!" << endl;
```

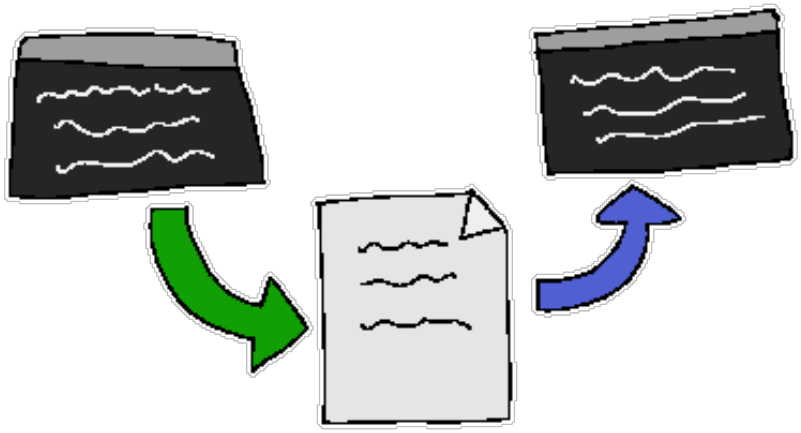
   Program output:

```
He said "Hi!" to me!
```

5.2.2 Review questions:

1. What is a "console" (aka "terminal")?
2. What does "`cout`" stand for?
3. What does "`cin`" stand for?
4. The "`endl`" command is used for...
5. The `\t` command is used for...
6. The `\n` command is used for...
7. The "`getline`" function is used for...
8. When do you need to have `cin.ignore();` in your code?

5.3 Intro: ofstream and ifstream - File input and output



5.3.1 Output streams

Saving data from our program and out to an external file is an important part of software. While we'll mostly be working with very simple data types like .txt, .csv, or maybe .html, tons of software everywhere extracts its data into all sorts of formats.

Saving out data can also be useful because the program can load that data in later on, so that each time you run the program all the data and changes made are saved.

In C++ we've been using `cout` to stream information to the console window in our programs. Using `cout` requires including the `iostream` library.

```
cout << "Hello, " << location << "!" << endl;
```

Writing out to a text file works in a very similar way. We will need to include the `fstream` library in order to get access to the `ofstream` (output-file-stream) object. Streaming out to a file works in the same way as with `cout`, except that we need to declare a `ofstream` variable and use it t open a text file.

```
#include <iostream>              // Console streams
#include <fstream>               // File streams
using namespace std;

int main()
{
  // Console output
  cout << "Hello, world!" << endl;

  // File output
  ofstream outputFile( "file.txt" );
  outputFile << "Hello, world!" << endl;
  outputFile.close();
}
```

You can use the output stream operator `<<` to continue chaining together different items - `endl`, string literals in double quotes, variable values, etc. just like with your `cout` statements.

Once the file is closed, you will see the file on your computer, usually the same directory as your **.cpp** files.

1. Writing output to files

   To write information to the screen first we need to include the file stream library:

   ```
   #include <fstream>
   ```

   Then, we declare a variable of type ofstream (output-file-stream). ofstream is the data type, and you can give it whatever variable name:

   ```
   ofstream outputFile;
   ```

   Sometimes, your program might read multiple files at once, so then you'd declare more than 1 ofstream variable. But for here, we're just working with the one.

   We can then open a file name that we will be writing out to:

   ```
   outputFile.open( "save.txt" );
   ```

   Now we can output text to our file in the same way we use cout, except replacing cout with the outputFile variable...

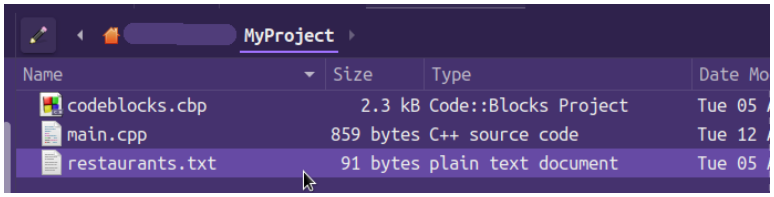
   ```
   outputFile << "Name: " << name << endl;
   ```

   If you want to close the file manually, use the .close() function:

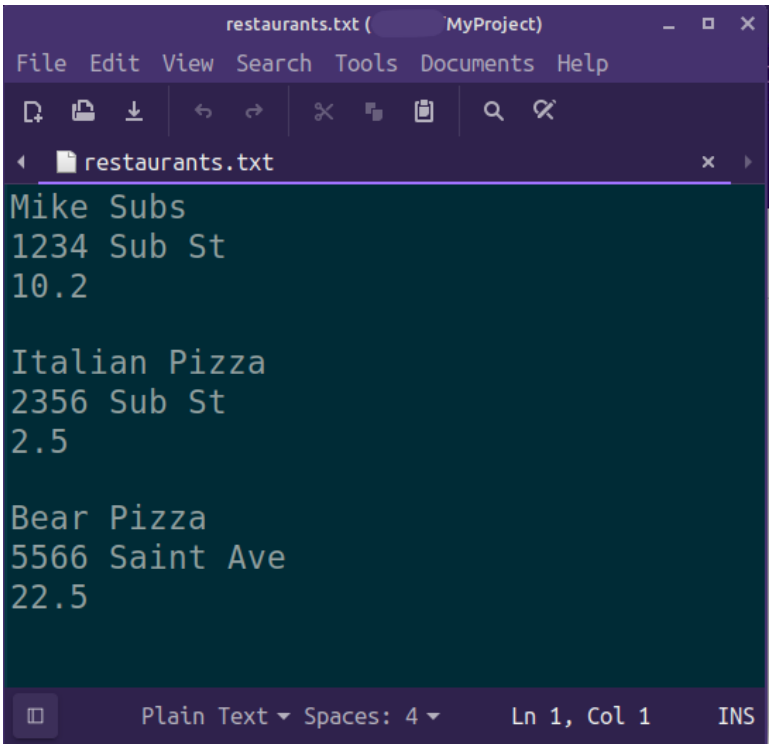
   ```
   outputFile.close();
   ```

   Otherwise, the ofstream object will close automatically when the program closes or when the function it's declared within exits.

2. The path of the output file



When you're saving a file with ofstream the default output path on your hard drive will be wherever your project file is, at least with Visual Studio and Code::Blocks. If you're building from the command line (with `g++`, for example), then the default base directory will be wherever your executable file is (e.g., `a.out`, `a.exe`).



You can open the text file output with a text editor like **notepad** or a code editor like VS Code. It's just plaintext!

3. Outputting different file types

   Many file formats are just plain text files if you open them in Notepad. You can save your program's output to more than just .txt files as well - you could use .csv (comma separated value) files, or .html files, or other file types as well.

   Just name your file a .csv or .html file when using the open function:

```
output.open( "data.html" );
```

   And then output text that is formatted as the correct file type:

```
output << "<h1>Q1 Data</h1>" << endl;
output << "<p>This year we found that <strong>potatoes</strong>" << endl;
output << "solid well, outselling tomatoes.</p>" << endl;
```

Any file type that is a plaintext file can be built as well - .html files, .csv files, heck, even .cpp files. However, *generally* if you wanted to write a program to output a different file type, you'd use a library to properly convert the data.

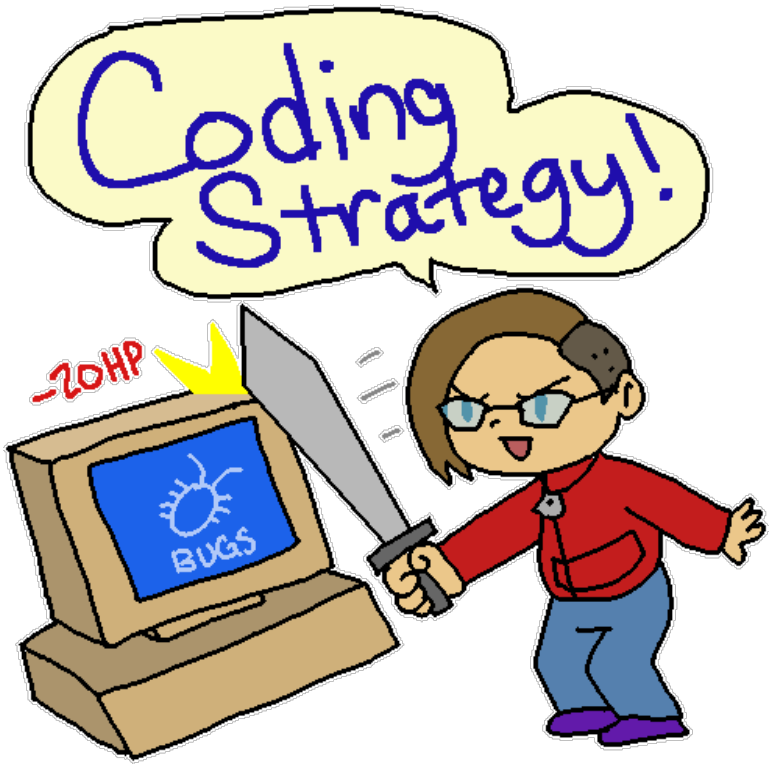
Example: Outputting HTML data:

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

int main()
{
  ofstream outputFile( "page.html" );
  outputFile << "<body>" << endl;
  outputFile << "<h1>This is a webpage</h1>" << endl;
  outputFile << "<p>Hello, world!</p>" << endl;
  outputFile << "</body>" << endl;
  outputFile.close();
}
```

Example: Outputting CSV data:

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

int main()
{
  ofstream outputFile( "spreadsheet.csv" );
  outputFile << "COLUMN1,COLUMN2,COLUMN3" << endl;
  outputFile << "cell1,cell2,cell3" << endl; // row 1
  outputFile << "cell1,cell2,cell3" << endl; // row 2
  outputFile << "cell1,cell2,cell3" << endl; // row 3
  outputFile.close();
}
```

4. Reading documentation



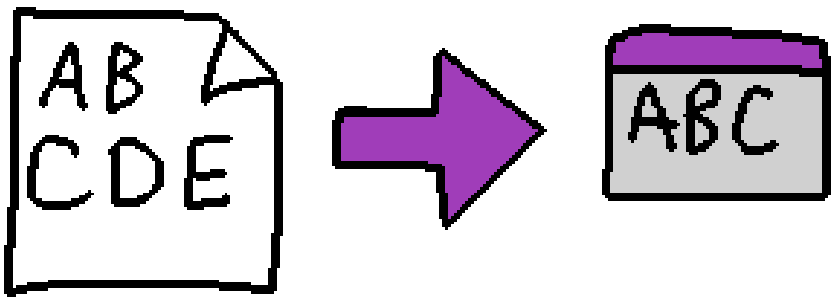
Documentation is an important part of software development. People who write code that's used by others need to provide documentation so that others can know how to use the functionality without having to read the source code.

The C++ ofstream library docs are here:

https://www.cplusplus.com/reference/fstream/ofstream/

Under the "Public member functions" header is a list of functions that ofstream provides, and clicking on a function will give you some example code and more information.

5.3.2 Input streams



We can use the **ofstream** library to write text and data out to text files. In the same family of tools, the **ifstream** library ("input file stream") allows us to read in files for use in our programs.

We could save/load save data files, such as in a video game, so you don't lose your process. You could also load in a document to "process" in some way.

Writing code to read in text is a little harder than just writing something out, because we have to tell our program how to read. **Ifstream** will bring in words or lines of text for us, but we have to tell our program how to make sense of what it's loaded in. This process is known as parsing.

File input streams work just line console input streams. You will need to create a `ifstream` (input-file-stream) object and open a file, and then you can read in the contents of that file. Files to be read should generally be placed in the same path as your .cpp file, though the working directory on your system may vary.

Example:

```
#include <fstream>              // File streams
#include <string>               // Strings
using namespace std;

int main()
{
  string data1, data2;

  // File input
  ifstream inputFile( "file.txt" );
  inputFile >> data1;             // Read one word
  inputFile.ignore();             // Clear buffer
  getline( inputFile, data2 );    // Read one line
  inputFile.close();
}
```

Just like with using `cin`, you can use the input stream operator `(>>)` and the `getline()` function with the file streams to get text. You will also need one or more variable to store the text read in.

1. Reading input

   We will use the same library to get access to both ofstream and ifstream. Remember "File Stream" is what "fstream" stands for.

   ```
   #include <fstream>
   ```

   Remember that ofstream was the data type of the variable we'd create to open a text file and write data out. For input, we use ifstream, but we still can open a file in the same way:

   ```
   ifstream inputFile;
   inputFile.open( "save.txt" );
   ```

We can read contents from our input file stream similarly to how we would work with cin for the keyboard input. Reading in one word or piece of data at a time will use the input stream operator »:

```
// Read from text file, save to playerExperience variable.
inputFile >> playerExperience;
```

And reading a whole line of text at a time will use getline:

```
getline( inputFile, playerName );
```

We still need to use the ignore() function when moving from a » statement to a getline statement:

```
inputFile >> playerHp;
inputFile.ignore();
getline( inputFile, playerName );
```

And, we can close the file manually if we'd like:

```
inputFile.close();
```

2. What if the file isn't found?

   When we're saving an file with ofstream we don't really have to worry about if the file already exists or not - if it doesn't exist, a new one will be created.

   For reading a file with ifstream, however, if no file exists to read, then what? Well, we should check to see if opening the file failed, and if it did, perhaps we use some default values instead, or give an error and exit the program.

   To check if the open process failed, we use an if statement:

```
ifstream input;
input.open( "userdata.txt" );
if ( input.fail() )
{
  cout << "ERROR: Could not find file!" << endl;
  return 2;
}
```

3. Reading an entire file

Reading chunks of data: Let's say you have a file full of data to read in. For this example, the file will be a list of numbers that we want to add together. The file might look something like....

File: Data.txt

```
9 15 16 0 10 13 5 16 1 9 2 17 3 3 8
```

In our program, we want to read all the numbers, but to do this, we need to use a **loop** and read in **one number at a time**. We can keep a running total variable to keep adding data to the sum as we go. We can use a loop to continue reading while it is successful, using this as the condition: `input >> readNumber`. This will stop the loop once there's nothing else to read, and it updates the `readNumber` variable with the input each cycle.

Example:

```
ifstream input( "data.txt" );

int sum = 0;    // Sum variable
int readNumber; // Buffer to store what we read in

// Keep reading while it's possible
while ( input >> readNumber )
  {
    sum += readNumber;  // Add on to the sum

    // Output what we did
    cout << "Read number " << readNumber
         << ",\t sum is now " << sum << endl;
  }

cout << "FINAL SUM: " << sum << endl;
```

File output:

```
Read number 9,   sum is now 9
Read number 15,  sum is now 24
(... etc ...)
Read number 3,   sum is now 119
Read number 8,   sum is now 127
FINAL SUM: 127
```

Reading lines of data: In other cases, maybe you're reading in text data from a file and want to read in a full line at a time. We can use a while loop with the `getline()` function as well to make sure we read each line of a text file:

Example:

```
ifstream input( "story.txt" );

string line;

while ( getline( input, line ) )
{
  cout << line << endl;
}
```

File output:

```
CHAPTER I.
Down the Rabbit-Hole


Alice was beginning to get very tired of sitting by her
sister on the bank, and of having nothing to do: once or
twice she had peeped into the book her sister was
reading, but it had no pictures or conversations in it,
"and what is the use of a book," thought Alice "without
pictures or conversations?"
```

4. Parsing plaintext files

   There are different ways you can design a text file's save format so that you can more easily read it in later on. Let's go over some common techniques. You might also want to take note of these for use in our class projects.

   A plaintext file is a file that just contains text, which can be opened in a program like notepad.

5. Saving and loading data

   (a) Parsing files Reading in data from a file is one thing, but making sense of what was read in is another. Are you storing saved data from the last session of the program? Are you trying to parse data to crunch? How do you read that data in logically?

   First, if your program is going to be *saving output* that will need to be read in and made sense of later on, how do you organize the data

that's output? It's up to you to make sure to structure the output save data in a consistent and readable way.

Let's say two developers come up with different formats to store a video game save file:

File: Output save game file (FORMAT 1)

```
RachelsGame
5
1000
RegnierCenter
```

File: Output save game file (FORMAT 2)

```
SAVEGAME        RachelsGame
LEVEL           5
GOLD            1000
LOCATION        RegnierCenter
```

- For **FORMAT 1**, we would have to make sure to write our program so that it always saves and loads in the specific order: GAME-NAME, LEVEL, GOLD, LOCATION.
- For **FORMAT 2**, since labels are added, we could read in a label, then decide whether the next thing to load in is the GAME-NAME, LEVEL, GOLD, or LOCATION. However, the extra information ("SAVEGAME", "LEVEL", "GOLD", "LOCATION") adds extra storage space to savegame file, and that information is mostly just useful to the humans reading it.

~

If every save file from the program is formatted in the same way (**FORMAT 1**), then when reading the file we can make assumptions...

- First word: Save game name - store in `gameFileName`.
- Second word: Player's level - store in `level`.
- Third word: Player's gold - store in `gold`.
- Fourth word: Player's location - store in `location`.

...And so on. This could work, but we could also take advantage of those human-readable labels that were added into the file (**FORMAT 2**):

- Read `firstWord`.
- If `firstWord` is "SAVEGAME", then read the next word into `gameFileName`.
  - Else if `firstWord` is "LEVEL", then read the next word into `level`.
  - Else if `firstWord` is "GOLD", then read the next word into `gold`.
  - Else if `firstWord` is "LOCATION", then read the next word into `location`.

So, let's say we have our four variables for a save game:

Name	Data type
gameFileName	string
level	int
gold	int
location	string

When we go to save our game file, here is what both formats' save process would look like:

Example: Saving for FORMAT 1

```
// Save the game
ofstream output( "save.txt" );
output << gameFileName << endl;
output << level << endl;
output << gold << endl;
output << location << endl;
```

File output: Saved game file:

```
SAVEGAME MyGame
LEVEL    1
GOLD     10
LOCATION OCB
```

Example: Saving for FORMAT 2

```
// Save the game
ofstream output( "save.txt" );
output << "SAVEGAME " << gameFileName << endl;
output << "LEVEL    " << level << endl;
output << "GOLD     " << gold << endl;
output << "LOCATION " << location << endl;
```

File output: Saved game file:

```
SAVEGAME MyGame
LEVEL    1
GOLD     10
LOCATION OCB
```

Now, to load in each format, we would have to take a couple of different approaches.

Example: Loading for FORMAT 1

For this format, the savegame data would have to always be in the same order, so we know we also load the data in the same order.

```
// Load the game
string buffer;
ifstream input( "save.txt" );
input >> gameFileName;
input >> level;
input >> gold;
input >> location;
```

Example: Loading for FORMAT 2

This one has the helper labels so we read in the first word, investigate what it says, and based on that decide which variable we're loading into next.

```
string buffer;
ifstream input( "save.txt" );

while ( input >> buffer )
{
  if ( buffer == "SAVEGAME" )
    input >> gameFileName;

  else if ( buffer == "LEVEL" )
    input >> level;

  else if ( buffer == "GOLD" )
    input >> gold;

  else if ( buffer == "LOCATION" )
    input >> location;
}
```

If we stepped through this, `buffer` is always going to store one of the data labels, because after buffer is read, we immediately read the second item in the line of text. Once the second item is read, the next thing to be read will be the next label.

5.3.3 Review questions:

1. What library is required in order to use file I/O in C++?
2. What data type is used to create a file that **inputs** (reads in) text from an outside file?
3. What data type is used to create a file that **outputs** (writes out) text to an outside file?
4. How do you write out the string literal "Hello world" to an output file?

5. How do you write out the value of a variable to an output file?
6. How do you read in **one word** from an input file?
7. How do you read in **one line** from an input file?

5.4 Lab: Strings and File I/O

5.4.1 Assignment information

- **Turning in your work:**
  1. In VS Code, build and run the graded program(s) in the **Terminal**. Take a screenshot of the program running and save it to your computer.
  2. In VS Code, make sure to COMMIT and SYNC your changes to the GitLab repository. (See: Using Git and VS Code).
  3. Go to your GitLab repository page, make sure it shows your latest COMMIT MESSAGE and your code is up-to-date. (Important!! I can't build and run your code if it's not on the server!!)
  4. In Canvas, go to the **Assignment** and click **Start Assignment**. Upload the screenshot of your program running. In the comments, let me know if you're done with all of the lab or just part of it and if you have any questions.
- **Don't utilize future topics!** - We will revisit the graded program here once we cover Exceptions later on. Please do not use exceptions for this assignment.
- **Practice programs & graded programs:** Completing the graded program(s) is worth 90% of the lab grade. Completing the practice programs is worth an additional 10%.

Links:

- How to turn in assignments on Canvas
- Using Git and VS Code
- Program arguments
- Assignment direct link

5.4.2 Included files:

```
wk04_StringsAndFileIO
 graded_program
    account1.txt
    bank.cpp
    tester.cpp
 instructions.org
 practice1_outfile
    outfile.cpp
 practice2_cinwords
    cin.cpp
 practice3_cinlines
    todo.cpp
```

```
practice4_infilewords
   alice.txt
   readwords.cpp
practice5_infilelines
   alice.txt
   readlines.cpp
practice6_readentirefile
    alice.txt
    readwhole.cpp
```

5.4.3 Practice programs

1. Practice 1 - Writing OUT to a text file

   (a) Reference
   To write out to a text file, first we need to create an output file stream variable:

   ```
   ofstream FILEVARIABLE;
   ```

   We can then open a file using its `open` function:

   ```
   FILEVARIABLE.open( "file.txt" );
   ```

   We can output text to the file just like we do with `cout`, except we replace "cout" with the name of the output file variable.

   ```
   FILEVARIABLE << "Hello!" << endl;
   ```

   Once we're done, we can close the file with its `close` function:

   ```
   FILEVARIABLE.close();
   ```

   (b) Example output
   Program output:

   ```
   ./a.out

   -- NATIVE FLOWERS v1 --
   Not enough arguments! Expected: ./a.out FLOWERNAME
   Flowers are coneflower, aster, phlox

   ./a.out coneflower

   -- NATIVE FLOWERS v1 --
   Plant information saved to flower-info.txt

   -- GOODBYE! --
   ```

   File output, flower-info.txt:

```
PURPLE CONEFLOWER
SUN:    Full
SOIL:   Dry/moderate
NATURE: Butterfly, pollinators
HEIGHT: 24 - 36 inches
SPREAD: 12 - 16 inches
```

---

2. Practice 2 - Console input with »

   (a) Reference

   We can read text from the user's keyboard into a variable using the `cin` (console-in) command and the input stream `>>` operator. We can use this to load integers, floats, booleans, doubles, strings, chars, etc.

   ```
   cin >> VARIABLE;
   ```

   (b) Example output

   Program output:

   ```
   ./a.out

   -- NATIVE FLOWERS v2 --
   SELECT A FLOWER:
   1. Eastern Blazing Star
   2. Bee Balm
   3. Butterfly Milkweed

   CHOICE: 1
   Plant information saved to flower-info.txt

   -- GOODBYE! --
   ```

   File output, flower-info.txt:

   ```
   EASTERN BLAZING STAR
   SUN:    Full/medium
   SOIL:   Dry/moderate
   NATURE: Butterfly, hummingbird, pollinators
   HEIGHT: 30 - 48 inches
   SPREAD: 10 - 18 inches
   ```

---

3. Practice 3 - Console input with getline

   (a) Reference

   We can read in a whole line of text from the keyboard and store it in a **string** variable using the `getline` function. This only works for strings, but will allow you to enter a string with spaces in it. (`cin >> STR;` won't allow spaces!)

```
getline( cin, VARIABLE );
```

(b) Example output
Program output:

```
./a.out

-- TO DO LIST --
Enter the next item on your TODO list, or QUIT to end:
>> eat breakfast
Enter the next item on your TODO list, or QUIT to end:
>> eat lunch
Enter the next item on your TODO list, or QUIT to end:
>> eat dinner
Enter the next item on your TODO list, or QUIT to end:
>> QUIT
List written to todo.txt

-- GOODBYE! --
```

File output, todo.txt:

```
1. eat breakfast
2. eat lunch
3. eat dinner
```

---

4. Practice 4 - Reading IN from file with »

(a) Reference
To read in from a text file we need to create an input file stream variable:

```
ifstream FILEVARIABLE;
```

We can then open a file using its `open` function:

```
FILEVARIABLE.open( "file.txt" );
```

For input files we should check to see if the `fail` function returns true... if it failed, that means the file wasn't found.

```
if ( FILEVARIABLE.fail() )
{
    cout << "ERROR: Couldn't find file!" << endl;
    return 1;
}
```

We can input data from the text files in the same manner as with `cin >> VARIABLE`:

```
FILEVARIABLE >> OTHERVARIABLE;
```

Once we're done, we can close the file with its `close` function:

```
FILEVARIABLE.close();
```

(b) Example output

```
./a.out

-- READ WORDS PROGRAM --
Not enough arguments! Expected: ./a.out FILENAME


./a.out alice.txt

-- READ WORDS PROGRAM --
READ: Alice
READ: was
READ: beginning

-- GOODBYE! --
```

---

5. Practice 5 - Reading IN from file with getline

(a) Reference

Once we have an input file variable we can also use the `getline` function to read in whole lines of text at a time, similar to with the `cin` command:

```
getline( FILEVARIABLE, STRINGVARIABLE );
```

(b) Example output

```
./a.out

-- READ LINES PROGRAM --
Not enough arguments! Expected: ./a.out FILENAME


./a.out alice.txt

-- READ LINES PROGRAM --
READ: Alice was beginning to get very tired of sitting by her sister on the
READ: bank, and of having nothing to do: once or twice she had peeped into
READ: the book her sister was reading, but it had no pictures or

-- GOODBYE! --
```

---

6. Practice 6 - Reading IN an entire file

For this program we're going to put our `getline` function call within a `while` loop. This way, it will read every line of the file - one line at a time.

(a) Reference

Reading in an entire file's contents one line at a time:

```
while ( getline( infile, VARIABLE ) )
{
    // VARIABLE holds one line from the document
}
```

(b) Example output

```
./a.out

-- READ WHOLE FILE PROGRAM --
Not enough arguments! Expected: ./a.out FILENAME
./a.out alice.txt

-- READ WHOLE FILE PROGRAM --
1: Alice was beginning to get very tired of sitting by her sister on the
2: bank, and of having nothing to do: once or twice she had peeped into
3: the book her sister was reading, but it had no pictures or
4: conversations in it, “and what is the use of a book,” thought Alice
5: without pictures or conversations?”

-- GOODBYE! --
```

5.4.4 Graded programs

1. Graded program 1: Bank account

   In this program we're going to use a saved file as our saved data. When we close the program, it will **SAVE** the program data, and when we run the program it will either **LOAD** the program data if it exists, or if it doesn't exist it will create the data from defaults. By using a data save file, each time we run the program we can retrieve our data and keep working with it, instead of having to re-enter our data from scratch.

   (a) Steps

   At the `// STARTUP: LOAD DATA FILE` area, the `infile` opens the filename given.

   - First we're going to check if the data file exists or not:
     i. Check to see if opening the `infile` failed, if it failed (is true), set `new_account` to true.
     ii. If it didn't fail (fail is false) then `new_account` is false.
   - Now we'll write the code for if we need to create a new account or not:
     i. If `new_account` is true, then we ask the user to enter their account type, account number, and starting input balance.
     ii. Add `cin` statements for each of these fields.

- Otherwise, if a new account is not needed, then we can load the contents of the file into the variables:
  i. Read from `infile` into `account_name`, then `account_number`, then `account_balance`.

After the load is done, we ask the user how much they want to deposit into their account.

- Add a `cin` statement to get their `deposit` amount.

Finally, when the program is done, it will save the data before closing. `outfile` opens the filename.

- Output the account data into the file, each variable on its own line:
  i. Output `account_name` to the `outfile`, then `account_number`, then `account_balance`.

(b) Example output

Program output:

```
./a.out account3.txt

-- ATM PROGRAM --
No save file found! Creating new account...
Enter account type (checking/savings): checking
Enter account number: 1337
Enter starting balance: 10000

-- WELCOME TO BAINC BANK --
YOUR ACCOUNT...checking
ACCOUNT #......1337
BALANCE $......10000.00

Enter amount of money to deposit to account: $500
You now have a balance of $10500.00

Thanks for banking with BAINC! Come again soon!

Account updates saved to "account3.txt"

-- GOODBYE! --



./a.out account3.txt

-- ATM PROGRAM --
Loading account...

-- WELCOME TO BAINC BANK --
YOUR ACCOUNT...checking
ACCOUNT #......1337
```

```
BALANCE $......10500.00

Enter amount of money to deposit to account: $200
You now have a balance of $10700.00

Thanks for banking with BAINC! Come again soon!

Account updates saved to "account3.txt"

-- GOODBYE! --
```

File output, account3.txt:

```
checking
1337
10700
```

---

2. Graded program 2: String library

   Within the set of if/else if statements in main() you'll utilize several functions from the **string** library. Utilize the documentation below for example code and explanations of how the functions work.

   (a) Steps

   - // **TODO: Use the size function**
     - Documentation: https://cplusplus.com/reference/string/string/size/
     - This function returns the amount of **characters** within a **string**, which includes letters, numbers, symbols, and spaces.
     - Form:

       ```
       size_t length = SOMESTRING.size();
       ```
   - // **TODO: Use the subscript operator**
     - Documentation: https://cplusplus.com/reference/string/string/operator[]/
     - This function returns the single `character` in the string at that index, where 0 is the very first letter in the string, and the last letter is at `.size()-1`.
     - Form:

       ```
       char single = STR[INDEX];
       ```
   - // **TODO: Use the find function**
     - Documentation: https://cplusplus.com/reference/string/string/find/
     - This function returns the position where the searched for substring is found, or `string::npos` if not found.
     - Form:

       ```
       size_t foundpos = STR.find( FINDMESTR );
       // foundpos will equal string::npos if nothing found, otherwise will
       ```

- // **TODO: Use the compare function**
  - Documentation: https://cplusplus.com/reference/string/string/compare/
  - This function compares two strings and returns a negative number if (case-sensitive) alphabetically STR1 comes before STR2, a positive number if STR1 comes after STR2, and a 0 if they're the same string.
  - Form:

```
int result = STR1.compare( STR2 );
// -1: STR1 is "less than" STR2
// 0: STR1 and STR2 are the same
// 1: STR1 is "greater than" STR2
```

- // **TODO: Create an if/else if statement using <, >, and ==**
  - Documentation: https://cplusplus.com/reference/string/string/operators/
  - These functions return `true` or `false` based on the comparison. Using STR1 < STR2 will check to see if STR1 comes before STR2 alphabetically (case-sensitive), and so on.
  - Form:

```
if ( STR1 < STR2 )        // ...
else if ( STR1 > STR2 )   // ...
else if ( STR1 == STR2 ) // ...
```

- // **TODO: Use the concatenation operator +=**
  - Documentation: https://cplusplus.com/reference/string/string/operator+/
  - This function appends one string to another.
  - Form:

```
string NEWSTR = STR1 + STR2;
STR1 += STR2;
```

- // **TODO: Use the insert function**
  - Documentation: https://cplusplus.com/reference/string/string/insert/
  - This function allows you to insert a substring into an original string at some starting position/index.
  - Form:

```
string NEWSTR = OLDSTR.insert( POSITION, UPDATESTR );
```

- // **TODO: Use the erase function**
  - Documentation: https://cplusplus.com/reference/string/string/erase/
  - This function allows you to erase some amount of characters from an original string starting at some position/index.
  - Form:

```
string NEWSTR = OLDSTR.erase( POSITION, LENGTH );
```

- // **TODO: Use the replace function**

- Documentation: https://cplusplus.com/reference/string/string/replace/
- This function allows you to erase an amount of characters in the original string at some position/index and replace them with a new substring.
- Form:

```
string NEWSTR = OLDSTR.replace( POSITION, LENGTH, UPDATESTR );
```

(b) Example output

```
TEXT:
--------------------------------------------------------------------------------
the quick brown fox jumps over the lazy dog
--------------------------------------------------------------------------------

- INFO --------
1. Get string length
2. Get letter #
3. Find substring
4. Compare strings
5. String relations

- MODIFY --------
6. Combine strings
7. Insert string
8. Erase from string
9. Replace in string

0. QUIT

 *Option 1:*
    The length of the string is... 43

 *Option 2:*
    Enter an index: 4
    Letter #4 is... q

 *Option 3:*
    Enter text to search for: jump
    String found at position... 20

 *Option 4:*
    Enter a second string: the slow tortoise
    Comparison result... -2

 *Option 5:*
    Enter a second string: the slow tortoise
    Comparison result... <
```

```
*Option 6:*
   Enter a second string: , whee!
   Text is now: the quick brown fox jumps over the lazy dog, whee!

*Option 7:*
   Enter a second string: slow
   Enter an index to insert the string: 4
   Text is now: the slowquick brown fox jumps over the lazy dog, whee!

*Option 8:*
   Enter an index to begin removing from: 8
   Enter # of characters to remove: 5
   Text is now: the slow brown fox jumps over the lazy dog, whee!

*Option 9:*
   Enter an index to begin removing from: 9
   Enter # of characters to remove: 5
   Enter string to add in its place: purple
   Text is now: the slow purple fox jumps over the lazy dog, whee!
```

6 Week 5: Structs

6.1 Intro: Structs

6.1.1 Introduction to objects

Programming paradigms (pronounced "pair-uh-dimes") are ways we can classify different programming styles. Some programming languages support multiple paradigm styles and some are restricted to one style of coding. Over the decades different paradigms have been developed and evolved over time. **Object Oriented Programming** is one of the most common styles of programming these days, and is a big part of how C++, C#, and Java-based programs are designed.

1. What are "Objects"?

   Defining **classes** and **structs** in our programs are a way that we can create our own **data types** for our variables. When we define our own structures, we can create them with **internal variables and functions** available to them. The variables we create, whose data types come from a defined **class or struct**, is known as an **object**.

   Design-wise, the idea is to take real-world objects and find a way to *represent them* in a computer program as an **object** that has **attributes** (variables) and **functionality** (functions).

2. OOP design ideals

   The concept of **Object Oriented Programming** is creating a design that is easy to maintain over time. In particular, there are some core design goals behind OOP, including:

- **Encapsulation:** Giving the user / other programmers an "interface" \ to interact with the objects, hiding the inner-workings within the class.
  - Certain functions are made **public**, which other programmers can use to interface with the object.
  - The other programmers don't need to worry about the inner-workings of the object in order to use it.
  - The developer of the class can modify *how* the internals work without breaking the public interface.
  - Helps protect the data within the class from being accessed or modified by external things that shouldn't have access to it.
- **Loose coupling:** Ideally, different objects in a program shouldn't have their functionality tied to other objects too closely; we want to reduce inter-dependence between objects. When objects are more *independent* from each other, we say they are *loosely coupled*.
- **High cohesion:** When we design our objects, we shouldn't just throw everything and the kitchen sink into one object. To design an object with *high cohesion* means that everything inside the object *belongs* to that object - reduce the clutter.

6.1.2 Separate variables to one struct

Structs are a way we can group related information together into one data type.

For example, let's say we were writing a program for a restaurant, and an item on the menu would have a name, a price, a calorie count, and a "is it vegetarian?" signifier. We could declare these all as separate variables:

```
string food1_name;
float food1_price;
int food1_calories;
bool food1_is_veggie;
```

But there isn't really anything in the program that says these variables are related, except that we as humans have given these variables similar prefixes in their names.

A better design would be to create a new datatype called MenuItem (or something similar), and the MenuItem will contain these four variables within it. We're basically making a variable data type that can contain multiple variables!

First we create the struct, which should go in its own .h file:

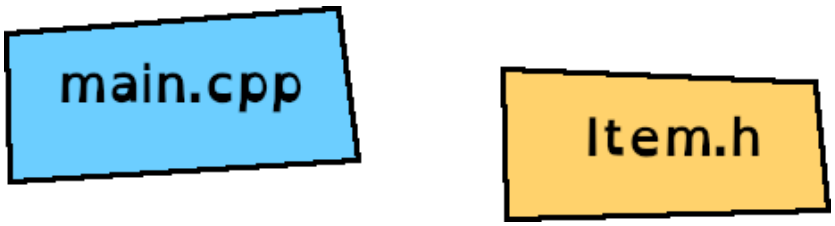
Example: MenuItem.h

```
struct MenuItem
{
  string name;
  float price;
  int calories;
  bool isVeggie;
};
```

And then we can declare variables of this type in our program:

```
MenuItem food1;
food1.name = "Bean Burrito";
food1.price = 1.99;
food1.calories = 350;
food1.isVeggie = true;

MenuItem food2;
food2.name = "Crunchy Taco";
food2.price = 1.99;
food2.calories = 170;
food2.isVeggie = false;
```

6.1.3 Multiple files in C++ programs



When creating a struct, we should create a new source code file in our project. Usually the name of the file will be the same name as the struct, with a .h at the end. This is a header file and declarations for structs (and functions and classes later) will go in these types of files.

Something you need to require in your .h files that isn't needed for your .cpp files are file guards. These prevent the compiler from reading the file more than once. If it reads it multiple times, it will think you're redeclaring things over and over again. Your .h files should always look like this:

```
#ifndef _MYFILE_H
#define _MYFILE_H
```

```
// put code here

#endif
```

Where the "_MYFILE_H" should be changed to something unique in each file - usually, the name of your file.

6.1.4 Creating our own structs

A basic struct declaration is of the form...

```
#ifndef _MYFILE_H
#define _MYFILE_H

struct STRUCTNAME
{
  // member variables
  int MEMBER1;
  string MEMBER2;
  float MEMBER3;
};

#endif
```

Note that the closing curly-brace } must end with a ; otherwise the compiler will give you errors.

You can name your struct anything you'd please but it has to abide by the C++ naming rules (no spaces, no keywords, can contain letters, numbers, and underscores).

Any variables we declare within the struct are known as **member variables** - they are members of that struct.

Any new variables you declare whose data type is this struct will have its own copy of each of these variables.

1. Declaring object variables

   A object is a type of variable whose data type is a struct (or class). To declare a variable whose data type is a struct, it looks like any normal variable declaration:

   ```
   DATATYPE VARIABLE;
   ```

2. Accessing member variables

   Our **struct object variables** each have their own **member variables**, which we access via the dot operator . . We can assign values to it, our read its value, similar to any normal variable.

   Example:

```
VARIABLE.MEMBER1 = 10;
VARIABLE.MEMBER2 = "ASDF";
VARIABLE.MEMBER3 = 2.99;

cout << VARIABLE.MEMBER1 << endl;
```

3. Example: Fraction struct

   First we would declare our Fraction struct within **Fraction.h:**

```
#ifndef _FRACTION_H
#define _FRACTION_H

struct Fraction
{
  int num;
  int denom;
};

#endif
```

   Within `main()`, we could then create Fraction variables and work with them:

```
  #include "Fraction.h"

int main()
{
  Fraction frac1, frac2, frac3;

  cout << "FIRST FRACTION" << endl;
  cout << "* Enter numerator: ";
  cin >> frac1.num;
  cout << "* Enter denominator: ";
  cin >> frac1.denom;

  cout << endl;
  cout << "SECOND FRACTION" << endl;
  cout << "* Enter numerator: ";
  cin >> frac2.num;
  cout << "* Enter denominator: ";
  cin >> frac2.denom;

  frac3.num = frac1.num * frac2.num;
  frac3.denom = frac1.denom * frac2.denom;
  cout << endl;
  cout << "PRODUCT: " << frac3.num << "/" << frac3.denom << endl;

  return 0;
}
```

6.1.5 Structs with other Structs as member objects

In addition to creating structs with normal variables, we can also create struct-object-variables as members of another struct.

Let's say we have a **Teacher** struct and a **Course** struct, for each Course we want to store the teacher of that course...

Teacher.h:

```
#ifndef _TEACHER_H
#define _TEACHER_H

#include <string>
using namespace std;

struct Teacher
{
  string name;
  string email;
};

#endif
```

Course.h:

```
#ifndef _COURSE_H
#define _COURSE_H

#include "Teacher.h"

#include <string>
using namespace std;

struct Course
{
  Teacher teacher;
  string department;
  int code;
  int hours;
};

#endif
```

6.1.6 Review questions:

1. File guards are needed because...
2. What kind of data would be good to represent with a struct?

- A character in a video game including the amount of experience points they have, their current hit points, and current level
- The tax rate of a city
- The URL of a website you're visiting
- An address, including recipient name, street address, city, state, and zip code

3. If we declared an object variable like `Course cs200;` , How would we assign "cs" to the `department` member variable and 200 to the `code` member variable?

6.2 Lab: Structs

6.2.1 Assignment information

- **Turning in your work:**
  1. In VS Code, build and run the graded program(s) in the **Terminal**. **Take a screenshot of the program running** and save it to your computer.
  2. In VS Code, make sure to COMMIT and SYNC your changes to the GitLab repository. (See: Using Git and VS Code).
  3. Go to your GitLab repository page, make sure it shows your latest COMMIT MESSAGE and your code is up-to-date. (Important!! I can't build and run your code if it's not on the server!!)
  4. In Canvas, go to the **Assignment** and click **Start Assignment**. **Upload the screenshot of your program running**. In the comments, let me know if you're done with all of the lab or just part of it and if you have any questions.
- **Don't utilize future topics!** - We will revisit the graded program here once we cover Exceptions later on. Please do not use exceptions for this assignment
- **Practice programs & graded programs:** Completing the graded program(s) is worth 90% of the lab grade. Completing the practice programs is worth an additional 10%.
- **Links:**
  - How to turn in assignments on Canvas
  - Using Git and VS Code
  - Program arguments
  - Debugging with gdb / lldb
  - Assignment direct link

6.2.2 Included files:

```
wk05_Structs
 graded_program
    pet.cpp
 instructions.html
 instructions.org
 practice1_struct
    bus.cpp
 practice2_struct2
     recipe.cpp
```

6.2.3 Practice programs

1. Practice 1 - Bus Stop struct

   With this program the `BusStop` struct has already been declared above `main()`.

   Beneath the `// TODO: Create BusStop variables` note, create 3 variables whose data types are `BusStop`. Set up each of the variables as follows:

Variable	id	name	lat	lon
stop1	510192	KU LAWRENCE - BECKER DRIVE #375	38.945608	-95.262773
stop2	25053	ON TROOST AT 61ST SOUTHBOUND	39.016731	-94.574434
stop3	23515	ON 35TH AT CLEVELAND WESTBOUND	39.062246	-94.539234

   (Or you can customize the data if you want!)

   Further down, I've coded the header of a table:

```
cout << left
     << setw( 10 ) << "ID"
     << setw( 15 ) << "LATITUDE"
     << setw( 15 ) << "LONGITUDE"
     << setw( 50 ) << "STOP NAME"
     << endl << string( 80, '-' ) << endl;
```

   When the program is run, it will look like this:

```
ID        LATITUDE       LONGITUDE      STOP NAME
--------------------------------------------------------------------------------
```

   The `setw` function sets the column width of each field, and `left` sets it to left align (the default is right aligned). Below this section of code, follow the same cout statement style but instead of writing string literals like `"ID"`, `"LATITUDE"`, etc, write each BusStop variable's data, like `stop1.id`, `stop1.lat`, and so on.

   (a) Reference

   Declaration of a struct:

```
struct STRUCTNAME
{
    // Member variables
    string STRVAR;
    float FLOATVAR;
    int INTVAR;
}; // semicolon reqiured!
```

   Declaring a variable whose type is that struct:

```
STRUCTNAME VARIABLENAME;
```

Accessing members of the struct via the variable:

```
VARIABLENAME.STRVAR = "string";
cout << VARIABLENAME.FLOATVAR << endl;
```

(b) Example output

```
-- BUS STOP PROGRAM --
ID        LATITUDE       LONGITUDE      STOP NAME
--------------------------------------------------------------------------------
510192    38.95          -95.26         KU LAWRENCE - BECKER DRIVE #375
25053     39.02          -94.57         ON TROOST AT 61ST SOUTHBOUND
23515     39.06          -94.54         ON 35TH AT CLEVELAND WESTBOUND

-- GOODBYE! --
```

---

2. Practice 2 - Recipe and Ingredient structs

In this program two structs are declared at the top. The `Ingredient` struct contains the following members:

- `string name` - the name of the ingredient (e.g., Peanut Butter)
- `float amount` - the amount of the ingredient to use (e.g., 1)
- `string unit` - the name of the measurement unit (e.g., "cups")

Beneath it is the `Recipe` struct. It has its own `name`, as well as 3 `Ingredient` variables *inside* of it. The Recipe *contains* three Ingredients.

Within `main()`, declare a `Recipe` variable and set up its name and its Ingredient data. If you want, you can use this example recipe:

Recipe name: Peanut butter cookies

Ingredient	Name	Amount	Unit
ing1	peanut butter	1	cups
ing2	sugar	0.5	cups
ing3	egg	1	total

After setting up all of the data, display the recipe and its ingredients to the screen. See the example output below.

(a) Reference

Accessing a normal member variable of a struct-object:

```
MYSTRUCT.MYVAR = 100;
```

Accessing a variable inside a struct-object inside a struct-object:

```
MYSTRUCT.SUBSTRUCT.MYVAR = "Hello";
```

(b) Example output

```
./recipe.exe 3

-- COOKIE PROGRAM --
RECIPE: Peanut butter cookies

INGREDIENTS (3.00 batches)
1. 3.00 cups of peanut butter
2. 1.50 cups of sugar
3. 3.00 total of egg

-- GOODBYE! --
```

6.2.4 Graded programs

1. Graded program: Virtual pet

   With this program you'll code a simple virtual pet, like a Tamagatchi.

   (a) Reference

   Create a random integer between 0 and 9:

   ```
   int VAR = rand() % 10
   ```

   Create a random integer between 1 and 10:

   ```
   int VAR = rand() % 10 + 1
   ```

   (b) Instructions

   You can customize the pet data if you'd like.

   - Within the game loop (`while ( running )`), first display the pet's name and their current stats (health, happiness, hunger).

   Next the user will select an action from a numbered menu.

   - **Within choice 1:** Add the `random` value to the pet's `happiness` variable. Also display a message like "You play with [NAME]! +[RAND] happiness!".
   - **Within choice 2:** Subtract the `random` value from the pet's `hunger` variable. Also display a message like "You feed [NAME]! -[RAND] hunger!".
   - **Within choice 3:** Add the `random` value to the pet's `health` variable. Also display a message like "You gave [NAME] medicine! +[RAND] health!".

   Further down in the while loop the pet's hunger will go up a little each cycle, and its health will go down a little each cycle. After the stat update, make sure to display warning messages to the player if the pet's stats are in a bad range:

   - **If pet health is less than 50** display a warning ("WARNING! Low health!") and also subtract a random amount from the pet's `happiness`.

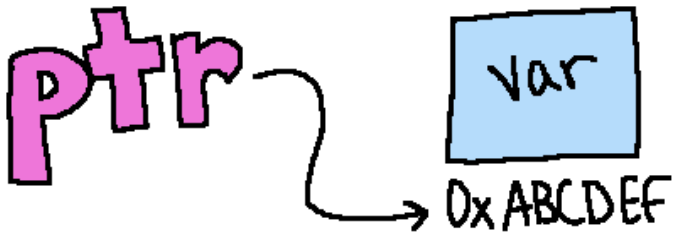
- **If pet hunger is greater than 50** display a warning ("WARNING! High hunger!") and also subtract a random amount from the pet's `health` and `happiness`.

(c) Example output

```
--------------------------------------------------------------------------------
PET: Petty McPetFace
Health: 47%     Happiness: 100%         Hunger: 52%

OPTIONS:
1. Play
2. Feed
3. Medicine

CHOICE: 1
You play with Petty McPetFace
+3 happiness!
WARNING! Low health!
WARNING! High hunger!
```

7 Week 6: Pointers

7.1 Intro: Pointers and memory



7.1.1 Bits and Bytes

When we declare a variable, what we're actually doing is telling the computer to set aside some **memory** (in RAM) to hold some information. Depending on what data type we declare, a different amount of memory will need to be reserved for that variable.

Data type	Size
boolean	1 byte
character	1 byte
integer	4 bytes
float	4 bytes
double	8 bytes

A **bit** is the smallest unit, storing just 0 or 1.

A **byte** is a set of 8 bits. With a byte, we can store numbers from 0 to 255, for an *unsigned* number (only 0 and positive numbers, no negatives).

The minimum possible value for 1 byte is 0.
(Decimal value = $128 \cdot 0 + 64 \cdot 0 + 32 \cdot 0 + 8 \cdot 0 + 4 \cdot 0 + 2 \cdot 0 + 1 \cdot 0$)

place	128	64	32	16	8	4	2	1
value	0	0	0	0	0	0	0	0

The maximum possible value for 1 byte is 255.
(Decimal value = $128 \cdot 1 + 64 \cdot 1 + 32 \cdot 1 + 8 \cdot 1 + 4 \cdot 1 + 2 \cdot 1 + 1 \cdot 1$)

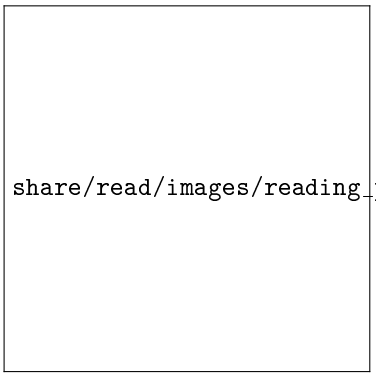
place	128	64	32	16	8	4	2	1
value	1	1	1	1	1	1	1	1

Some data types, like a Boolean and a Character, use only 1 byte. But others, like Integers and Floats, use more. Since an integer uses 4 bytes, that

means it has $8 \cdot 4 = 32$ bits available to store data. $2^{32} = 4,294,967,296$, so floats and integers can store this many *different* values. (Note that this doesn't mean that an integer goes from 0 to 4,294,967,296, because we have to account for negative values.)

7.1.2 Memory addresses

Whenever a **variable is declared**, we need space to store what its **value** is. We have already looked at how many *bytes* a data type takes up, but where is the variable's data stored? – In *working memory*. You can think of this as the RAM, though the Operating System interacts with the RAM and gives us a "virtual memory space" to work with. But, to keep it simple, we will think of this as the RAM.



Each block of space in memory has a **memory address**, one after another...

Bit address	0	1	2	3	4	5	6	7

Though to the computer, it represents these addresses in **binary** (base 2) or **hexadecimal** (base 16):

Address	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07

Hexadecimal goes from 0 to 16, but values from 10 and up are represented with letters. This is so each "number" in the value only takes 1 character (10 is two characters: 1 and 0). So, a quick reference for hexadecimal is like this:

Base 16 (Hex)	A	B	C	D	E	F
Base 10	10	11	12	13	14	15

Example: Variable in memory

Let's say we're investigating some blocks of memory.

If we declare a `char`, which gets 1 byte, it will take up any available memory where 8 bits are available side-by-side. Some blocks in memory might already be taken, so whatever is available will be used:

Address	Variables (BEFORE)		Address	Variables (AFTER)
0x0	(used)		0x0	(used)
0x1	(used)		0x1	(used)
0x2			0x2	**char1**
0x3			0x3	**char1**
0x4			0x4	**char1**
0x5			0x5	**char1**
0x6			0x6	**char1**
0x7			0x7	**char1**
0x8			0x8	**char1**
0x9			0x9	**char1**
$0xA_{(10)}$	(used)		$0xA_{(10)}$	(used)
$0xB_{(11)}$	(used)		$0xB_{(11)}$	(used)
$0xC_{(12)}$			$0xC_{(12)}$	
$0xD_{(13)}$			$0xD_{(13)}$	
$0xE_{(14)}$	(used)		$0xE_{(14)}$	(used)
$0xF_{(15)}$	(used)		$0xF_{(15)}$	(used)

In this case, declaring our variable,

```
char char1 = 'A';
```

its data will be placed with its first bit at address `0x2`.

We can use the **address-of operator** `&` to see what any variable's address in memory is:

Example: Displaying the `char1` variable's value and its memory address

```
cout << "Value:   " << char1 << "\t";
cout << "Address: " << &char1 << endl;
```

Getting the *address-of* a variable will return the address of its first bit, so the output here would be:

Program output:

```
Value:   A          Address: 0x2
```

(Again, keep in mind that the representation of addresses here is simplified.)

Example: Variable value in memory

We can see where the variable is stored at in memory, but let's look at how it's value will be stored as well. Given the declaration:

```
char char1 = 'A';
```

The value of `'A'` is stored in 1 byte. Technically, our computer stores the letter "uppercase A" as the number code `65`. This can be converted into binary: $(65)_{10}$ = `0100 0001`. This is the data that would be stored in that memory address.

Address	Variables	Value
0x0	(used)	
0x1	(used)	
0x2	**char1**	0
0x3	**char1**	1
0x4	**char1**	0
0x5	**char1**	0
0x6	**char1**	0
0x7	**char1**	0
0x8	**char1**	0
0x9	**char1**	1
$0xA_{(10)}$	(used)	
$0xB_{(11)}$	(used)	
$0xC_{(12)}$		
$0xD_{(13)}$		
$0xE_{(14)}$	(used)	
$0xF_{(15)}$	(used)	

This is just a basic representation, again. One thing you'll learn more about in future Computer Science courses is *endian-ness*, such as whether a number has its most-significant bit on the left side or the right side. We're not worrying about that here. :)

7.1.3 Pointer variables

Pointer variables are another type of variable, but instead of storing a value like `"ABC"`, `'X'`, `10.35`, or `4`, it stores a **memory address** instead.

1. Pointer declaration

   When we declare a normal variable, we have to specify its **data type**. With a pointer variable, we specify the **data type** of the data it's pointing to, plus the `*` character (after the data type) to show that this is a pointer variable.

   Declaration forms

- `DATATYPE* PTRNAME;`
- `DATATYPE * PTRNAME;`
- `DATATYPE *PTRNAME;`
- `DATATYPE* PTRNAME = nullptr;`
- `DATATYPE* PTRNAME{nullptr};`

The pointer variable declaration takes the same form as a normal variable's declaration *except* we have to put the asterisk `*` after the data type. The asterisk can be attached to the data type, the name, or free-standing, it doesn't really matter, but *I* prefer keeping it with the data type.

2. Pointer assignment

   Once we have a pointer, we can point it to the address of any variable with a matching data type. To do this, we have to use the **address-of** operator to access the variable's address - this is what gets stored as the pointer's value.

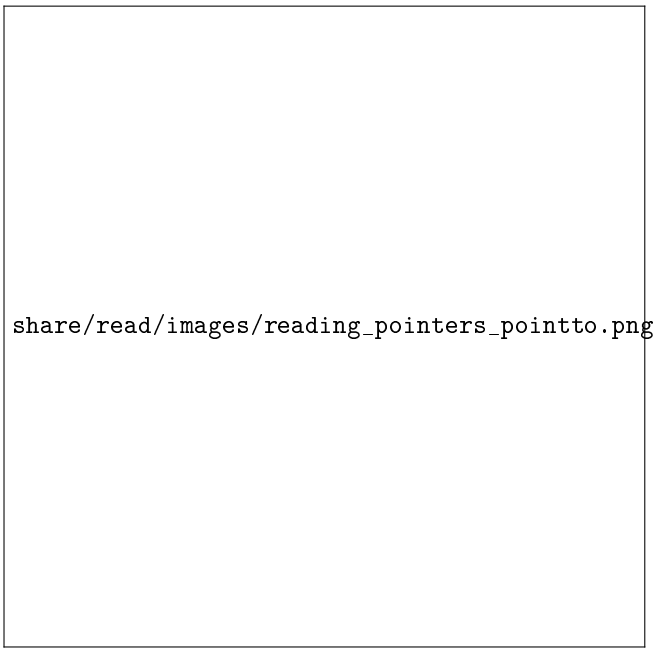
   Assignment forms

   - Assigning an address during declaration:
     `int* ptr = &somevariable;`
   - Assigning an address *after* declaration:
     `ptr = &somevariable;`

   After assigning an address to a pointer, if we `cout` the pointer it will display the memory address of the *pointed-to* variable - just like if we had used `cout` to display the *address-of* that variable.

   (a) **Example: A variable and a pointer**

   ```
   // Declare normal variable
   int someVariable = 100;

   // Declare pointer variable,
   // point to someVariable's address
   int* ptr = &someVariable;

   // Both show someVariable's address
   cout << &someVariable;
   cout << ptr << endl;
   ```



3. Dereferencing pointers to get values

   Once the pointer is pointing to the address of a variable, we can *access* that pointed-to variable's value by *dereferencing* our pointer. This gives us the ability to read the value stored at that memory address, or overwrite the value stored at that memory address. We **dereference** the pointer by prefixing the pointer's name with a `*` - again, another symbol being reused but in a different context.

   Dereference forms

   - Displaying pointed-to value:
     `cout << *ptr;`
   - Overwriting pointed-to value:
     `*ptr = 200;`
   - Overwriting pointed-to value with user input:
     `cin >> *ptr;`

   (a) **Example: A variable and a pointer**

   ```
   // Declare normal variable
   int someVariable = 100;

   // Declare pointer variable,
   // point to someVariable's address
   int* ptr = &someVariable;

   // Both show someVariable's address
   ```

```
cout << &someVariable;
cout << ptr << endl;

// Both show someVariable's value
cout << someVariable << endl;
cout << *ptr << endl;

// Update the variable's value via the poitner
*ptr = 50;
```

Safety with pointers!

Remember how variables in C++ store **garbage** in them initially? The same is true with pointers - it will store a garbage memory address. This can cause problems if we try to work with a pointer while it's storing garbage.

To play it safe, any pointer that is not currently in use should be initialized to `nullptr` (or `NULL` if you're going back to plain old C language :).

7.1.4 Pointer cheat sheet

Declare a pointer	`int* ptrInt;` `float *ptrFloat;`
Assign pointer to address	`ptrInt = &intVar;` `ptrFloat = &floatVar;`
Dereference a pointer	`cout << *ptrChar;` `*ptrInt = 100;`
Assign to nullptr	`float *ptrFloat{nullptr};` `ptrChar = nullptr;`

7.2 Lab: Pointers

7.2.1 Assignment information

- **Turning in your work:**
  1. In VS Code, build and run the graded program(s) in the **Terminal**. **Take a screenshot of the program running** and save it to your computer.
  2. In VS Code, make sure to COMMIT and SYNC your changes to the GitLab repository. (See: Using Git and VS Code).
  3. Go to your GitLab repository page, make sure it shows your latest COMMIT MESSAGE and your code is up-to-date. (Important!! I can't build and run your code if it's not on the server!!)
  4. In Canvas, go to the **Assignment** and click **Start Assignment**. **Upload the screenshot of your program running**. In the comments, let me know if you're done with all of the lab or just part of it and if you have any questions.
- **Don't utilize future topics!** - We will revisit the graded program here once we cover Exceptions later on. Please do not use exceptions for this assignment
- **Practice programs & graded programs:** Completing the graded program(s) is worth 90% of the lab grade. Completing the practice programs is worth an additional 10%.
- **Links:**
  - Using Git and VS Code
  - Program arguments
  - Debugging with gdb / lldb
  - Assignment direct link

7.2.2 Included files:

```
wk05_Pointers
 graded_program
    courses.cpp
 instructions.html
 instructions.org
 practice1_sizes
    sizes.cpp
 practice2_addresses
    addresses.cpp
 practice3_pointers
    pointers.cpp
 practice4_dereferencing
     dereference.cpp
```

7.2.3 Practice 1 - Type sizes

1. Instructions

   A `cout` statement is provided that displays the `sizeof( int )`. Add additional lines to display the size of float, double, bool, char, and string.
   ~

2. Reference

   You can use the `sizeof` function to see how much space, in bytes, a data type takes up.

   ```
   cout << sizeof( VARIABLE );
   cout << sizeof( DATATYPE );
   ```

   ~

3. Example output

   ```
   integer size:  4
   float size:    4
   double size:   8
   bool size:     1
   char size:     1
   string size:   32
   ```

7.2.4 Practice 2 - Variables' addresses

1. Instructions

   The address of variables `int1` and `bool1` are already being displayed with `cout` statements. Finish up this program by displaying the addresses of all int and bool variables.
   ~

2. Reference

   You can access the address of a declared variable by using the address-of operator `&`.

   ```
   cout << &VARIABLE;
   ```

   ~

3. Example output

   ```
   int1's address is: 0x7ffd2b5fa974
   int2's address is: 0x7ffd2b5fa978
   int3's address is: 0x7ffd2b5fa97c
   int4's address is: 0x7ffd2b5fa980
   ```

```
int5's address is: 0x7ffd2b5fa984

bool1's address is: 0x7ffd2b5fa96f
bool2's address is: 0x7ffd2b5fa970
bool3's address is: 0x7ffd2b5fa971
bool4's address is: 0x7ffd2b5fa972
bool5's address is: 0x7ffd2b5fa973
```

7.2.5 Practice 3 - Pointers

1. Instructions

   At the start of this program the addresses and values of `studentA`, `studentB`, and `studentC` are displayed.

   I have one example written where I set `ptrStudent` to point at `studentA`'s address, then it displays the location where `ptrStudent` is pointing.

   Follow along and do the same to point `ptrStudent` to `studentB` and `studentC` and display the updated address stored in `ptrStudent` each time.

   ~

2. Reference A pointer-variable is another type of variable... except its VALUE isn't an integer or character or float, it's a memory address!

   To declare a pointer, we use the datatype of the thing it's going to *point to*, plus an asterisk to mark it as a pointer, then give it a name. The * can be attached to the data type or the variable name or neither, but I prefer the first way here:

   ```
   int* integerPointer;
   float * floatPointer;
   string *stringPointer;
   ```

   To assign a pointer-variable the *address* of a different variable, we prefix the variable's name with the '&' address-of operator:

   ```
   integerPointer = &someInteger;
   ```

   ~

3. Example output

   ```
   studentA address: 0x61fef0, value: Luna
   studentB address: 0x61fed8, value: Kabe
   studentC address: 0x61fec0, value: Korra

   ptrStudent is now pointing to: 0
   ptrStudent is now pointing to address: 0x61fef0
   ptrStudent is now pointing to address: 0x61fed8
   ptrStudent is now pointing to address: 0x61fec0
   ```

7.2.6 Practice 4 - Dereferencing pointers

1. Instructions

   In this program the table of students is shown at the start. I have code that sets `ptrStudent` to point to `studentA`, display the address that `ptrStudent` is pointing to, the value at that address, and then ask the user to enter a new value for that student, which is done via the dereferenced pointer.

   Repeat the same for `studentB` and `studentC` beneath it.
   ~

2. Reference

   To assign a pointer-variable the *address* of a different variable, we prefix the variable's name with the '&' address-of operator:

   ```
   integerPointer = &someInteger;
   ```

   You can then **de-reference** a pointer to access the value at the address it's pointing to. You can display values, overwrite values, and more - indirectly, through the pointer.

   ```
   *integerPointer = 1000;
   cout << "someInteger is now " << *integerPointer << endl;
   ```

   ~

3. Example output

   ```
   ORIGINAL TABLE
   studentA address: 0x7fff050dde10, value: Luna
   studentB address: 0x7fff050dde30, value: Kabe
   studentC address: 0x7fff050dde50, value: Korra

   ptrStudent is pointing to address: 0

   ptrStudent is now pointing to address: 0x7fff050dde10
   CURRENT VALUE: Luna
   Enter a new name: Buddy

   ptrStudent is now pointing to address: 0x7fff050dde30
   CURRENT VALUE: Kabe
   Enter a new name: Daisy

   ptrStudent is now pointing to address: 0x7fff050dde50
   CURRENT VALUE: Korra
   Enter a new name: Freya


   UPDATED TABLE
   ```

```
studentA address: 0x7fff050dde10, value: Buddy
studentB address: 0x7fff050dde30, value: Daisy
studentC address: 0x7fff050dde50, value: Freya
```

7.2.7 Graded program

1. Reference

   Dereferencing a pointer and access a member: If your pointer is pointing to an object variable, you can dereference the pointer and access a member variable or function using the `->` operator:

   ```
   cout << ptrThing->name << endl;
   ```

   When pointers are not in use they should be set to `nullptr`, then you can use an if statement to tell whether that pointer is being used or not:

   ```
   if ( ptrThing->anotherPointer != nullptr )
   {
       // Safe to do the thing
   }
   ```

   If a pointer contains another pointer, you can point it to its internal ptr:

   ```
   myPtr = myPtr->next;
   ```

2. Instructions

   At the top of this file I have a `Course` struct, which contains a course code (e.g., CS 200) and pointers to other Courses - a previous and a next.

   Within `main()`, four Course variables have been created. Beneath the `// TODO: Set each course's 'ptrPrev' and 'ptrNext'`, set each of the following:

Course	ptrPrev	ptrNext
course1	nullptr	&course2
course2	&course1	&course3
course3	&course2	&course4
course4	&course3	nullptr

   Afterwards, I've created a `Course*` course pointer to point to the current class that we're looking at. It starts at the address of `&course1`. I've also created a while loop that will end whenever `ptrCurrent` becomes null.

   Within the while loop, I'm displaying the course code via the pointer: `ptrCurrent->code`.

   Afterwards, set `ptrCurrent` equal to its own next pointer: `ptrCurrent->ptrNext`. We use this to traverse forward between the pointers.

3. Example output

```
-- POINTERS TO COURSES --
Current course: CS 134
Current course: CS 200
Current course: CS 235
Current course: CS 250

-- GOODBYE! --
```

8 Week 7: Arrays and Vectors

8.1 Intro: Arrays and storing lists of data

cs200/read/https:/gitlab.com/moosadee/courses/-/raw/main/images/topics/arrays.png?ref_type=heads

8.1.1 Array basics - what are arrays?

In C++, arrays are a built-in way to store a series of similar data all under one name.

Before now, if we wanted to store a list of students in a class (or similar kind of data), we would have to declare a bunch of separate variables and write the same code over and over to manage it:

Example: Creating separate variables for three students and creating 3 different prompts and inputs for each variable

```
string student1, student2, student3;

cout << "Enter student 1 name: ";
cin >> student1;

cout << "Enter student 2 name: ";
cin >> student2;

cout << "Enter student 3 name: ";
cin >> student3;
```

This would quickly become unmanageable if you were writing a program with tens, hundreds, or thousands of students stored in it. Instead, we can make use of **arrays** to store a series of related data together.

Example: Creating an array and asking the user to enter a value for every element of the array

```
string students[100];
for ( int i = 0; i < 100; i++ )
{
  cout << "Enter student " << (i+1) << " name: ";
  cin >> student[i];
}
```

Arrays allow us to operate on the same name (e.g., `student`), but addressing different **elements** of the array with an **index** number. This way, we can write code to act on the data *once*, just modifying that index to work with different pieces of data.

Example: student array of size 4:

Element	Rai	Anuj	Rebekah	Rose
Index	0	1	2	3

Each item in the array has a corresponding **index** marking its position in the list, with 0 being the first value. If an array is of size n, then the valid indices are 0 through $n - 1$.

An **element** is the information stored at that index position, which is essentially a single variable in an array of variables.

8.1.2 Declaring arrays

In C++, we declare an array similarly to how we declare a variable, except that we need to specify an array **size** during declaration:

Example: Declaring an array of strings of size 100

```
// An array of size 100
string students[100];
```

Or, if we already have data to put into it, we can initialize it with an **initializer list**. Then the array will be sized at however many items you give it.

Example: Declaring an array of strings, initializing the data with an initializer list

```
// An array of size 4
string students[] = {"Rai", "Anuj", "Rebekah", "Rose"};
```

1. Size must be known at compile-time

   Traditional C-style arrays must have its size be defined during its declaration and the array **cannot** be resized during the program's execution. Because the array size can't change, and because we may need to know the size of the array throughout the program, we will generally use a **named constant** to store the size of the array.

   Example: Using a named constant integer and using it to define how many elements are in a new array

   ```
   const int MAX_STUDENTS = 100;
   string students[MAX_STUDENTS];
   ```

2. A variable to store the # of items in the array

   Because an array's size cannot be changed after its declaration, it often becomes necessary to *overshoot* the amount of spaces we need in the array so that we always have enough room for our data. Perhaps a school's classrooms range from 20 to 70 seats, so we would want to declare the array of the biggest size so that we don't run out of space. Because of this, not all spaces in the array may be taken up at any given time.

C++ doesn't have a function to directly get the amount of elements in an array, so generally when declaring an array we need to also have an associated variable to track how many items we've stored in the array.

Example: Declaring an array, storing a value for the element at index 0, adding 1 to the item count

```
const int MAX_STUDENTS = 100;   // total array size
int studentCount = 0;           // how many elements
string students[MAX_STUDENTS];  // array declaration

students[0] = "Rai";            // setting up first student
studentCount++;                 // adding 1 to student count
```

3. Array data types

   Arrays can be declared with any data type. To the computer, we are just declaring n amount of some variable, and it takes care of putting all the `n` variables back-to-back in memory.

4. Valid indices

   As stated earlier, if an array is of size n, then its valid indices are 0 to `n-1`. In most computer languages, arrays and lists start at index 0 and go up from there, meaning to count for items we have "0, 1, 2, 3".

   Warning!

   A common source of program crashes is **going outside the bounds of an array**! This happens if you try to access an invalid index, such as `student[-1]` or `student[100]` (for an array of size 100; valid indices are 0-99).

8.1.3 Accessing array elements via index

After we've declared an array, we can treat its **elements** like normal variables, storing data within it and accessing that data. Each one can be accessed via its **index** (position in the array) and the **subscript operator**: `[]`.

In math, you might be used to these subscripts:

$$a_n = a_{n-1} + 2$$

But with C++, subscripts look like this:

```
a[n] = a[n-1] + 2
```

So, declaring an array and accessing its values can look like this:

Example: Declaring an array and storing values for each element, displaying the value of the element at index 0

```
string courses[4];
courses[0] = "CS 134";
courses[1] = "CS 200";
courses[2] = "CS 210";
courses[3] = "ASL 120";

cout << "First course: " << courses[0] << endl;
```

Or, with modern C++ (from 2011 and later) you can initialize an array with an initializer list, `{}`:

Example: Using an initializer list to initialize an array's elements

```
string courses[4] = {
  "CS 134",
  "CS 200",
  "CS 210",
  "ASL 120"
};

cout << "First course: " << courses[0] << endl;
```

Design!

It's common for **array names to be plural** - it is a structure that stores *multiple items* - multiple courses, multiple products, multiple grades, etc. So, to be clear, it is best to give your arrays *plural* names: `courses`, `products`, `grades`, instead of singular.

1. Accessing indices with variables

   Since the array **index** is always an integer, we could use a variable to determine which item in an array to modify - such as asking the user which item they want to edit.

   Example: Asking the user to enter an index and displaying the element at that index

   ```
   cout << "Edit which item? (0-9): ";
   cin >> itemIndex;
   cout << "Enter price for item: ";
   cin >> prices[ itemIndex ];
   ```

8.1.4 Using for loops with arrays

Using for loops is the most common way to iterate over all the data in an array, setting data or accessing data. Have an array of size `n`? We want to iterate from `i` $= 0$ to $n - 1$, going up by 1 each time.

1. Design pattern: Asking the user to enter all elements

   We can iterate over all the items in an array and have the user enter information for each element by using a loop. Since the first index is 0, we sometimes just add +1 to the index for the user's benefit, since people generally aren't used to lists starting at 0.

   Example: Iterating over an array, asking the user to enter each element's value

```
const int TOTAL_ITEMS = 10;
float prices[TOTAL_ITEMS];

for ( int i = 0; i < TOTAL_ITEMS; i++ )
{
  cout << "Enter price for item " << (i+1) << ": ";
  cin >> prices[i];
}
```

2. Code pattern: Displaying all items in an array

   We can display all the elements of an array by using a loop as well, though we usually *don't* want to show *all elements* of the array - usually just the items we know we're storing data in. Recall that we usually will have an extra integer variable to count how many items have actually been stored in the array, which is different from the total array size.

```
const int MAX_STUDENTS = 100;
int studentCount = 0;
string students[MAX_STUDENTS];

// (...Let's say 20 students were added here...)

// Iterate from index 0 to 19, since we have stored
// 20 students so far.
for ( int i = 0; i < studentCount; i++ )
{
  cout << "Student " << (i+1)
    << " is " << students[i] << endl;
}
```

   You can also use a **range-based for loop** in versions of C++ from 2011 or later:

```
for ( auto & student : students )
{
  cout << student << endl;
}
```

8.1.5 Parallel arrays

Let's say we're writing a simple restaurant program where we need a list of dishes *and* their prices together. Later on, we will write our own data type using **structs and classes** to keep these items together. But for now, we would implement this relationship by keeping track of two separate arrays with the data we need.

Example: Using 3 parallel arrays to store a dish's name, price, and if its vegetarian

```
const int MAX_DISHES = 20;
int dishCount = 0;

// Information about a dish
string  dishNames[MAX_DISHES];
float   dishPrices[MAX_DISHES];
bool dishVegetarian[MAX_DISHES];

// ... Let's say we created some dishes here...

// Display the menu
cout << "What would you like to order?" << endl;
for ( int i = 0; i < dishCount; i++ )
{
  cout << "Dish #" << i << ":" << endl;
  cout << "* Name:   " << dishNames[i] << endl;
  cout << "* Price: $" << dishPrices[i] << endl;

  if ( dishVegetarian[i] ) {
    cout << "* Is vegetarian" << endl;
  }

  cout << endl;
}

// Get the index of dish they want
int whichDish;
cout << "Selection: ";
cin >> whichDish;
```

8.1.6 Arrays with Functions: Arrays as arguments

they could potentially take up a lot of memory. Because of this, passing an array as a pass-by-value parameter would be inefficient - remember that pass-by-value means that the parameter is *copied* from the caller argument.

C++ automatically passes arrays around as **pass-by-reference** instead. You don't have to inclue the `&` symbol in your parameter list for an array, it just happens! But - keep in mind that any problems with pass-by-reference

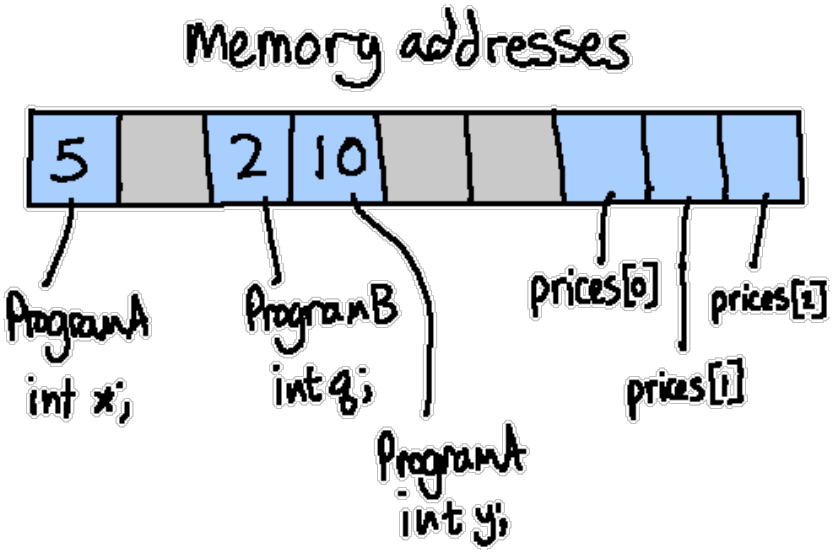
also apply to arrays. If you want to pass an array to a function but you **don't want the data changed**, then you would need to mark that array parameter as **const**.

Example: Creating a function to iterate over all element of an array

```
void DisplayAllItems( const string arr[], int size )
{
  for ( int i = 0; i < size; i++ )
  {
    cout << arr[i] << endl;
  }
}
```

When using an array as a parameter, you don't have to hard-code a size to it. You can leave the square brackets empty (but you DO need square brackets to show that it's an array!) and then pass any array (with matching data type) to the function. However, you will also want to have an int parameter to pass the size of the array as well.

8.1.7 Arrays and memory



When we declare any variable it is assigned a memory address as it is stored somewhere in memory. You can think of this as it being stored in RAM (though the operating system abstracts this a bit more). When a program starts and variables are declared, space is taken up "in RAM" and then freed when the program closes or when the variable is destroyed. This way, other programs running in the operating system can utilize that space.

A memory address is a numeric value. Computers work on binary, but we often write memory addresses in hexadecimal to shorten the length of the addresses.

An example memory address is: 0x7ffd1f7ba26c.

Example: Here's a simple program that creates two different variables, assigns values, and outputs the values and their memory addresses:

```
int main()
{
    int num = 10;
    string name = "Hello!";

    cout << "The variable 'num' has a value of " << num
         << "\n at the address " << &num << endl << endl;
    cout << "The variable 'name' has a value of " << name
         << "\n at the address " << &name << endl;

    return 0;
}
```

When it is run, it will give output like this:

```
The variable 'num' has a value of 10
at the address 0x7fff28b210bc

The variable 'name' has a value of "Hello!"
at the address 0x7fff28b210c0
```

Because we need to know the array's size at compile time (while writing the program, vs. run time, while the program is running), we can set the array size with an integer literal:

```
string studentNames[22];
```

Or a named constant:

```
const int TOTAL_SEATS = 22;
string studentNames[TOTAL_SEATS];
```

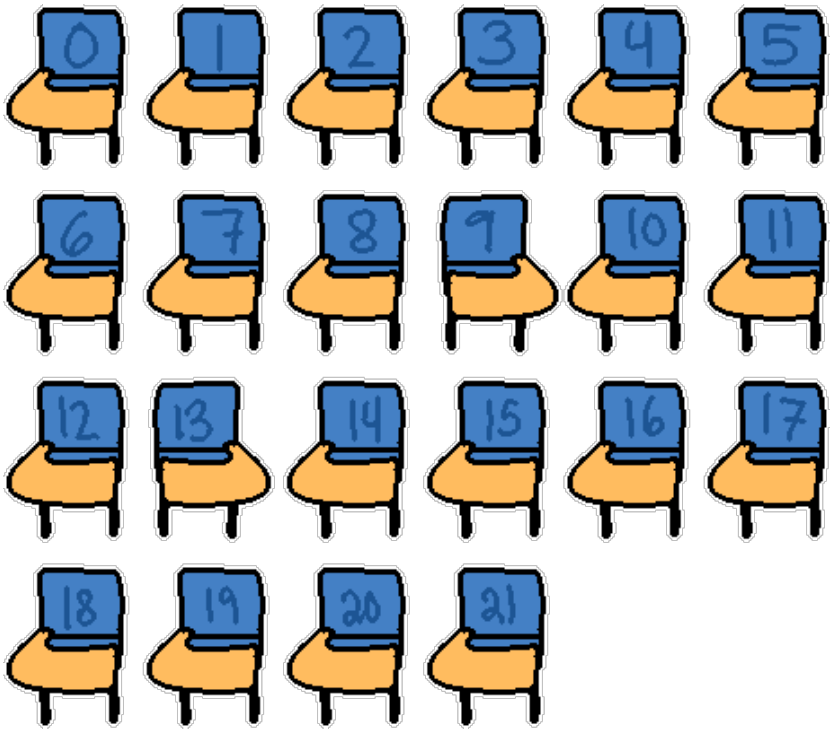
But not a variable:

```
int totalSeats;
cout << "Enter total seats: ";
cin >> totalSeats;
string studentNames[totalSeats];
```

Though down the road we will be able to create a "Dynamic Variable", which we can set its size during runtime with a variable - but not right now with these standard ol' arrays.

Now, some compilers (especially old ones) may let you actually declare an array with a variable size. HOWEVER, not every compiler allows this so it should not be used.

8.1.8 Array size vs. item count



Let's say we have our student array and a maximum amount of 22 seats:

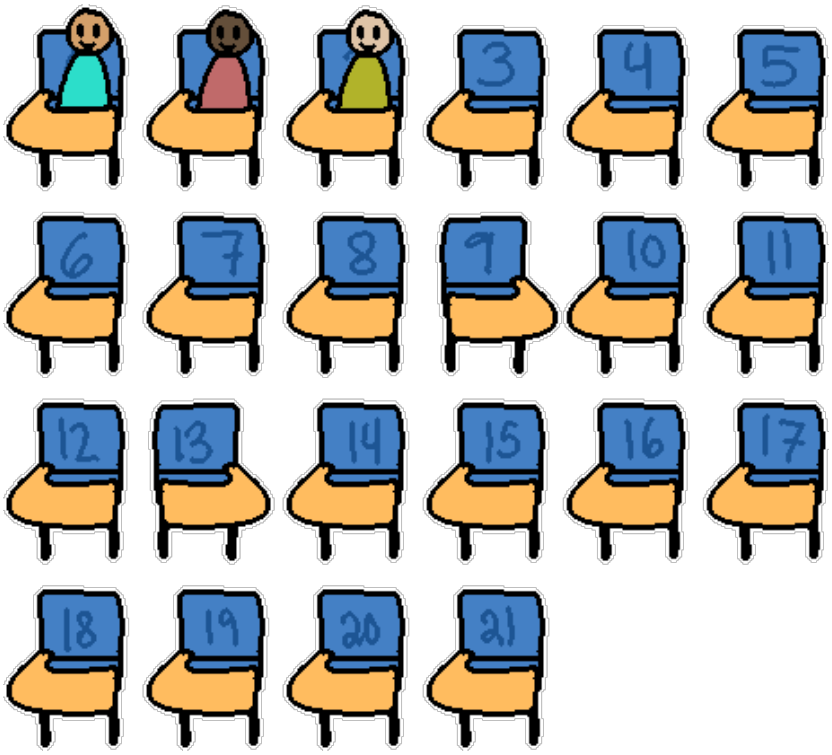
```
const int TOTAL_SEATS = 22;
string students[TOTAL_SEATS];
```

One design consideration to think about here is that, if we iterate from i = 0 to `TOTAL_SEATS`, we're going to iterate over seats that have students AND seats that are empty. What if we were writing a program and JUST wanted to display a list of students enrolled in the course, and not waste paper space or screen space drawing blank links for empty seats?

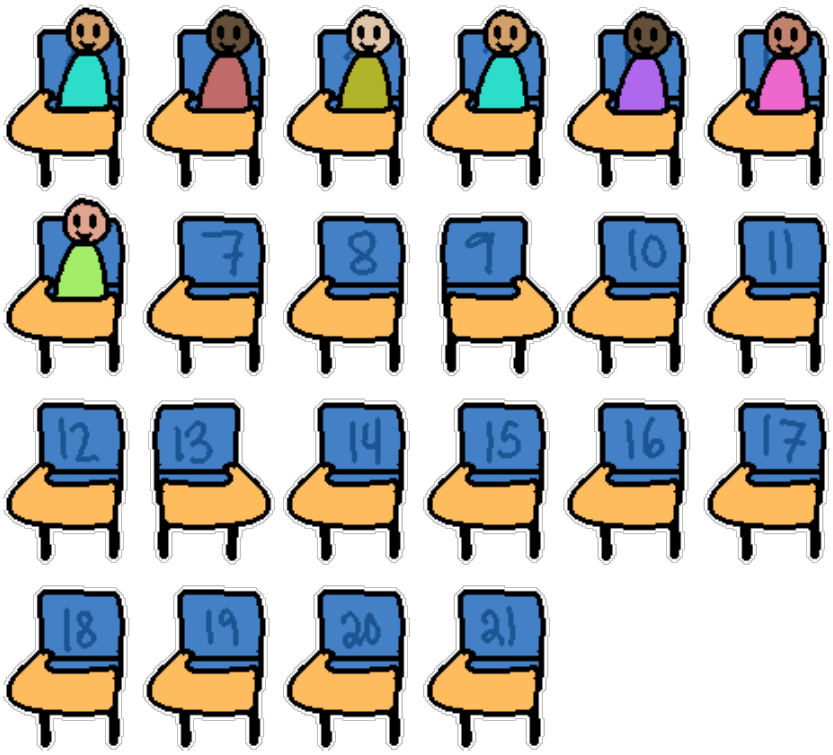
In this case we usually need to keep a variable to track how many items we're actually storing in the array. For example:

```
int studentCount;
```

so the `TOTAL_SEATS` would be the size of the array, but the studentCount might be less than or equal to that number. Every time we added a student we would add +1 to the studentCount. That way, we could use studentCount in our for loop to display just active students, none of the empty seats.



... The `TOTAL_SEATS` remain the same, even as the `studentCount` grows from more students taking up space in available seats...



8.1.9 Array management functionality

Since arrays require quite a bit of management to work with, you could implement some basic functions to do this management, instead of having to re-write the same code over and over. For example...

1. Clear array

   Sets all elements of this string array to an empty string and resets the `elementCount` to 0 afterwards.

   Example: Iterating over an array and setting each element's value to 0

```
void Clear( string arr[], int & elementCount )
{
  for ( int i = 0; i < elementCount; i++ )
  {
    arr[i] = "";
  }
  elementCount = 0;
}
```

2. Display all elements

   Shows all the elements of an array.

   Example: Iterating over an array and display each element's index and value

```
void Display( const string arr[], int elementCount )
{
  for ( int i = 0; i < elementCount; i++ )
  {
    cout << i << "\t" << arr[i] << endl;
  }
}
```

3. Add new element to array

   Often in our programs we will only set up one item at a time (perhaps when selected from a menu). This means we need to get the data for the array and figure out where in the array it will go.

   Example: Asking the user t o enter a new value, storing it in the array at the first available empty space

```
void AddItem( string arr[], int & elementCount )
{
  cout << "Enter new element: ";
  cin >> arr[ elementCount ];
  elementCount++;
}
```

   For an array, `elementCount` starts at 0 when the array is empty. This also happens to be the **index** where we will insert our first element - at 0.

Element				
Index	0	1	2	3

   Once we add our first element, our `elementCount` will be 1, and the next index to insert data at will also be 1.

Element	Cats			
Index	0	1	2	3

   Then, `elementCount` will be 2, and the next index to insert at will be 2.

Element	Cats	Dogs		
Index	0	1	2	3

   Because of this, the `elementCount` variable both tells us how many items have been stored in the array *and* what is the next index to store new information at.

8.1.10 Multidimensional arrays

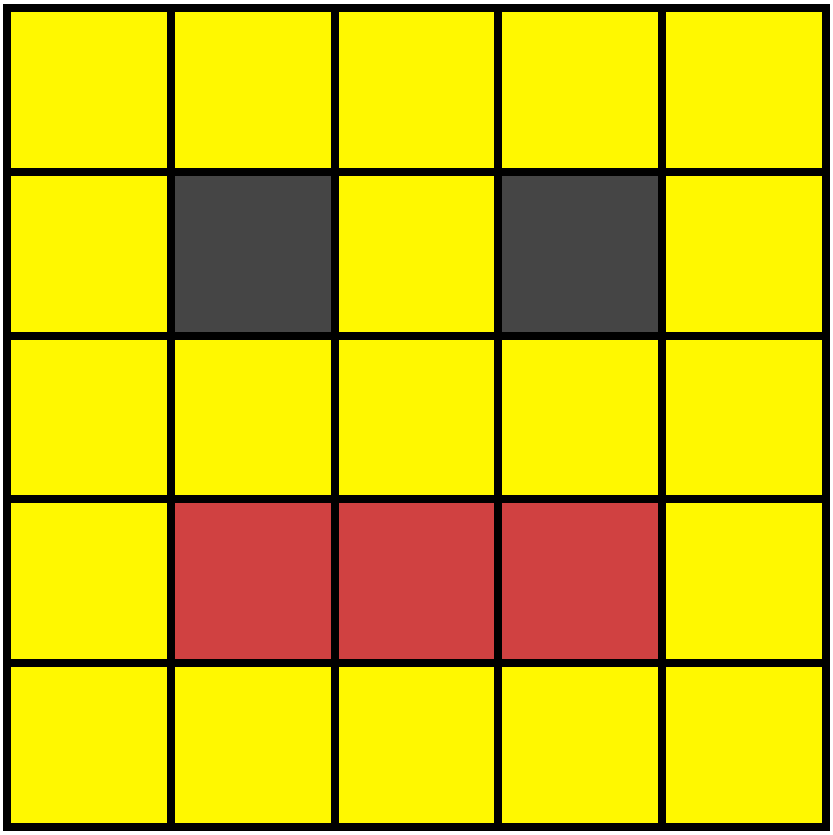
We can declare an array with two dimensions. A 1 dimensional array we can think of like this:

CS134 CS200 CS210 CS211 CS235

Whereas a 2 dimensional array we can think of like a spreadsheet with multiple rows and columns:

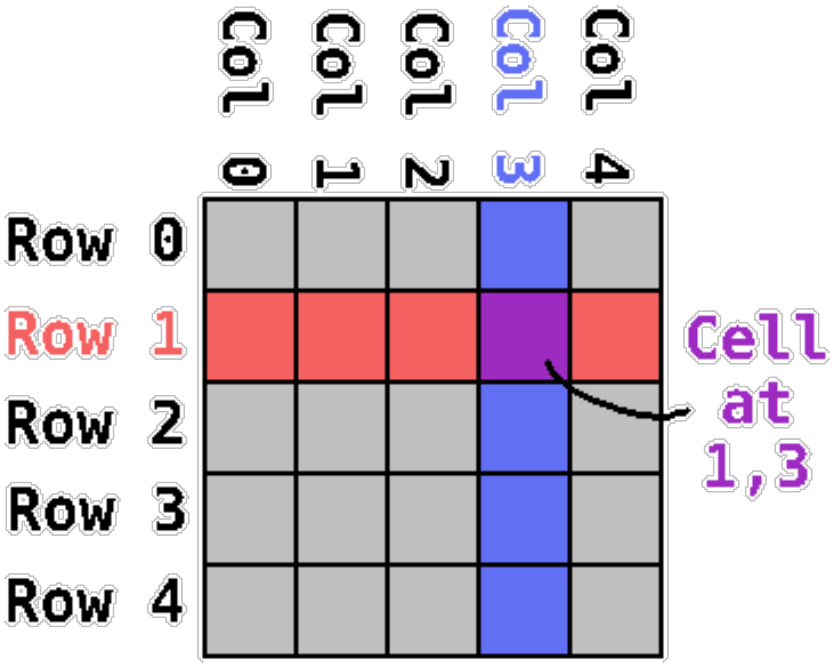
CS134	Spring 2022	PW	A
CS200	Summer 2022	RS	A
CS235	Fall 0222	RS	A

Or even as a 2D display on a screen with an (x, y) position for pixels:



Generally when talking about rows and columns, rows comes first and columns come second.

Row is a horizontal grouping of information, a column is a vertical grouping of information. On a spreadsheet, a "cell" is a single item denoted by a row and a column.



Often, the **column** relates to a specific field of data (such as a student's name, age, gpa, etc.) - Think of a struct and its member variables.

Then, each **row** corresponds to one record. So if we had an array of students, each row would be student 0, 1, 2, 3, etc. and then each column would be name, age, gpa, etc.

Example: Example of declaring a 2D and a 3D array

```
string spreadsheet[32][32]; // rows and columns
int vertices[4][4][4]; // x, y, z
```

Let's say we wanted to turn a day planner into a program. Perhaps we originally stored the day plan like this:

Day	Time	Task
Monday	8:00	Work meeting with Bob
Monday	13:00	Project review
Tuesday	10:00	Customer meeting
Tuesday	12:00	Crying in car
Wednesady	14:00	Sprint Retrospective

We can convert this into a 2D array of strings (the "task"), with one dimension being for "day of the week" and the other dimension being for "hour of the day"...

Example: Declaring a 2D array of strings

```
int DAYS_OF_WEEK = 7;
int HOURS_IN_DAY = 24;
string todo[ DAYS_OF_WEEK ][ HOURS_IN_DAY ];
```

We could ask the user what day they want to set a task for, and if they type "Sunday" that could translate to 0, and "Saturday" could translate to 6 (or however you want to organize your week)...

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
0	1	2	3	4	5	6

We could ask them next for what hour the task is at, and that can map to the second index: "0" for midnight, "8" for 8 am, "13" for 1 pm, and so on to 23.

With this information, we can get the user's todo task and store it at the appropriate indices:

Example: Asking the user to enter a day and hour and using those as indices

```
int day, hour;

cout << "Enter the day:" << endl;
cout << "0. Sunday    \n"
     << "1. Monday    \n"
     << "2. Tuesday   \n"
     << "3. Wednesday \n"
     << "4. Thursday  \n"
     << "5. Friday    \n"
     << "6. Saturday  \n";

cout << "Day: ";
cin >> day;

cout << "Enter the hour (0 to 23): ";
cin >> hour;

cout << "Enter your task: ";
cin >> todo[ day ][ hour ];
```

With the user's week planned out, you could then display it back to the user by using a **nested for loop**: One loop for the day, one loop for the hour.

Example: Nested for loop to display all elements of the 2D array

```
cout << "YOUR SCHEDULE" << endl;

for ( int day = 0; day < DAYS_OF_WEEK; day++ )
{
  for ( int hour = 0; hour < HOURS_IN_DAY; hour++ )
  {
```

```
    cout << day << ", "
         << hour << ": "
         << todo[ day ][ hour ] << endl;
  }
}
```

Though you'd probably want to do some extra formatting; such as determining that if the day is 0, then write "Sunday" instead of the number "0".

1. Displaying all the elements of a 2D array

   To display the contents of a 2D array, it can be useful to have two for loops, one nested in the other. The outer for loop will only increment once the inner one finishes its cycle. I usually put the **rows** as the outer loop and the **columns** as the inner-loop.

```
for ( int row = 0; row < ROWS; row++ )
{
    cout << "Row: " << row << endl;
    for ( int col = 0; col < COLUMNS; col++ )
    {
        cout << "\t Col: " << col
             << ": " << arr[row][col]
             << endl;
    }
}
```

8.1.11 Review questions:

1. Do all items in an array need to be the same data type in C++?
2. What is an *element* of an array?
3. What is an *index*?
4. The subscript operator is...
5. What code would you write to display the item at position 0 in an array?
6. What code would you write to display the item at position 2 in an array?
7. Given an array of size n, the valid indices of the array are...
8. How do you *iterate* over all elements of an array?

8.2 Intro: STL Array and STL Vector

8.2.1 C++ Standard Template Library: Arrays

C++ provides various handy functionality in it's standard library (abbreviated, unfortunately, as "std"). This includes a smarter array, as well as another structure we'll look at in a moment.

The documentation for the std::array (aka array object) is available here: https://www.cplusplus.com/reference/array/array/

Just like our standard array, we still need to know its size at compile time, but it keeps track of its own size so we don't need to declare a named constant like const int `ARRAY_SIZE = 10;`

If we put `#include <array>` at the top of our file, we can use an `array` object instead of a traditional array to store data. They are basically interchangable, ecxept the `array` object gives us access to the `.size()` function, making our job slightly easier. However, the `array` still can't be resized.

- **Creating an STL array:**

```
array<string, 5> products;
```

- **Setting elements of the array:**

```
products[0] = "Pencil";
products[1] = "Eraser";
products[2] = "Pencil case";
products[3] = "Pencil sharpener";
products[4] = "Ruler";
```

- **Iterating over the array:**

```
for ( unsigned int i = 0; i < products.size(); i++ )
{
    cout << i << ". "; // Display index
    cout << products[i] << endl; // Display element at that index
}
```

8.2.2 C++ Standard Template Library: Vectors

Documentation: https://cplusplus.com/reference/vector/vector/

The C++ Standard Template Library contains a special structure called a **vector**. A vector is a **class** (something we'll learn about later) and it is implemented on top of a **dynamic array** (which we will learn about later with pointers). Basically, it's a resizable array and it has functionality to make managing data a bit more manageable.

Generally, in Computer Science curriculum, we teach you how to build **data structures** (structures that store data) like vectors, lists, and other items because it's important to know how they work (thus why we're covering arrays), but for your own projects and in the real world, you would probably use a vector over an array.

- **Declaring a vector:**

  Vectors are a type of **templated** object, meaning it can store any data type - you just have to specify what kind of type the vector stores. The format of a vector declaration looks like this:

```
// Declaring vectors
vector<string> students;
vector<float>prices;
```

- **Adding data:**

  You can add data to a vector by using its `push_back` function, passing the data to add as the argument:

```
vector<string> students;
students.push_back( "Rai" );
```

- **Getting the size of the vector:**

  The `size` function will return the amount of elements currently stored in the vector.

```
cout << "There are " << students.size() << " students" << endl;
```

- **Clearing the vector:**

  You can erase all the data in a vector with the `clear` function:

```
students.clear();
```

- **Accessing elements by index:**

  Accessing an element at some index looks just like it does with an array:

```
cout << "Student: " << students[0] << endl;
```

- **Iterating over a vector:**

  You can use a for loop to iterate over all the elements of a vector, similar to an array:

```
cout << "Students:" << endl;

for ( int i = 0; i < students.size(); i++ )
{
    cout << i << "\t" << students[i] << endl;
}
```

You can also use C++11 style **range-based for loop** if you don't need the index. It allows you to iterate over all the elements of `students`, using an alias of `student` for each element.

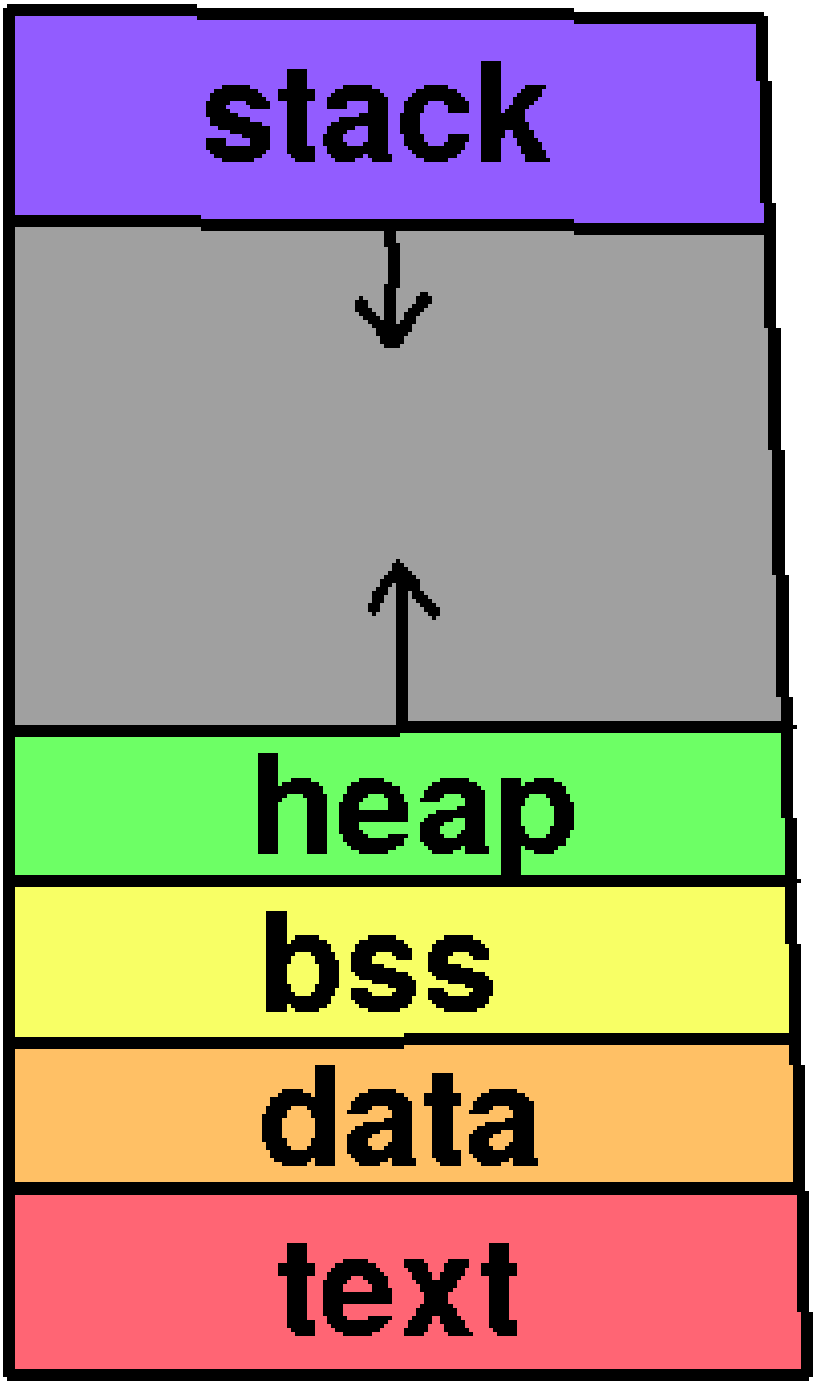
```
for ( auto & student : students )
{
    cout << student << endl;
}
```

8.2.3 Review questions:

1. How do you declare an STL `array` object?
2. How do you declare an STL `vector` object?
3. How do you add new items to a `vector` object?

8.3 Intro: Dynamic arrays

8.3.1 Program memory



Programs have a memory space where different types of data is stored.

- **Text** is program instructions.

- **Data** is where global and static variables are stored.
- **Heap** space is where dynamically allocated variables/arrays are stored.
- **Stack** space is where local variables, parameter variables, are stored.

There is a limit to stack space. When you have something like a logic error in a recursive function and it keeps allocating space for variables eventually you'll get a Stack Overflow, running out of stack space.

Dynamically allocated variables and arrays are stored on the Heap.

8.3.2 Dynamic array

We can use Dynamic Arrays to allocate space for an array at run-time, without having to know or hard-code the array size in our code. To do this, we need to allocate memory on the heap via a **pointer**.

In C++ we use the **new** and **delete** keywords to allocate and deallocate memory. Because we have to manually manage this memory, it's important to remember to **delete** anything that has been allocated via **new**.

Creating a dynamic array

We don't have to know the size of the array at compile-time, so we can do things

```
int size;
cout << "Enter size: ";
cin >> size;
string* products = new string[size];
```

like ask the user to enter a size, or otherwise base its size off a variable.

Setting elements of the array

We can access elements of the dynamic array with the subscript operator, as before. However, we won't know the size of the array unless we use the `size` variable, so it'd be better to use a for loop to assign values instead of this example:

```
products[0] = "Pencil";
products[1] = "Eraser";
products[2] = "Pencil case";
products[3] = "Pencil sharpener";
products[4] = "Ruler";
```

Iterating over the array

Accessing elements of the array and iterating over the array is done the same way as with a traditional array.

```
for ( unsigned int i = 0; i < size; i++ )
{
  cout << i << ". "; // Display index
  cout << products[i] << endl; // Display element at that index
}
```

Freeing the memory when done

Before your pointer loses scope we need to make sure to free the space that we allocated:

```
delete [] products;
```

1. "Resizing" the array

   When memory is allocated for an array we must know the entire size at once because all of the array's elements are *contiguous in memory*. Because of this, we don't technically "resize" a dynamic array, instead we allocate more space *elseware* and copy the data over to the new array. Here are the steps:

   One: Allocate space for a new, bigger array

   ```
   string* newArray = new string[ size + 10 ];
   ```

   Two: Copy the data from the old array to the new array

   ```
   for ( unsigned int i = 0; i < size; i++ )
   {
     newArray[i] = products[i];
   }
   ```

   Three: Free the space at the old address

   ```
   delete [] products;
   ```

   Four: Update the main array pointer to the new address

   ```
   products = newArray; // Point to same address
   ```

   Five: Update your size variable

   ```
   size = size + 10;
   ```

8.3.3 Review questions:

1. How do you allocate memory for an array via a pointer?
2. How do you deallocate memory for a dynamic array?
3. What are the steps to "resize" a dynamic array?

8.4 Lab: Arrays and vectors

8.4.1 Assignment information

- **Turning in your work:**
  1. In VS Code, build and run the graded program(s) in the **Terminal**. **Take a screenshot of the program running** and save it to your computer.
  2. In VS Code, make sure to COMMIT and SYNC your changes to the GitLab repository. (See: Using Git and VS Code).
  3. Go to your GitLab repository page, make sure it shows your latest COMMIT MESSAGE and your code is up-to-date. (Important!! I can't build and run your code if it's not on the server!!)
  4. In Canvas, go to the **Assignment** and click **Start Assignment**. **Upload the screenshot of your program running**. In the comments, let me know if you're done with all of the lab or just part of it and if you have any questions.
- **Don't utilize future topics!** - We will revisit the graded program here once we cover Exceptions later on. Please do not use exceptions for this assignment
- **Practice programs & graded programs:** Completing the graded program(s) is worth 90% of the lab grade. Completing the practice programs is worth an additional 10%.
- **Links:**
  - Using Git and VS Code
  - Program arguments
  - Debugging with gdb / lldb
  - Assignment direct link

8.4.2 Included files:

```
wk07_ArraysVectors
 example_stlarray.cpp
 graded_program
    gpa.cpp
 instructions.org
 practice1_oldArray
    oldarray.cpp
 practice2_dynamicArray
    ptrarray.cpp
 practice3_vector
    stlvector.cpp
 practice4_variableindex
    shop.cpp
```

```
practice5_2darray
   draw.cpp
   image1.txt
   image2.txt
practice6_structureArray
    employees.cpp
```

8.4.3 Practice programs

1. Practice 1 - Old array

   (a) Instructions

      i. Near the top of 'main()', declare an array of strings. Its size should be 'ARRAY$_{\text{SIZE}}$'.

      ii. Within the for loop after "Getting input:", the message "Enter class #" is displayed. Afterwards, get the user's input from the keyboard and store it in your array, at position '[i]'.

      iii. After the text "Resulting array:", write a new for loop like the previous one. Within the loop, display each index 'i' and each element at that position, 'ARRAYNAME[i]'.

   ~

   (b) Reference

   C-style array declaration:

   ```
   TYPE ARRAYNAME[SIZE];
   ```

   Accessing an element at **index** i:

   ```
   cout << ARRAYNAME[i];
   ```

   Notes:

   - "Element of the array at position i" means use ARRAYNAME[i] to access a specific element.
   - `size_t` is short for `unsigned int`. Sizes of arrays cannot be negative, so using `unsigned` means the integer can be 0 to some MAX INT value.

   ~

   (c) Example output

   ```
   MY CLASSES v1

   Getting input:
   Enter class #0: CS 134
   Enter class #1: CS 200
   Enter class #2: CS 235
   Enter class #3: CS 250
   Enter class #4: ASL 120
   ```

```
Resulting array:
0 = CS 134
1 = CS 200
2 = CS 235
3 = CS 250
4 = ASL 120
```

---

2. Practice 2 - Dynamic array

   (a) Instructions

      i. After the user enters `total_classes`, create a dynamic array of strings. Use `total_classes` as the size.

      ii. After the "Getting input:" text, write a for loop that goes from `i=0` to the size of the array `total_classes`, incrementing by 1 each time. Within the loop, do the following: a. Display "Enter class #" and the value of `i`. b. Get the user's input from the keyboard and store it in the array element at position `i`.

      iii. After the "Resulting array" text, within this second for loop, display the value of `i` (the index) and the element's value at that position.

      iv. Before `return 0;`, make sure to free the array's memory.

   ~

   (b) Reference

   Create a dynamic array:

   ```
   TYPE* ARRAYNAME = new TYPE[SIZE];
   ```

   Free array's memory:

   ```
   delete [] ARRAYNAME;
   ```

   Iterate through an array, given some SIZE:

   ```
   for ( size_t i = 0; i < SIZE; i++ )
   {
       // index is i
       // element is ARRAYNAME[i]
   }
   ```

   ~

   (c) Example output

   ```
   MY CLASSES v2

   How many classes do you have? 3

   Getting input:

   Enter class #0: CS 210
   ```

```
Enter class #1: CS 200

Enter class #2: ASL 120

Resulting array:
0 = CS 210
1 = CS 200
2 = ASL 120
```

---

3. Practice 3 - Vector

   (a) Instructions

      i. Near the top of `main()`, declare a vector of strings named `my_classes`.
      ii. Within the do/while loop, after the getline statement, push the `new_class` variable into the `my_classes` vector.
      iii. After the "Resulting array:" text, create a for loop that iterates over all the elements of the `my_classes` vector. Within the for loop, display the index and the value of that element.

   ~

   (b) Reference

   Declaring a vector:

   ```
   vector<TYPE> VECTORNAME;
   ```

   Accessing a vector's size:

   ```
   VECTORNAME.size()
   ```

   Pushing a new item into the vector:

   ```
   VECTORNAME.push_back( DATA );
   ```

   ~

   (c) Example output

   ```
   MY CLASSES v3

   Enter class #0: CS 134
   Enter another? (Y/N): y

   Enter class #1: CS 210
   Enter another? (Y/N): y

   Enter class #2: ASL 120
   Enter another? (Y/N): n


   Resulting array:
   0 = CS 134
   1 = CS 210
   2 = ASL 120
   ```

---

4. Practice 4 - Variable index

   (a) Instructions

      i. Under the "Product:" text, display the `name` of the `product_list` element at index `index`.

      ii. Under the "Cost: $" text, display the `price` of the `product_list` element at index `index`.

   ~

   (b) Reference

   Accessing an array element's member variable:

   ```
   ARRAYNAME[INDEX].VARIABLE
   ```

   Displaying an array element's member variable:

   ```
   cout << ARRAYNAME[INDEX].VARIABLE;
   ```

   ~

   (c) Example output

   ```
   TACO PLACE

   MENU
   0. Bean Burrito ($1.99)
   1. Crunchy Taco ($1.79)
   2. Baja Blast ($1.29)

   What do you want to eat? 1

   You chose:

   Product: Crunchy Taco
   Cost: $1.79
   ```

---

5. Practice 5 - 2D arrays

   (a) Instructions

   After the `WIDTH` and `HEIGHT` named constants are declared, declare an array of `char` items using the `HEIGHT` and the `WIDTH`. (Rows, the height, should go first, then width, the columns, should go second.)

   After the image is loaded into the array, write a nested for loop pair to iterate over all of the characters and draw each one.

      i. **Outer loop:** From y = 0 to `HEIGHT`...

         A. **Inner loop:** From x = 0 to `WIDTH`...

         B. **Inside:** Output `image[y][x]`, no endl.

         C. **After inner loop, before outer loop ends:** cout an endl.

~

(b) Reference
Declaring a 2D array:

```
TYPE ARRNAME[SIZE1][SIZE2];
```

~

(c) Example output
Image 1:

```
./draw.exe image1.txt

-- IMAGE DRAWER --
- - O - -
- / | \ -
- - | - -
- | - | -
- n - n -

-- GOODBYE! --
```

Image 2:

```
./draw.exe image2.txt

-- IMAGE DRAWER --
. _ . _ .
/ . v . \
| . . . |
. \ . / .
. . v . .

-- GOODBYE! --
```

---

6. Practice 6 - Array of structures

(a) Instructions
An `Employee` struct has already been declared and a vector of `employees` has been created. Complete the program by writing a for loop to display all of the employee information in a table.
~

(b) Reference

- Accessing a vector's size: `VECTOR.size()`
- Accessing a specific index from a vector: `VECTOR[INDEX]`
- Accessing a member variable from a vector element: `VECTOR[INDEX].MEMBER`

~

(c) Example output

```
-- SALARY PROGRAM --
YEAR      NAME                POSITION                      SALARY
--------------------------------------------------------------------------------
2023      Droo                President                     391567
2023      Barry               VP/Academic Affairs           202022
2023      Flan                Associate Professor           141140
2023      Zebra               Assistant Dean                115973
2023      Berra               Associate Professor           103906
2023      Arrdu               Associate Professor           85835
2023      Maure               Associate Professor           81470

-- GOODBYE! --
```

8.4.4 Graded program

1. Graded program

   This program allows the user to enter a series of grades and then it will calculate their GPA.

   (a) Instructions

   Within the while loop the user will be entering grades they've received in different classes, or 'N' to stop.

   - Within the while loop, add the `this_grade` value to the `course_grades` vector.

   After the while loop we will calculate the GPA.

   - Use a for loop to iterate over all the grades in the `course_grades` vector. Within the loop, add the individual grade value to the `total`.
   - After the for loop, calculate the average by taking the `total` divided by the `course_grades.size()`. Assign the result to the `result_gpa` variable.

   ~

   (b) Example output

   Running the program:

   ```
   1. Run AUTOMATED TESTS
   2. Run PROGRAMS
   >> 2
   GPA CALCULATOR
   ```

```
A = 4.0          B = 3.0         C = 2.0         D = 1.0         F = 0.0
Enter a grade: 4.0
Enter another? (Y/N): y

Enter a grade: 2.0
Enter another? (Y/N): y

Enter a grade: 1.0
Enter another? (Y/N): n

RESULT: 2.33
```

Running the automated tests:

```
1. Run AUTOMATED TESTS
2. Run PROGRAMS
>> 1
[PASS]  TEST 1, StudentCode({ 4.00, 4.00, 3.00, 2.00 }) = 3.25
[PASS]  TEST 2, StudentCode({ 4.00, 3.00, 2.00, 2.00, 1.00, 0.00 }) = 2.00
```

9 Mastery Check information

9.1 About the Mastery Check assignment, CS 200

9.1.1 Overview

- Assessments can be taken an unlimited amount of times. Highest score is saved. There is a 7 day "cooldown time" between attempts.
- Time limit is 6 hours, but taking the exam shouldn't take this long. The time limit is set so that you finish the exam *in the same day you start.*
  - Note that if you close the assignment you can come back to it later, but the timer won't stop.
- Questions in assessments are built from pools of questions, giving you a unique test each time.
- Assessment must be done solo, no help from other people or AI.
- Assessment is open-book, open-note, open-IDE.
- Take the assignment throughout the semester to gauge your progress in the course.

9.1.2 Questions

- Total questions: About 46
- Question types: Multiple choice, fill-in-the-blank/word bank, "essay" (manually graded coding)

Questions are split up into 3 tiers:

- The first tier is easier questions and they count for getting up to a C grade (70%). If you reference **Bloom's Taxonomy**, this would be the **Remembering**, **Understanding**, and **Applying** tier.
- The second tier is a little more in-depth and they count for up to a 90% grade. From **Bloom's Taxonomy**, this would be the **Analyze** and **Evaluate** tier.
- The third tier combines multiple topics together and doesn't hold your hand beyond giving the requirements to implement. From **Bloom's Taxonomy**, this would be the **Create** tier.

1. Question breakdown

Tier	Topic	Question count
Tier 1 (0-70%)	Computer basics	1 question
Tier 1 (0-70%)	C++ basics	4 questions
Tier 1 (0-70%)	Branching	3 questions
Tier 1 (0-70%)	Looping	3 questions
Tier 1 (0-70%)	File I/O, cin	6 questions
Tier 1 (0-70%)	Structs	3 questions
Tier 1 (0-70%)	Arrays and vectors	4 questions
Tier 1 (0-70%)	Functions	4 questions
Tier 1 (0-70%)	Classes and OOP	4 questions
Tier 1 (0-70%)	Pointers and memory	3 questions
Tier 2 (70-90%)	Debugging	3 questions
Tier 2 (70-90%)	Arrays	2 questions
Tier 2 (70-90%)	Functions	2 questions
Tier 2 (70-90%)	Classes	1 question
Tier 3 (90-100%)	Functions	1 question
Tier 3 (90-100%)	Classes	1 question

10 Semester project information

10.1 Semester project

10.1.1 Overview

Throughout this semester you will be build a single program for your project. You will be updating it periodically with new features as you work through the semester. As you implement each feature, your score for the project will increase. By the end of the semester, with all features completed, you should have 90-100% on the assignment.

Project should be worked on **solo**, though you can brainstorm and white-board with other students. Please refer others to snippets of code from the course contents if they need help with a topic.

Direct link:

- Project 1 documentation file

. . .

10.1.2 Program options

You'll decide what the program is "about". You can choose from one of these ideas below, or come up with your own.

1. A library book check-in check-out system. (Focus: book object)
2. A vending machine system. (Focus: soda/snack object)
3. A video game monster bestiary. (Focus: monster object)
4. A movie reviewing system. (Focus: movie object)
5. A recipe program. (Focus: ingredient object)

Your program should have one main **object** that it revolves around.

. . .

10.1.3 Turning in your work (first time)

1. **Code files:** You will be creating the files you need for the project, but make sure to put your work in the "Project" folder in your repository. **Only you and the instructor will be able to see your code.**

    - Use the "New Folder" command to create a folder named "Project".
    - Use the "New File" command to create a file within the Project folder. File name should be "main.cpp".
    - 

2. **Canvas post:** Post in the Canvas " **Semester project**" discussion board. Include the following:

    - Link to your repository (e.g., "https://gitlab.com/rsingh13/cs200") - only the instructor can access this, but it's to make grading a little easier on 'em. :)
    - Give a brief description of what your program is "about".
    - Show your program output. You can copy/paste the output as "preformatted" text, or upload a screenshot JUST OF YOUR PROGRAM, we don't need your entire desktop.
    - Answer the postmortem questions:
        (a) What went well?
        (b) What didn't go well?
        (c) What will you change for next time?

Example post:
My repository: https://gitlab.com/rsingh13/cs200
Program: A recipe program that shows ingredients (currently just 1).
Program output:

```
BUTTER RECIPE

Ingredient 1: Name: Butter
Measurement: Sticks    Amount: 2
```

1. What went well: Creating the main() program and the variables went well.
2. What didn't go well: I had a hard time formatting the output.
3. What will you change for next time: I will give myself more time to work on the output.

...

10.1.4 Turning in updates to your work

When updating your work, post a **reply** to your original thread for the new version. Include the following in your new post:

1. Which version number is this? ("Recipe program 2.0")
2. Program output again (copy/paste preformatted text, or a small screenshot)
3. Postmortem questions:
    (a) What went well?
    (b) What didn't go well?
    (c) What will you change for next time?

10.1.5 Program iterations

1. Iteration 1: main() and variables

   (a) Code requirements:

   - **Create a main() function.**
     - Create your source code file, add in main().
     - Add `#include` for any required libraries.
   - **Use variables to store data about your *object*.**
     - Come up with a SHORT list of attributes related to your *object*. For example, a "book" could have an ISBN, title, author, and price. A snack could have its name, calorie count, and price.
     - Create these variables and assign values to them. For now, the program just has information about *one object*, not many.
   - **Use `cout` to display information.**
     - Display your object's information in a user-friendly readable format.

   (b) Example output:

   ```
   BUTTER RECIPE

   Ingredient 1:
   - Name: Butter
   - Measurement: Cups
   - Amount: 1
   ```

   Make sure to answer the **postmortem questions** in your submission!

   ---

2. Iteration 2: branching

   (a) Code requirements:

   - **Update `main()` to take in arguments.**
     - Add `int argCount, char* args[]` into your main.
     - The argument input should be the user selecting option 1, 2, or 3, so convert the argument to an integer: `int choice = stoi( args[1] );`
   - **Use a branching statement (if statements, switch statement) to set values to your *variables*.**

- Based on whether the `choice` is 1, 2, or 3, assign your *object variables* to different values - such as having book 1, book 2, and book 3.

- **Continue displaying the *object* information.**

(b) Example output:

```
./proj.exe 1

COOKIE RECIPE

Ingredient 1:
- Name: Butter
- Measurement: Cups
- Amount: 1

./proj.exe 2

COOKIE RECIPE

Ingredient 2:
- Name: Flour
- Measurement: Cups
- Amount: 2.75

./proj.exe 3

COOKIE RECIPE

Ingredient 3:
- Name: Vanilla extract
- Measurement: Teaspoons
- Amount: 1
```

Make sure to answer the **postmortem questions** in your submission!

---

3. Iteration 3: file I/O

(a) Code requirements:

- **Add the `#include <fstream>` library to your program**
- **Create separate variables for all 3 of your *objects* now.** (e.g., "ingredient1Name", "ingredient2Name", etc.)
- **Save your *object data* at the end of the program.**

- Save all three objects' data at the end of the program, to a text file. Make sure it is in a structured order, such as "name", "calories", "price", each on their own lines.
- Test the program before implementing the load feature. Open up your save text file afterwards to make sure the save was successful.

- **Load your *object data* at the beginning of the program.**
  - Load all three objects' data at the start of the program, *after* declaring your variables. Remember that you saved the data out in a specific order, so you will load the data back in in a specific order.
- *Update the functionality where the user enters *object* 1, 2, or 3, so that now it displays the *object variables*' information, instead of setting information to a single set of variables.

(b) Example output:

Program output will look the same as before, but now you will have a text file. Make sure to include your text file in your program submission.

Example recipe.txt:

```
Butter
Cups
1
Flour
Cups
2.7
Vanilla extract
Teaspoons
1
```

Make sure to answer the **postmortem questions** in your submission!

---

4. Iteration 4: struct objects

(a) Code requirements:

- **Declare a struct for your objects in a OBJECT.h file** (use your object name instead of "OBJECT", e.g., "Igredient.h", "Book.h", etc.)
- **Update your program so that you declare and use 3 object-variables of that struct type.**

- Replace separate variables (e.g. "`string ingredient1Name; float ingredient1Amount; string ingredient1Measure;`") with one object variable using the struct type (e.g., "`Ingredient ingredient1;`")
- Update usages of the old variables with your new struct variables (e.g., instead of couting "`ingredient1Name`", it'd be "`ingredient1.name`").

(b) Example output:

Program output will look the same as before.

Make sure to answer the **postmortem questions** in your submission!

---

5. Iteration 5: arrays/vectors

(a) Code requirements:

- **Add `#include <vector>` to your program** so that you can use the vector class.
- **Replace your separate object variables with a vector of struct-objects.**
  - Instead of declaring three separate variables (e.g. "`Ingredient ingredient1, ingredient2, ingredient3;`"), declare a vector of those objects.
- **Remove the branching based on `choice` -** instead, use a for loop to iterate over all the *objects* and display information for every object.
- **Update the program argument to be # of new items:**
  - Now, have `choice` be the amount of new items the user would like to add to the list of objects.
  - If `choice` is greater than 0, then loop that amount of times. Create a temporary object variable (e.g. "`Ingredient newItem;`") and ask the user to fill out information for each of its fields. Then, push that new object into your vector.
- **Update your save/load to use loops**
  - Since we now cannot know exactly how many items are now in the vector, use a loop.
  - To save, you can use `MYVECTOR.size()` to get the amount of items to write out.
  - To load, you can load three items at a time each iteration through the loop.
  - If you're unsure of how to implement this, we will have examples in class.

- You may want to update your program output to display objects in a list or table so that it's more concise.

(b) Example output:

Program output:

```
./recipe.exe 1

ADD 1 NEW INGREDIENT
Name: Sugar
Measurement: Cups
Amount: 1.5

COOKIE RECIPE

- 1 Cups of Butter
- 2.75 Cups of Flour
- 1 Teaspoons of Vanilla extract
- 1.5 Cups of Sugar
```

File output:

```
Butter
Cups
1
Flour
Cups
2.7
Vanilla extract
Teaspoons
1
Sugar
Cups
1.5
```

Make sure to answer the **postmortem questions** in your submission!

---

6. Iteration 6: functions

(a) Code requirements:

- **Create Functions.h and Functions.cpp files.**
  - Your function declarations will go in the .h file.
  - Your function definitions will go in the .cpp file.

- **Move your program's features into separate functions, ONE ITEM AT A TIME!** (If you try to do multiple at once, it will be hard to test!)
  - `void CreateNew( vector<THING>& objects );`
    - Contains the code that asks the user for new object information, then pushes the new object to the vector.
  - `void Save( const vector<THING>& objects, const string& filename );`
    - Saves the vector's contents to a file, using the filename given.
  - `void Load( vector<THING>& objects, const string& filename );`
    - Loads the data into the vector, from the filename specified.
  - `void Display( const vector<THING>& objects );`
    - Iterates over all the items in the vector and displays each one.

(b) Example output:

Will be the same as previously.

Make sure to answer the **postmortem questions** in your submission!

---

7. Iteration 7: classes

(a) Code requirements:

- **Create an OBJECT.cpp file to go with your OBJECT.h file.**
  - Convert your OBJECT struct into a **class**.
  - Make the member variables `private`.
  - Add Get/Set functions to the class.
  - (Optional): Add a Setup function to set everything up at once.
  - (Optional): Add constructor functions to make setup easier.
- **Update the program to use the object's functions instead.**
  - Access to object variables will have to be updated to functions (e.g., `ing[1].name` becomes `ing[1].GetName()`.)

(b) Example output:

Will be the same as previously.

Make sure to answer the **postmortem questions** in your submission!

11 Common general issues

11.1 g++: error: No such file or directory. fatal error: no input files.

Error:

```
g++: error: No such file or directory. fatal error: no input files.
```

Solution:
This means that you used "Open Folder" in the wrong location. Please make sure to open up the subfolder for a given practice or graded program to work on in VS Code.

11.2 make is not recognized as the name of a cmdlet.

Error:

```
make : The term 'make' is not recognized as the name of a cmdlet, function, script file, or ope
that the path is correct and try again.
At line:1 char:1
+ make debug
+ ~~~~
+ CategoryInfo : ObjectNotFound: (make:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Solution:
You need to go to C:on your computer and rename the "mingw32-make.exe" program to "make.exe".

12 Common mac-related issues

12.1 non-aggregate type cannot be initialized with an initializer list

```
error: non-aggregate type 'vector<std::string>' cannot be initialized with an initializer list.
```

Cause: Building on Mac defaults to the 1998 version of C++ instead of a more modern one. In order to fix this for any programs using `vector<THING> = { /* stuff */ };`, you'll need to specify the C++ version during your build:
Example: Build with C++ 2011:

```
g++ myfile.cpp -std=c++11
```

1998, 2011, 2014, 2017, and 2023 are major versions.
~

12.2 gdb doesn't work on Mac!

The GDB debugger isn't available on mac, but you can use the LLDB debugger instead, which should already be installed if you installed your tools via XCode.
See the LLDB Reference Page] for steps!

---

13 Syllabus

13.1 Course information

College	Johnson County Community College
Division	CSIT (Computer Science/Information Technology)
Instructor	R.W. Singh (they/them)
Semester	Spring 2025 (1/27/2025 - 5/19/2025)
Course	CS 200: Concepts of Programming with C++ (4 credit hours)
Section	Section 400 / CRN 11909 / HyFlex
Schedule	Mondays, 6:00 - 8:50 pm
Room	Regnier center, room 347
Office hours	Mondays, 3:30 - 5:30 pm; Tuesdays 9:30 - 10:30 am; Tuesdays 4:30 - 5:30 pm; Thursdays 9:30 - 10:30am

Course description This course emphasizes problem solving using a high-level programming language and the software development process. Algorithm design and development, programming style, documentation, testing and debugging will be presented. Standard algorithms and data structures will be introduced. Data abstraction and an introduction to object-oriented programming will be studied and used to implement algorithms. 3 hrs. lecture, 2 hrs. lab by arrangement/wk. Catalog link: https://catalog.jccc.edu/coursedescriptions/cs/#CS_200

Prerequisites (CS 134 with a grade of "C" or higher or CIS 142 with a grade of "C" or higher or department waiver test and MATH 131 or higher) or MATH 241 or department approval.

Drop deadlines To view the deadline dates for dropping this course, please refer to the schedule on the JCCC website under Admissions>Enrollment Dates> Dropping Credit Classes. After the 100% refund date, you will be financially responsible for the tuition charges; for details, search on Student Financial Responsibility on the JCCC web page. Changing your schedule may reduce eligibility for financial aid and other third party funding. Courses not dropped will be graded. For questions about dropping courses, contact the Student Success Center at 913-469-3803.

- Academic Calendar: https://www.jccc.edu/modules/calendar/academic-calendar.html
- **First week attendance and Auto-withdraws:** Attendance for the first week of classes at JCCC are recorded and students marked as NOT IN ATTENDANCE get auto-withdrawn from the course. Please pay attention to course announcements / emails from the instructor for instructions on how to make sure you are marked as IN ATTENDANCE.
- To learn more about reinstatement, please visit: https://www.jccc.edu/admissions/enrollment/reinstatement.html
- **Faculty-Initiated Withdrawal:** The instructor may opt to withdraw you from the class as a result of extended lack of contact and coursework being done; see Attendance policies section for more.

13.1.1 Instructor information

- Name: R.W. Singh (aka "Moose")
- Pronouns: they/them
- Office: RC 348 H
- Email: rsingh13@jccc.edu (Canvas Inbox messages preferred)
- Office phone: (913) 799-3671

1. Class communication

    - **Please prefer Canvas Inbox** - My direct @ jccc work email is full of other work related emails, I will see your email *fastest* if you email me via Canvas.
    - **Reply speed** - I will attempt to reply within 1 business day of receiving your message.
    - **Course announcements** - I will periodically post Announcements on Canvas, which may have assignment fixes, course news, etc. Please make sure to keep an eye on it.

13.1.2 Course delivery (HyFlex)

This course is set up as a **HyFlex** course. This means the following:

- Courses have a scheduled "in-class" time each week.
- Students can choose to attend class in one of three ways:

    1. In person in the classroom during class time.
    2. Remotely via Zoom during class time.
    3. Watch the archived Zoom lecture *after* class time. **If you do not attend the class section, I expect that you will watch the archived class video afterward so that you stay up-to-date on course news.**

- A Zoom link will be available for each class session. Please see the Canvas page, under "Zoom", for the link.
- Class sessions will be recorded and you can view them afterwards. They will be posted to the Canvas main page once available.

To see more about JCCC course delivery options, please visit: https://www.jccc.edu/student-resources/course-delivery-methods.html

13.1.3 Student drop-in times (office hours)

Office hours for **Spring 2025** are:

- Mondays, 3:30 - 5:30 pm
- Tuesdays, 9:30 - 10:30 am
- Tuesdays, 4:30 - 5:30 pm
- Thursdays, 9:30 - 10:30 am

I will be available on campus or via Zoom. Zoom link will be posted on Canvas under "office hours". Office hours are time for you to drop in whenever and ask questions as needed.

13.1.4 Course supplies

1. **Textbook:** Rachel's CS 200 course notes (https://moosadee.gitlab.io/courses/)
    - I will link to each reading item on Canvas.
2. **Zoom:** Needed for remote attendance / office hours
3. **Tools:** See first week assignments for setup instructions.
    - WINDOWS: g++ (MinGW), make, git, gdb
    - LINUX: g++, make, git, gdb
    - MAC: brew, g++, make, git, gdb
    - VS Code (https://code.visualstudio.com/) or VS Codmium (https://vscodium.com/).
4. **Accounts:** See first week assignments for setup instructions.
    - GitLab (https://gitlab.com/) for code storage
5. **Optional:** Things that might be handy
    - Dark Reader plugin (Firefox/Chrome/Safari/Edge) to turn light-mode webpages into dark-mode. (https://darkreader.org/)

13.1.5 Recommended experience

Computer skills - You should have a base level knowledge of using a computer, including:

- Navigating your Operating System, including:
    - Installing software
    - Running software

  - Locating saved files on your computer
  - Writing text documents, exporting to PDF
  - Taking screenshots
  - Editing .txt and other plaintext files
- Navigating the internet:
  - Navigating websites, using links
  - Sending emails
  - Uploading attachments

Learning skills - Learning to program takes a lot of reading, and you will be building up your problem solving skills. You should be able to exercise the following skills:

- Breaking down problems - Looking at a problem in small pieces and tackling them one part at a time.
- Organizing your notes so you can use them for reference while coding.
- Reading an entire part of an assignment before starting - these aren't step-by-step to-do lists.
- Learning how to ask a question - Where are you stuck, what are you stuck on, what have you tried?
- Recognizing when additional learning resources are needed and seeking them out - such as utilizing JCCC's Academic Achievement Center tutors.
- Managing your time to give yourself enough time to tackle challenges, rather than waiting until the last minute.

How to ask questions - When asking questions about a programming assignment via email, please include the following information so I can answer your question:

1. Be sure to let me know WHICH ASSIGNMENT IT IS, the specific assignment name, so I can find it.
2. Include a SCREENSHOT of what's going wrong.
3. What have you tried so far?

13.2 Course policies

13.2.1 Grading breakdown

Assessment types are given a certain weight in the overall class. Breakdown percentages are based off the course catalog requirements (https://catalog.jccc.edu/coursedescriptions/cs/#CS_200)

Final letter grade: JCCC uses whole letter grades for final course grades: F, D, C, B, and A. The way I break down what your receive at the end of the semester is as follows:

Total score	Letter grade
89.5% <= grade <= 100%	A
79.5% <= grade < 89.5%	B
69.5% <= grade < 79.5%	C
59.5% <= grade < 69.5%	D
0% <= grade < 59.5%	F

Grading style: All assignments begin at 0% at the start of the semester. As you work through course content, your grade will grow and will not go down. Toward the end of the semester the score you have reflects your final grade in the course, so you will be working towards the grade you want.

Assignment types:

- **Exams** (40% of grade)
- **Project** (20% of grade) - A programming project that you will work on throughout the semester.
- **Labs** (20% of grade) - Weekly programming labs to practice new topics.
- **Concept intros** (10% of grade) - Weekly reading and review quiz material.
- **Exercises** ()
  - **Tech Literacy** (5% of grade) - Discussion boards to expand learning of tech topics.
  - **Status updates** (5% of grade) - Weekly updates on how you're doing with the course topics.

13.2.2 Due dates, late assignments, re-submissions

- **Due dates** are set as a guide for when you *should* have your assignments in by.
  - I do not count off points for "late" assignments. Just get the assignment in by the "available until" date.
- **End dates/available until dates** are a hard-stop for when an assignment can be turned in. Assignments cannot be turned in after this date.
- **Resubmissions** to some assignments are permitted:

- **Concept Introductions** are auto-graded and you can resubmit them as much as you'd like, with the highest score being saved.

- **Labs** are manually graded but you can turn in fixes after receiving feedback from the instructor.

13.2.3 Attendance

First week of class: JCCC requires us to take attendance during the first week of the semester. Students are required to attend class (if there is a scheduled class session) this first week. If there are scheduling conflicts during the first week of class, please reach out to the instructor to let them know. JCCC auto-drops students marked as not in attendance during the first week of class, but students can be reinstated. See https://www.jccc.edu/admissions/enrollment/reinstatement.html for more details.

HyFlex classes: The following three scenarios count as student attendance for my classes:

1. Attending class in person during the class times, or

2. Attending class remotely via Zoom during class times, or

3. Watching the recorded Zoom class afterwards. **If you do not attend the class session I will expect you to watch the archived class video so that you keep up-to-date on class news and topics.**

Online classes: Attendance is counted as completion of assignments for a given week.

Faculty-Initiated Withdrawal: Following the Administrative Drop for Non-Attendance period of each semester (see Section I.A above), a faculty member may choose to withdraw a student whose absences have exceeded the attendance guidelines stated in the course syllabus. There is no reimbursement or forgiveness of tuition and fees for a Faculty-Initiated Withdrawal. Students should not assume that a faculty member will initiate this optional process, and it remains the ultimate responsibility of the student to withdraw and accept all financial and academic consequences as a result of the withdrawal.

Faculty initiated withdrawal may be taken after the faculty member has notified the student through the Excessive Absence Alert procedure that excessive absence has potentially placed the student in academic jeopardy. The withdrawal will be recorded in the student's record in accordance with the published drop deadlines and the Grading System Policy. The student may also be withdrawn from other scheduled courses if the withdrawn course is a required course. The last date each semester for a faculty-initiated withdrawal shall be the same last date allowed for a student-initiated withdrawal.

The Excessive Absence Alert shall consist of a written notice from the faculty member to the student advising the student that the student's excessive absence has placed the student in academic jeopardy. The notice shall further state that the student may be withdrawn from the class as per course syllabus guidelines if satisfactory arrangements for the student's regular class attendance cannot be made with the faculty member. Such written notice shall be provided to the student via email to the student's College-provided email account and

shall constitute adequate notice to the student. Students are responsible for monitoring their College-provided email accounts.

See JCCC's Student Attendance Operating Procedure 314.01: https://www.jccc.edu/about/leadership-governance/policies/students/academic/procedure-attendance.html

- I may initiate student withdrawal if student fails to submit a month work of course assignments.
- You should notify me of extended absenses (not able to do coursework) before your absense. Upon returning, you should make an effort to work on the late work weekly and work to catch up on course content.

13.2.4 Tentative schedule

This schedule is **recommended**, but the modules are available whenever you meet the prerequisites for any of them.

Week #	Monday	Recommended topics
1	Jan 27	Course policies, tools setup, computer history, C++ programs
2	Feb 3	Branching, testing
3	Feb 10	Looping, debugging
4	Feb 17	Strings, console I/O, file I/O
5	Feb 24	Structs
6	Mar 3	Pointers and memory
7	Mar 10	Arrays, Vectors, and dynamic arrays
		Mar 17 - Mar 23 - Spring Break
8	Mar 24	Functions, searching algorithms
9	Mar 31	Class basics
10	Apr 7	More class features
11	Apr 14	Composition and inheritance
12	Apr 21	Recursion, sorting algorithms
13	Apr 28	Catch-up / Project time
14	May 5	Catch-up / Project time
15	May 12	Finals week (See JCCC finals schedule)

13.2.5 Class format

Class sessions are flexible and can be changed to suit student requests. By default, class sessions are usually used for:

- Working through example code
- An overview of the assignments for the week
- In-class working time

Generally, I do not lecture during class times; there are video lectures and reading assignments that students can complete independently. Class times for this course are better used for students to get hands-on experience with the new topics while having the instructor available to answer questions and make clarifications.

Grade scoring:

- All assignment scores begin at **0%** at the start of the semester, so your grade begins at 0% at the start.
- As you progress through the course and finish more course content, your grade will continue rising. Ideally, if you get 100% on all assignments in the course, you'll end with 100% as your final grade.
- Most assignments can be resubmitted after grading to improve your score afterwards. (e.g., if I notice bugs in your lab, I'll make comments on why it happens, and you can go back and fix it.)
- Assignments don't close until the **last day of class - July 25th**. See the Academic Calendar (https://www.jccc.edu/modules/calendar/academic-calendar.html) if needed.

13.2.6 Academic honesty

The assignments the instructor writes for this course are meant to help the student learn new topics, starting easy and increasing the challenge over time. If a student does not do their own work then they miss out on the lessons and strategy learned from going from step A to step B to step C. The instructor is always willing to help you work through assignments, so ideally the student shouldn't feel the need to turn to third party sources for help.

Generally, for R.W. Singh's courses:

- OK things:
  - Asking the instructor for help, hints, or clarification, on any assignment.
  - Posting to the discussion board with questions (except with tests - please email me for those). (If you're unsure if you can post a question to the discussion board, you can go ahead and post it. If there's a problem I'll remove/edit the message and just let you know.)

  - Searching online for general knowledge questions (e.g. "C++ if statements", error messages).
  - Working with a tutor through the assignments, as long as they're not doing the work for you.
  - Use your IDE (replit, visual studio, code::blocks) to test out things before answering questions.
  - Brainstorming with classmates, sharing general information ("This is how I do input validation").

- Not OK Things:
  - Sharing your code files with other students, or asking other students for their code files.
  - Asking a tutor, peer, family member, friend, AI, etc. to do your assignments for you.
  - Searching for specific solutions to assignments online/elseware.
  - Basically, any work/research you aren't doing on your own, that means you're not learning the topics.
  - Don't give your code files to other students, even if it is "to verify my work!"
  - Don't copy solutions off other parts of the internet; assignments get modified a little bit each semester.

If you have any further questions, please contact the instructor.

Each instructor is different, so make sure you don't assume that what is OK with one instructor is OK with another.

13.2.7 Student success tips

- **I need to achieve a certain grade for my financial aid or student visa. What do I need to plan on?**
  - If you need to get a certain grade, such as an A for this course, to maintain your financial aid or student visa, then you need to set your mindset for this course immediately. You should prioritize working on assignments early and getting them in ahead of time so that you have the maximum amount of time to ask questions and get help. You should not be panicking at the end of the semester because you have a grade less than what you need. From week 1, make sure you're committed to staying on top of things.

- **How do I contact the instructor?**

- The best way to contact the instructor is via Canvas' email system. You can also email the instructor at rsingh13@jccc.edu, however, emails are more likely to be lost in the main inbox, since that's where all the instructor's work-related email goes. You can also attend Zoom office hours to ask questions.

- **What are some suggestions for approaching studying and assignments for this course?**

  - Each week is generally designed with this "path" in mind:

    * Watch lecture videos, read assigned reading.
    * Work on Concept Introduction assignment(s).
    * Work on Exercise assignment.

  - Those are the core topics for the class. The Tech Literacy assignments can be done a bit more casually, and the Topic Mastery (exams) don't have to be done right away - do the exams once you feel comfortable with the topic.

- **Where do I find feedback on my work?**

  - Canvas should send you an email when there is feedback on your work, but you can also locate assignment feedback by going to your Grades view on Canvas, locating the assignment, and clicking on the speech balloon icon to open up comments. These will be important to look over during the semester, especially if you want to resubmit an assignment for a better grade.

- **How do I find a tutor?**

  - JCCC's Academic Achievement Center

    (https://www.jccc.edu/student-resources/academic-resource-center/academic-achievement-center/)

    provides tutoring services for our area. Make sure to look for the expert tutor service and you can learn more about getting a tutor.

- **How do I keep track of assignments and due dates so I don't forget something?**

  - Canvas has a CALENDAR view, but it might also be useful to utilize something like Google Calendar, which can text and email you reminders, or even keeping a paper day planner that you check every day.

13.2.8 Accommodations and life help

- **How do I get accommodations? - Access Services**

  https://www.jccc.edu/student-resources/access-services/

  Access Services provides students with disabilities equal opportunity and access. Some of the accommodations and services include testing accommodations, note-taking assistance, sign language interpreting services, audiobooks/alternative text and assistive technology.

- **What if I'm having trouble making ends meet in my personal life? - Student Basic Needs Center**

  https://www.jccc.edu/student-resources/basic-needs-center/

  Check website for schedule and location. The JCCC Student Assistance Fund is to help students facing a sudden and unforeseen emergency that has affected their ability to attend class or otherwise meet the academic obligations of a JCCC student. If you are experiencing food or housing insecurity, or other hardships, stop by COM 319 and visit with our helpful staff.

- **Is there someone I can talk to for my degree plan? - Academic Advising**

  https://www.jccc.edu/student-resources/academic-counseling/

  JCCC has advisors to help you with:

  - Choose or change your major and stay on track for graduation.
  - Ensure a smooth transfer process to a 4-year institution.
  - Discover resources and tools available to help build your schedule, complete enrollment and receive help with coursework each semester.
  - Learn how to get involved in Student Senate, clubs and orgs, athletics, study abroad, service learning, honors and other leadership programs.
  - If there's a hold on your account due to test scores, academic probation or suspension, you are required to meet with a counselor.

- **Is there someone I can talk to for emotional support? - Personal Counseling**

  https://www.jccc.edu/student-resources/personal-counseling/

  JCCC counselors provide a safe and confidential environment to talk about personal concerns. We advocate for students and assist with personal issues and make referrals to appropriate agencies when needed.

- **How do I get a tutor? - The Academic Achievement Center**

  https://www.jccc.edu/student-resources/academic-resource-center/academic-achievement-center/

  The AAC is open for Zoom meetings and appointments. See the website for their schedule. Meet with a Learning Specialist for help with classes

and study skills, a Reading Specialist to improve understanding of your academic reading, or a tutor to help you with specific courses and college study skills. You can sign up for workshops to get off to a Smart Start in your semester or analyze your exam scores!

- **How can I report ethical concerns? - Ethics Report Line**

  https://www.jccc.edu/about/leadership-governance/administration/audit-advisory/ethics-line/

  You can report instances of discrimination and other ethical issues to JCCC via the EthicsPoint line.

- **What other student resources are there? - Student Resources Directory**

  https://www.jccc.edu/student-resources/

13.3 Additional information

13.3.1 ADA compliance / disabilities

JCCC provides a range of services to allow persons with disabilities to participate in educational programs and activities. If you are a student with a disability and if you are in need of accommodations or services, it is your responsibility to contact Access Services and make a formal request. To schedule an appointment with an Access Advisor or for additional information, you can contact Access Services at (913) 469-3521 or accessservices@jccc.edu. Access Services is located on the 2nd floor of the Student Center (SC202)

13.3.2 Attendance standards of JCCC

Educational research demonstrates that students who regularly attend and participate in all scheduled classes are more likely to succeed in college. Punctual and regular attendance at all scheduled classes, for the duration of the course, is regarded as integral to all courses and is expected of all students. Each JCCC faculty member will include attendance guidelines in the course syllabus that are applicable to that course, and students are responsible for knowing and adhering to those guidelines. Students are expected to regularly attend classes in accordance with the attendance standards implemented by JCCC faculty.

The student is responsible for all course content and assignments missed due to absence. Excessive absences and authorized absences are handled in accordance with the Student Attendance Operating Procedure.

13.3.3 Academic Dishonesty

No student shall attempt, engage in, or aid and abet behavior that, in the judgment of the faculty member for a particular class, is construed as academic dishonesty. This includes, but is not limited to, cheating, plagiarism or other forms of academic dishonesty.

Examples of academic dishonesty and cheating include, but are not limited to, unauthorized acquisition of tests or other academic materials and/or distribution of these materials, unauthorized sharing of answers during an exam, use of unauthorized notes or study materials during an exam, altering an exam and resubmitting it for re-grading, having another student take an exam for you or submit assignments in your name, participating in unauthorized collaboration on coursework to be graded, providing false data for a research paper, using electronic equipment to transmit information to a third party to seek answers, or creating/citing false or fictitious references for a term paper. Submitting the same paper for multiple classes may also be considered cheating if not authorized by the faculty member.

Examples of plagiarism include, but are not limited to, any attempt to take credit for work that is not your own, such as using direct quotes from an author without using quotation marks or indentation in the paper, paraphrasing work that is not your own without giving credit to the original source of the idea, or failing to properly cite all sources in the body of your work. This includes use of complete or partial papers from internet paper mills or other sources of non-original work without attribution.

A faculty member may further define academic dishonesty, cheating or plagiarism in the course syllabus.

13.3.4 College Wellness and Safety

College Wellness and Safety (https://www.jccc.edu/media-resources/wellness-safety/)

- Stay home when you're sick
- Wash hands frequently
- Cover your mouth when coughing or sneezing
- Clean surfaces
- Facial coverings are available and welcomed but not required
- Wear yoru name badge or carry your JCCC photo id while on campus

13.3.5 College emergency response plan

https://www.jccc.edu/student-resources/police-safety/police-department/college-emergency-response-plan/

13.3.6 Student code of conduct policy

http://www.jccc.edu/about/leadership-governance/policies/students/student-code-of-conduct/student-code-conduct.html

13.3.7 Student handbook

http://www.jccc.edu/student-resources/student-handbook.html

13.3.8 Campus safety

Information regarding student safety can be found at http://www.jccc.edu/student-resources/police-safety/. Classroom and campus safety are of paramount importance at Johnson County Community College and are the shared responsibility of the entire campus population. Please review the following:

- **Report emergencies:** to Campus Police (available 24 hours a day)
  - In person at the Midwest Trust Center (MTC 115)
  - Call 913-469-2500 (direct line) – Tip: program in your cell phone
  - Phone app - download JCCC Guardian (the free campus safety app: www.jccc.edu/guardian) - instant panic button and texting capability to Campus Police
  - Anonymous reports to KOPS-Watch - https://secure.ethicspoint.com/domain/en/report_company.asp?clientid=25868 or 888-258-3230
- **Be Alert:**
  - You are an extra set of eyes and ears to help maintain campus safety
  - Trust your instincts
  - Report suspicious or unusual behavior/circumstances to Campus Police (see above)
- **Be Prepared:**
  - Identify the red/white stripe Building Emergency Response posters throughout campus and online that show egress routes, shelter, and equipment
  - View A.L.I.C.E. training (armed intruder response training - Alert, Lockdown, Inform, Counter and/or Evacuate)
    - Student training video: https://www.youtube.com/watch?v=kMcT4-nWSq0
  - Familiarize yourself with the College Emergency Response Plan: (jccc.edu/student-resources/police-safety/college-emergency-response-plan/)
- **During an emergency:** Notifications/Alerts (emergencies and inclement weather) are sent to all employees and students using email and text messaging
  - students are automatically enrolled, see JCCC Alert - Emergency Notification: (jccc.edu/student-resources/police-safety/jccc-alert.html)

- **Weapons policy:** Effective July 1, 2017, concealed carry handguns are permitted in JCCC buildings subject to the restrictions set forth in the Weapons Policy. Handgun safety training is encouraged of all who choose to conceal carry. Suspected violations should be reported to JCCC Police Department 913-469-2500 or if an emergency, you can also call 911.

13.4 Course catalog info

https://catalog.jccc.edu/coursedescriptions/cs/#CS_200

Objectives:

1. Describe computer systems and examine ethics.
2. Solve problems using a disciplined approach to software development.
3. Utilize fundamental programming language features.
4. Implement procedures.
5. Employ fundamental data structures and algorithms.
6. Write code using object-oriented techniques.
7. Write code according to commonly accepted programming standards.
8. Utilize a professional software development environment.

Content Outline and Competencies:

```
I. Computer Systems and Ethics
  A. Describe basic software components.
    1. Describe operating systems.
    2. Describe high-level and machine languages.
    3. Describe compilers.
  B. Examine ethics.
    1. Examine ethics in the context of software development.
    2. Examine the impact of ethics violations on software developers.
    3. Examine the impact of ethics violations on software users.
II. Problem-Solving in Software Development
  A. Define the problem.
  B. Develop a solution.
    1. Utilize top-down design.
    2. Consider previous problems and solutions.
    3. Reuse pertinent algorithms.
    4. Represent algorithms with pseudo-code.
    5. Identify input, output, processing and modules.
  C. Code the solution.
  D. Test the solution.
    1. Perform unit and integration testing.
    2. Select appropriate test data.
    3. Trace code by hand (desk-checking) and with a debugger.
    4. Evaluate code efficiency and simplicity.
```

III. Fundamental Programming Features
  A. Declare and initialize variables and constants.
  B. Use built-in operators to create expressions and statements.
    1. Write assignment statements.
    2. Create expressions with arithmetic, relational and logical operators.
    3. Use the conditional (ternary) operator.
    4. Evaluate expressions using rules of operator precedence.
    5. Compare strings and numeric types.
    6. Dereference and assign values to pointers.
  C. Perform input and output.
    1. Retrieve data from the keyboard.
    2. Retrieve data from input files.
    3. Write data to the console window.
    4. Write data to output files.
  D. Call built-in mathematical functions.
  E. Implement type-casting.
  F. Control program flow.
    1. Implement selection statements.
      a. Write code with if, else and else-if statements.
      b. Use switch statements.
      c. Write nested selection statements.
    2. Implement repetition statements
      a. Write while, for and do loops.
      b. Create nested loops.
      c. Analyze break and continue semantics.
  G. Trap errors using selection or repetition.
IV. Procedures
  A. Define and call functions with void and non-void return values.
  B. Declare functions (prototyping).
  C. Implement pass-by-value and pass-by-reference parameters.
  D. Differentiate between actual and formal parameters.
  E. Analyze and write elementary recursive code.
  F. Analyze variable scope and lifetime.
  G. Implement static variables.
V. Fundamental Data Structures and Algorithms
  A. Implement single dimensional arrays.
    1. Implement an array of integers.
    2. Implement null-terminated strings.
  B. Implement two-dimensional arrays.
  C. Implement dynamic arrays.
    1. Use new and delete to manage memory.
    2. Declare pointers.
  D. Search arrays.
    1. Implement sequential search.
    2. Implement binary search.
  E. Sort arrays.
    1. Sort data using bubble sort.
    2. Sort data using selection sort.
    3. Sort data using insertion sort.
  F. Implement structures.

G. Implement an array of structures.

VI. Object-oriented Programming

A. Write code using the built-in string class and associated methods.

B. Write code using the built-in vector class and associated methods.

C. Implement encapsulation and data abstraction by writing user-defined classes.

D. Differentiate between private and public access modifiers.

E. Hide member data.

F. Write accessors, mutators and other member functions that process member data.

G. Write code that utilizes objects.

H. Implement an array of objects.

VII. Code Standards

A. Create descriptive identifiers according to language naming conventions.

B. Write structured and readable code.

C. Create documentation.

VIII. Professional Development Environment

A. Write code using a professional, integrated development environment (IDE).

B. Utilize key editor features.

C. Debug code using the integrated debugger.

D. Include and use standard libraries.