CS 250: Basic Data Structures using C++ (Spring 2025 version)

Rachel Wil Sha Singh

March 9, 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 1: Welcome, setup, CS 200 review

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.

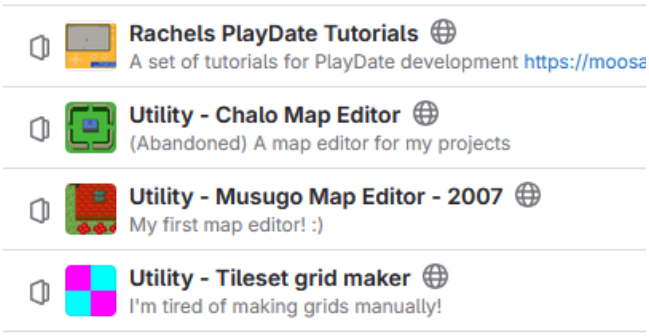
1.4 Intro: Source Control and git

1.4.1 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.

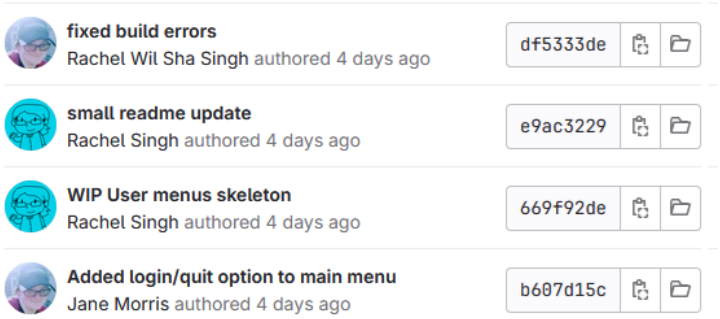
1.4.2 Repositories



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.

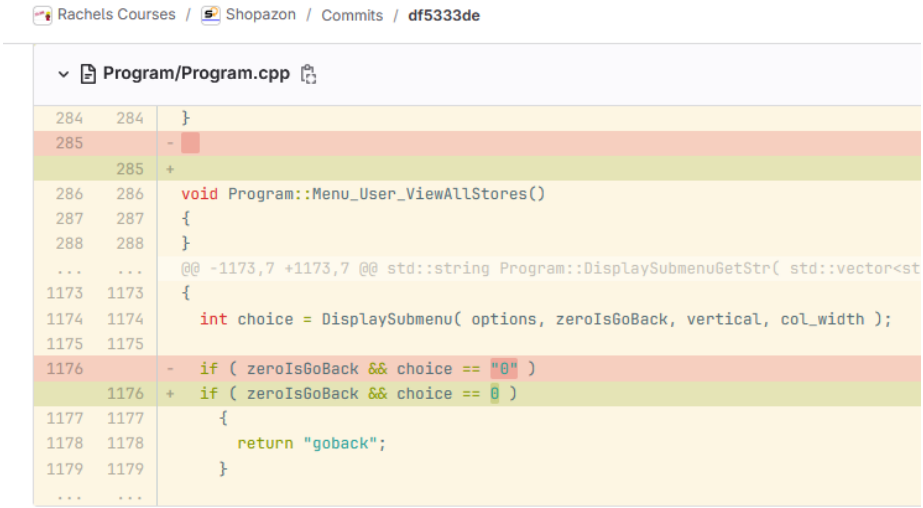
A Git Repository is set up so that Git takes care of tracking changes over time, create branches to work on new features, automatically merge changes with other people, and more.

1.4.3 Commit log



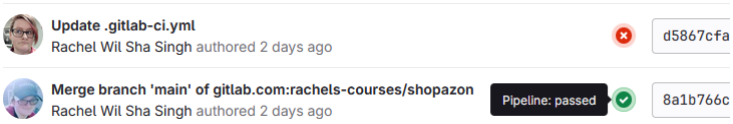
Git keeps track of changes made by different developers over time. Each time we use the commit command, a "snapshot" of our changes are made. Once synced to the GitLab server, we can see a list of all commits in the history of the project.

1.4.4 Git blame



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

1.4.5 Continuous integration

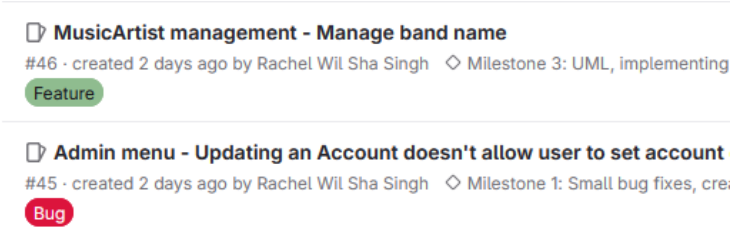


The system architect can also set up scripts so that whenever new code is synced on the server, a build is run - this builds the project (and usually runs automated tests), and reports if the build/tests are successful or not.

This helps us know if a new feature we're working on breaks the build or makes other tests fail.

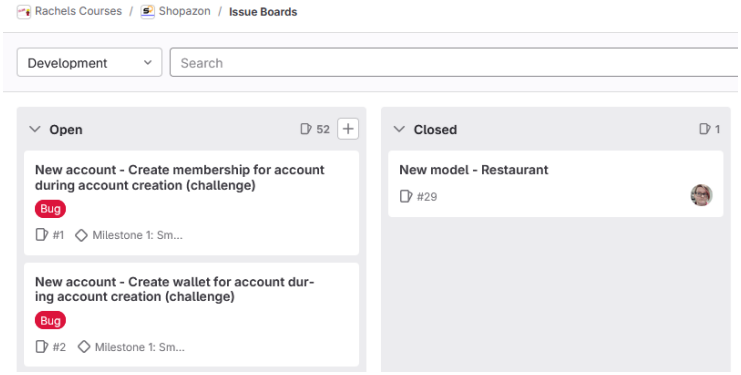
(You won't have to configure this.)

1.4.6 GitLab - Issues



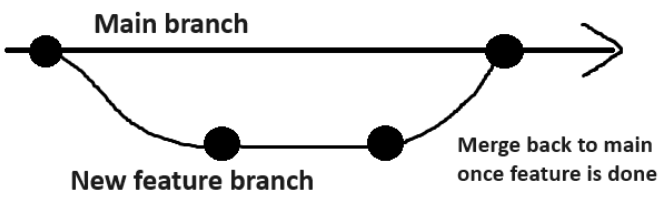
GitLab also has an Issue Tracker, where we can create Issues (aka Tickets) to log tasks that need to be completed or bugs found in the code. We can associate our commits to an issue number # so that we can easily reference when and where a feature was added or a bug was fixed.

1.4.7 GitLab - Issue board



GitLab also has other project management features such as a "Issue Board" (often called a Kanban board) to more visually track what needs to get done.

1.4.8 Branches

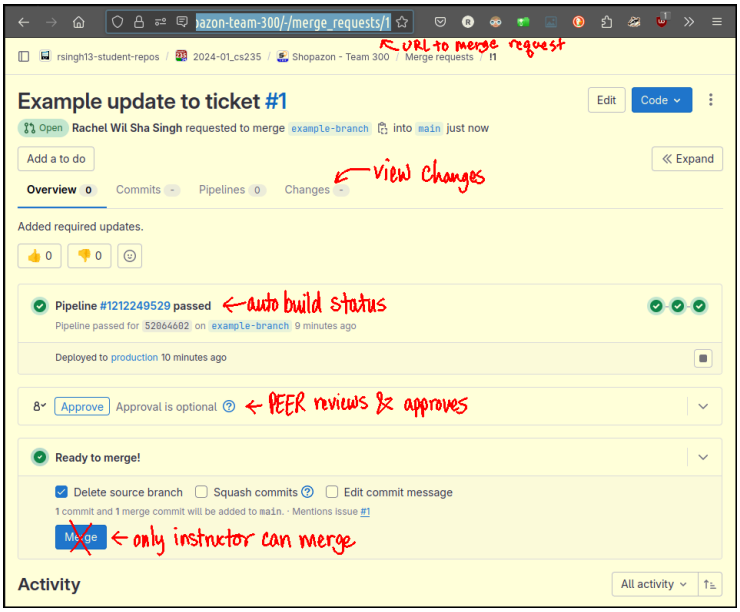


When a developer begins working on a new feature the common practice is for them to create a new branch. This makes a clone of our main branch and allows the developer to add new code, without affecting main.

Generally, the main branch should only contain code that is complete and tested, so the developer works in their feature branch while developing a feature.

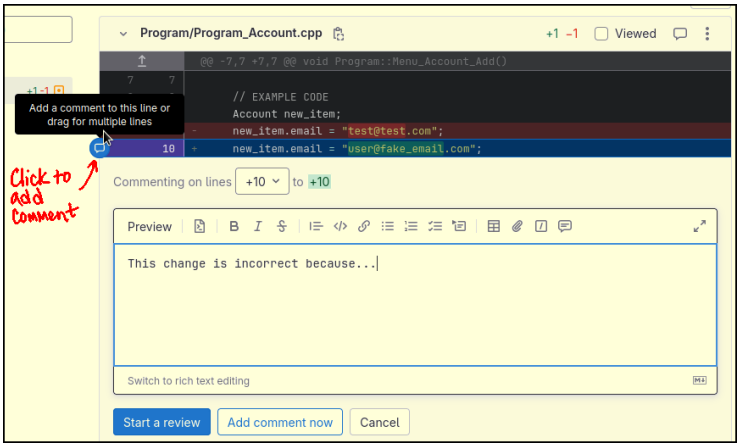
Once they're done with a feature, usually a peer will review their code, and the team lead will merge the feature branch with main.

1.4.9 Merge requests



Once a developer is done with creating a new feature on their own feature branch, then they create a Merge Request on GitLab. This will create a view where other developers can view all the commits/changes related to this feature, add comments, and approve the merge request if everything looks good.

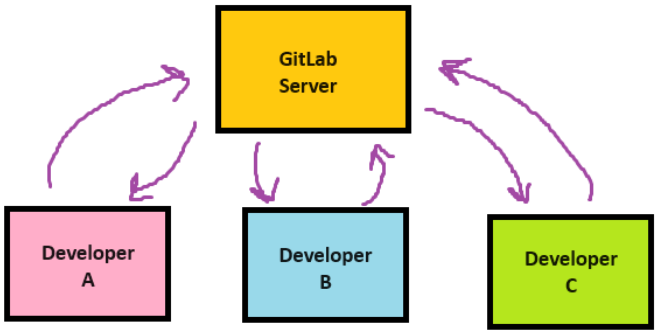
1.4.10 Code reviews



Code reviews are a common part of software development. When a developer is done with a feature they will create a merge request for their branch. Other developers will read through the code commit looking for obvious issues, adding comments if anything is found.

If the merge request gets approved then the team lead merges the feature branch into the main branch.

1.4.11 Git vs. GitLab

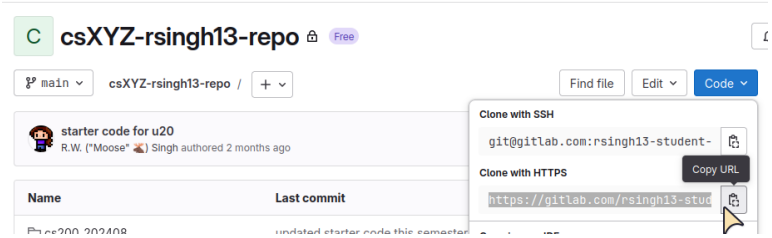


- Git itself is a whole system that facilitates this organized software development.
- GitLab is a service that provides hosting for Git repositories. (GitHub is another common example.)
- Each developer installs the Git software to their computer, which allows them to communicate to the repository's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

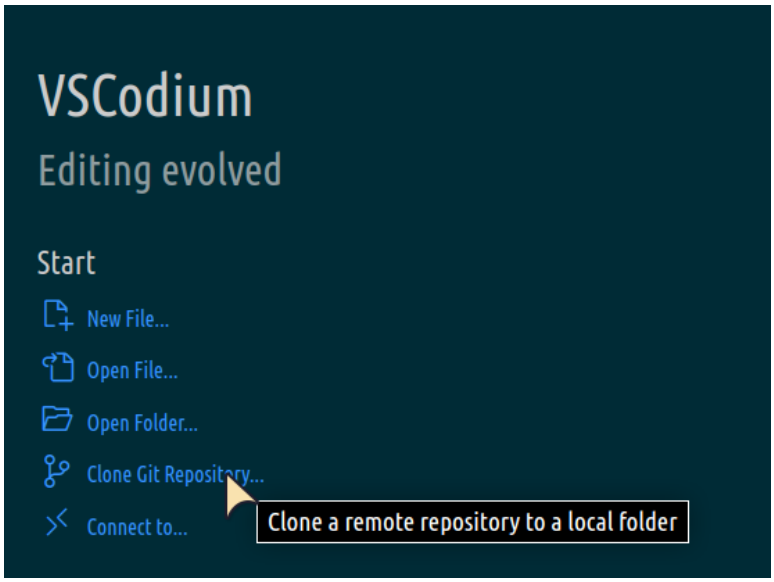
1.5 Reference: Using Git and VS Code

1.5.1 First time setup: Cloning your repository

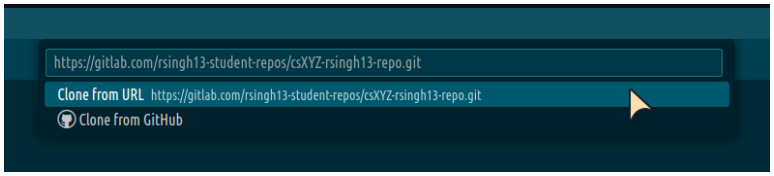
- After you tell the instructor your GitLab name they will give you access to a personal repository for your classwork.



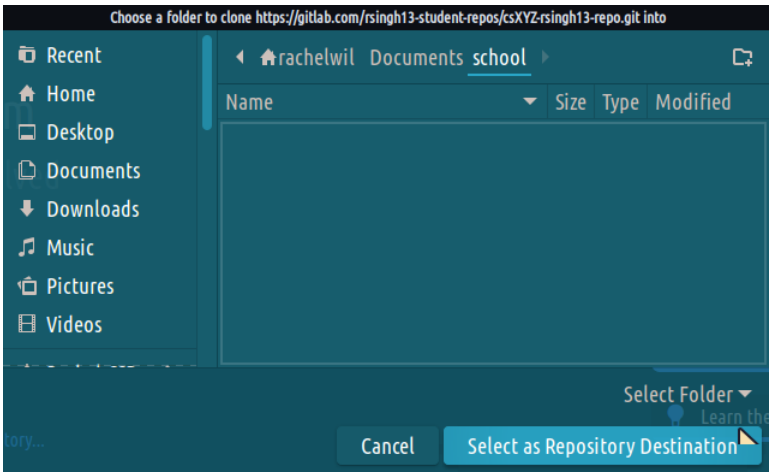
  - From your repository GitLab webpage, click the blue "Code" button, then copy the "HTTPS" link. (If you know about SSH keys, go for it.)



  - 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.



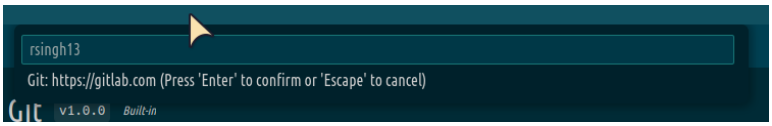
  - Paste your repo URL in this box and hit ENTER.

- 

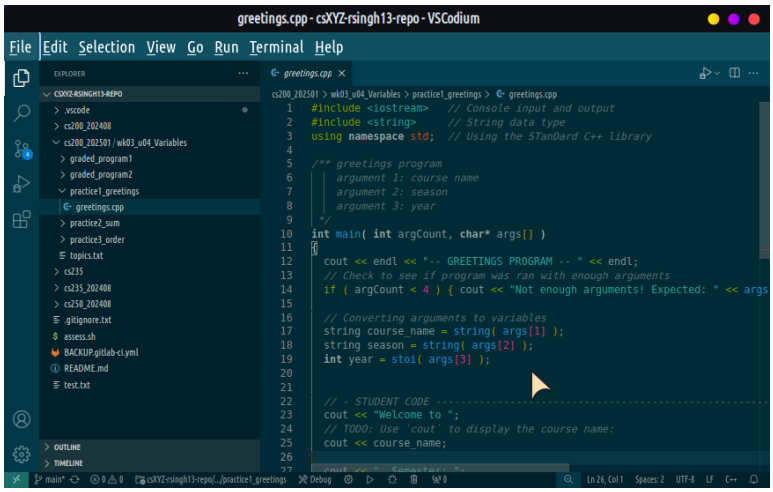
  - It will ask you where you want to store the repository folder on your hard drive. Find a location that you won't forget. :)

- 

  - 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.



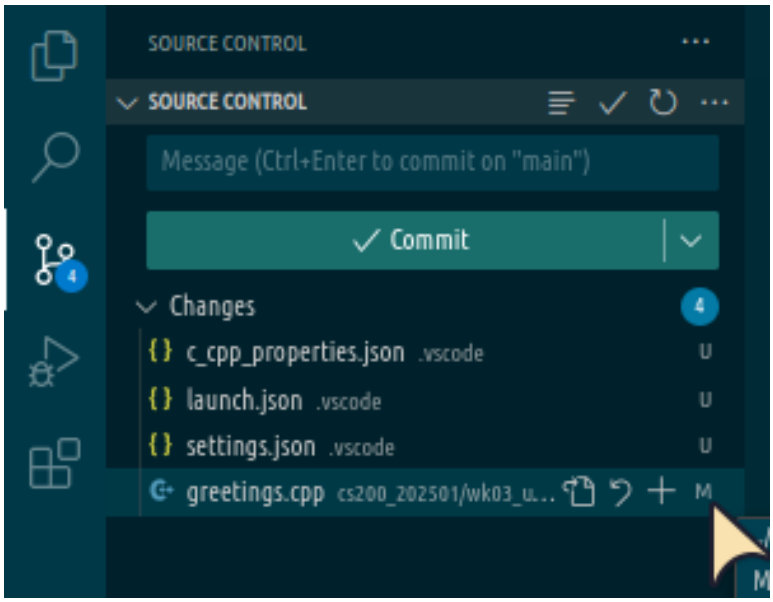
- Once cloning is done, you can open the FOLDER for your repo and see any files within it, such as lab starter code.

1. Configuring Git

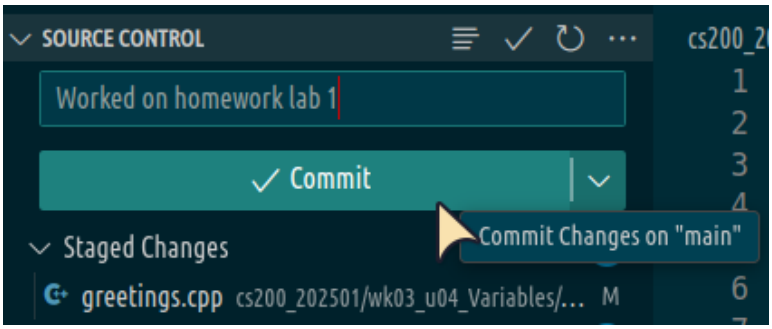
   You can run these commands either in VS Code or in Git Bash, but you'll need to enter a few commands to get everything set up.

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

   git config --global user.email "YOUREMAIL"

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

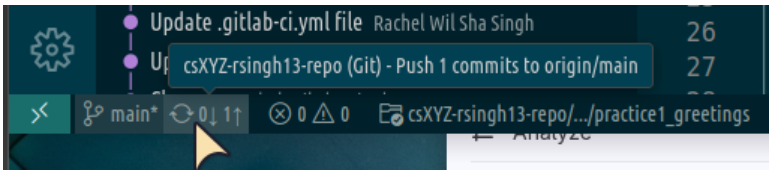
1.5.2 Making changes and backing them up



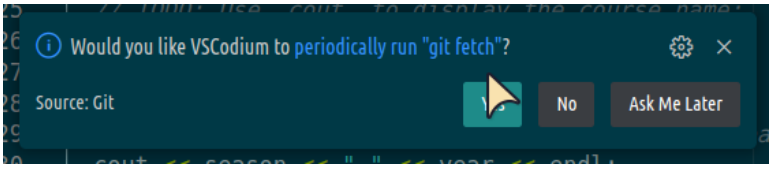
- After you've made a chance to one or more files, it will show up in the list of Changes under the Source Control button.
- Next to a file you want to backup, press the "+" button. It will then be categorized under "Staged Changes".

- 

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

- 

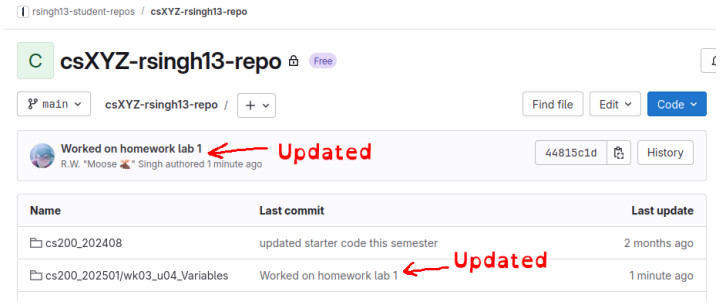
- To send your changes to the server for backup, click on the circular arrow icon at the bottom of VS Code. This will send your changes to the server and pull any changes (e.g., from the instructor) to your computer.

- 

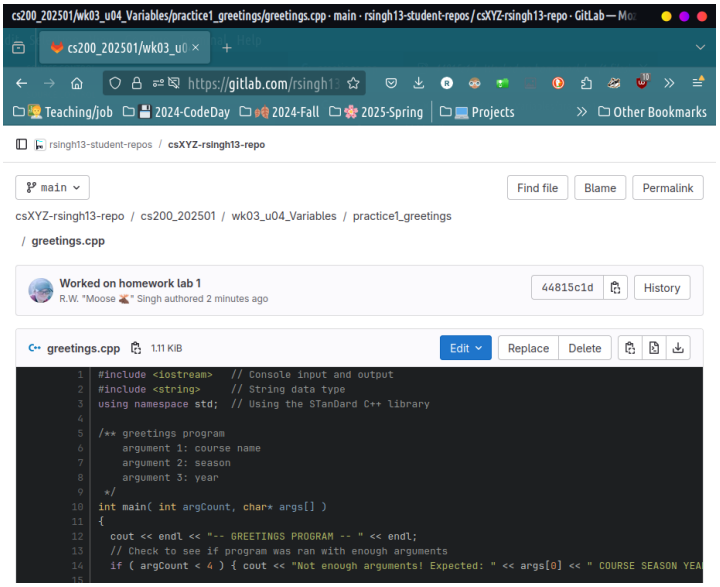
- It might ask if you want to periodically run "git fetch". This command pulls changes from the server. You can select "Yes".

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.

1.6 Intro: Program arguments

1.6.1 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];
   }
   ```

1.7 Intro: CS 200 review

1.7.1 Includes

Include	Features
`#include <iostream>`	Use of `cout` (console output), `cin` (console input), `getline`
`#include <string>`	Use of the `string` data type and its functions
`#include <fstream>`	Use of `ofstream` (output file stream), `ifstream` (input file stream), and related functions
`#include <array>`	Use of `array` from the Standard Template Library
`#include <vector>`	Use of `vector` from the Standard Template Library
`#include <cstdlib>`	C standard libraries, usually used for `rand()`.
`#include <ctime>`	C time libraries, usually used for `srand( time( NULL ) );` to seed random # generator
`#include <cmath>`	C math libraries, such as `sqrt`, trig functions, etc.
`#include "file.h"`	Use "" to include .h files within your own project. DON'T USE INCLUDE ON .cpp FILES!!

About the `using` command The `using namespace std;` command states that we're using the `std` namespace. If this is left off, then we need to prefix any C++ types and functions with `std::`, such as

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

Best practices:

- If your program ONLY uses C++ standard library includes, use `using namespace std;` for brevity.
- If your program uses MULTIPLE LIBRARIES, avoid `using namespace std;` and prefix each type/function with the library it belongs to (e.g., `std::string` from the STD library, `sf::vector2f` from the SFML library)

1.7.2 Bare minimum C++ programs

- **Without arguments:**

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

int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

- **With arguments:**

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

int main( int argCount, char* args[] )
{
  if ( argCount < 4 )
  {
    cout << "Not enough arguments!"
         << endl; return 1;
   }

  int my_int = stoi( args[1] );
  float my_float = stof( args[2] );
  string my_string = string( args[3] );

  return 0;
}
```

1.7.3 Building and running from the command line

- **Building a single source file:**

  ```
  g++ SOURCEFILE.cpp -o PROGRAMNAME.exe
  ```

- **Building multiple source files:**

  ```
  g++ *.cpp *.h -o PROGRAMNAME.exe
  ```

- **Running a program without arguments:**

  ```
  ./PROGRAMNAME.exe
  ```

- **Running a program with arguments:**

  ```
  ./PROGRAMNAME.exe arg1 arg2 arg3
  ```

Note that Linux and Mac usually use ".out" instead of ".exe".

1.7.4 Variable declaration

- Declaring a variable:

  ```
  DATATYPE VARIABLENAME;
  ```

- Declaring a variable and assigning a value:

  ```
  DATATYPE VARIABLENAME = VALUE;
  ```

- Declaring and assigning a named constant:
  `const DATATYPE NAME = VALUE;`
- Assigning a new value to an existing variable:
  `VARIABLENAME = VALUE;`
- Copying a value from one variable to another:
  `UPDATEDVAR = COPYME;`
- Data types and values:

Data type	Value examples
int	`-5`, `0`, `100`
float	`0.99`, `-3.25`, `1.0`
string	`"Hello world!"`
char	`'a'`, `'$'`
bool	`true`, `false`

- Increment/Decrement statements:
  - `a++`, `a--`
  - `++a`, `--a`
  - `a+=5;`, `a-=5;`
  - `a = a + 5;`, `a = a - 5;`

Notes:

- `LHS = RHS;` copies FROM `RHS` TO `LHS`; make sure you have the right order!
- A hard-coded value, like `"Hello"`, is known as a **literal**.

1.7.5 Console input and output

- Output a variable's value:

```
cout << VARIABLENAME;
```

- Output a string literal:

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

- Chain together multiple items:

```
cout << "Label: " << VARIABLENAME << endl;
```

- Input to a variable:

```
cin >> VARIABLENAME;
```

- Input a whole line to a string variable:

`getline( cin, STRINGVARIABLE );`

Notes:

- The `<<` operator is called the **output stream operator** and is used on `cout` statements.
- The `>>` operator is called the **input stream operator** and is used on `cin` statements.
- `endl` is only to be used with `cout` statements, not `cin`!
- `getline` can only be used with **strings**!! Use `cin >>` for other data types.
- You need a `cin.ignore()` ONLY in between

`cin >> ...`

and

`getline( cin, ... )`

- If your program is **skipping an input** then you're missing a `cin.ignore();`.
- If your program is getting input but **not storing the first letter** then you have too many `cin.ignore();` statements / they're in the wrong place.

1.7.6 Boolean expressions

- AND (`&&`)
  - Expression is TRUE if all sub-expressions are TRUE
  - Expression is FALSE if at least one sub-expression is FALSE

a	b	a AND b
T	T	T
T	F	F
F	T	F
F	F	F

- OR (`||`)
  - Expression is TRUE if at least one sub-expression is TRUE
  - Expression is FALSE if all sub-expressions are FALSE

a	b	a OR b
T	T	T
T	F	T
F	T	T
F	F	F

- NOT (!)
  - Expression is TRUE if sub-expression is FALSE
  - Expression is FALSE if sub-expression is TRUE

a	NOT a
T	F
F	T

1.7.7 If statements

IF STATEMENT:

```
if ( CONDITION_A )
{
  // Action A
}
```

IF/ELSE STATEMENT:

```
if ( CONDITION_A )
{
  // Action A
}
else
{
  // Action Z
}
```

IF/ELSE IF STATEMENT:

```
if ( CONDITION_A )
{
  // Action A
}
else if ( CONDITION_B )
{
  // Action B
}
else if ( CONDITION_C )
{
  // Action C
}
```

IF/ELSE IF/ELSE STATEMENT:

```
if ( CONDITION_A )
{
  // Action A
}
else if ( CONDITION_B )
{
  // Action B
}
else if ( CONDITION_C )
{
  // Action C
}
else
{
  // Action Z
}
```

- Condition types can be a boolean variable or a boolean expression:

```
bool done = true;
int a = 10, b = 5;
if ( done ) // variable
```

```
{
  cout << "Goodbye!" << endl;
}
if ( a > b ) // expression
{
  cout << "a is bigger!" << endl;
}
```

- `else` statements NEVER TAKE A CONDITION
- Whichever **condition** evaluates to `true`, all future else if and else statements are skipped.

1.7.8 Switch statements

```
switch( myNumber )
{
  case 1:
    cout << "It's one!" << endl;
    break;

  case 2:
    cout << "It's two!" << endl;
    break;

  default:
    cout << "I don't know what it is!" << endl;
}
```

- You can leave off `break;` from a case statement. In this case, **fallthrough** occurs, where the following case's code will be executed, up until it hits a `break;` statement.
- Switch statements like the one above can be replaced with "if myNumber equals 1" and "else if myNumber equals 2".
- In C++, `switch` doesn't work with `string`, only primitive data types like `int`, `float`, `char`.
- Fallthrough example:

```
char choice;
cout << "Enter a choice: (y/n): ";
cin >> choice;

switch( choice )
{
  case 'y':
  case 'Y':
    cout << "YES" << endl;
```

```
      break;

    case 'n':
    case 'N':
      cout << "NO" << endl;
      break;

    default:
      cout << "INVALID INPUT!" << endl;
  }
```

1.7.9 While loops

`while ( CONDITION ) { }`

- While loops use CONDITIONS like if statements do.
- While loops are susceptible to **infinite loop** errors if nothing within the loop causes the CONDITION to evaluate to false eventually. Be careful!
- Example:

```
while ( a < b )
{
  cout << a << endl;
  a++;
}
```

1.7.10 For loops

Normal for loop: `for ( INIT; CONDITION; UPDATE ) { }`

- Example:

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

- When using STL arrays or vectors, `size_t` or `unsigned int` should be used instead of `int` for `i`. This will remove the warning relating to testing `.size()` (which returns `size_t`) against a signed integer.

Range-based for loop: `for ( INIT : RANGE ) { }`

- In versions of C++ past C++98 (from 1998) you can use range-based for loops to iterate over a range of items:
- Example:

```
vector<int> myVec = { 1, 2, 3, 4 };
for ( int element : myVec )
{
  cout << element << endl;
}
```

1.7.11 Arrays and vectors

- Declare a C-style array:

  `DATATYPE ARRAYNAME[ SIZE ];`

- Declare a C++ STL array:

  `array<DATATYPE, SIZE> ARRAYNAME;`

- Declare a C++ STL vector:

  `vector<DATATYPE> VECTORNAME;`

- Declare a dynamic array via pointer:

  `DATATYPE * PTR = NEW DATATYPE[ SIZE ];`

- Destroy a dynamic array:

  `delete [] PTR;`

- Initialize an array with an initializer list:

  `DATATYPE ARRAY[] = { VAL1, VAL2, VAL3 }`

  (also works for `array` and `vector`).

- The **position** of an item within an array is called its **index**.

- The variable within the array at some position is called its **element**.

- A `vector` is a dynamic array, but you don't have to worry about the memory management. :) This means that it can be resized.

- A `array` is a fixed size, just like the C-style array.

- Example: Iterate over an array/vector

```
// C-style array:
for ( int i = 0; i < TOTAL_STUDENTS; i++ )
{
  cout << "index: " << i << ", value: " << students[i] << endl;
}

// STL Array and STL Vector:
for ( size_t i = 0; i < bankBalances.size(); i++ )
{
  cout << "index: " << i << ", value: " << students[i] << endl;
}
```

(`size_t` is another name for an `unsigned int`, which allows values of 0 and above - no negatives.)

1.7.12 File I/O

- `#include <fstream>` is required to use `ifstream` (input file stream) and `ofstream` (output file stream) types.
- An `ifstream` open will fail if the file requested does not exist or cannot be found. Use `if ( input.fail() )` to check for a failure scenario.
- An `ofstream` open will create a file if it doesn't already exist.
- The default path where files are written to or read from is **wherever your project file is** (.vcxproj for Visual Studio, .cbp for Code::Blocks). You can also set a **default working directory** in your project settings.
- Example: Create an output file and write

```
ofstream output;
output.open( "file.txt" );

// Write to text file
output << "Hello, world!" << endl;
```

- Example: Create an input file and read

```
ifstream input;
input.open( "file.txt" );
if ( input.fail() )
{
  cout << "ERROR: could not load file.txt!" << endl;
}
string buffer;

// read a word
input >> buffer;

// read a line
getline( input, buffer );
```

1.7.13 Functions

1. **Function headers**

   A function **header** contains the following information:

   `RETURNTYPE FUNCTIONNAME( PARAMETERLIST )`

   - The **return type** is the type of data returned (like an `int`), or `void` for functions that don't need to return any data.
   - The **function name** follows the same naming rules as a variable - letters, numbers, and underscores allowed, no spaces or other special characters.
   - The **parameter list** is a series of variable declarations to be used within the function. These parameters are assigned values during the *function call*, when **arguments** are provided.

2. **Function declarations** Function **declarations** should go in .h files. A function declaration is the function header, with a semicolon at the end:

   ```
   int Sum( int a, int b );
   void DisplayMenu();
   ```

3. **Function definitions** Function **definitions** should go in .cpp files. A function definition is the function header, plus a code block, starting and ending with curly braces `{}`:

   ```
   int Sum( int a, int b )
   {
     int result = a + b;
     return result;
   }
   ```

4. **Function calls** Function **calls** will happen within other functions. A function call includes the function's name, input arguments to be passed in, and the return data will need to be stored in a variable, if applicable.

   ```
   int num1, num2, result;
   cout << "Enter two numbers, separated by a space: ";
   cin >> num1 >> num2;
   result = Sum( num1, num2 ); // Function call
   cout << "Result: " << result << endl;
   ```

5. **Header files**

   - Function declarations should go in **header files** (.h files).
   - **Header files must have file guards**, this prevents the .h file from being "copied" into multiple files, which will cause a "duplicate code" build error.
   - For the file guard, a label like `_FILENAME_H` is used. **This must be unique for each file!**

- Visual Studio supports using `#pragma once` for .h files, but this is **not cross platform**, so you should use these preprocessor file guards!
- Example file guard for a .h file:

```
#ifndef _FILENAME_H
#define _FILENAME_H

// Code goes here

#endif
```

6. **Source files**

   - Function definitions should go in **source files** (.cpp files). **File guards are not put in .cpp files.**

7. **Using your functions in other files**

   - Any file that utilizes your function **must include the .h file**:

     ```
     #include "MyFunctions.h"
     ```

   - Note that something like `#include <iostream>` is used for including libraries that are not *inside the project*. Using `""` is for including files that are in your project.
   - **NEVER INCLUDE .cpp FILES, THIS WILL GENERATE ERRORS!**

8. **Function overloading**

   - Multiple functions can have the same **name** as long as their function signatures are different. This means that two functions with the same name either need a *different number of parameters*, or *different parameter data types*, so that they can be unambiguously identified.
   - **AMBIGUOUS:**

     ```
     void MyFunction( int a, int b );
     void MyFunction( int num1, int num2 );
     ```

   - **UNAMBIGUOUS:**

     ```
     void MyFunction( int num1, int num2 );
     void MyFunction( string name1, string name2 );
     void MyFunction( int oneNum );
     ```

1.7.14 Structs and classes

1. **Accessibility levels**

   - We can specify accessibility levels for struct and class members. This information dictates where a struct/class' internal contents can be accessed from:

Accessibility level	Class' functions?	Child's functions?	External functions?
`private`	Yes	NO	NO
`protected`	Yes	Yes	NO
`public`	Yes	Yes	Yes

   - `private` members can only be accessed from the class' own functions.
   - `protected` members can only be accessed from the class' own functions, and also the functions of any other classes that INHERIT from this class.
   - `public` members can be accessed from anywhere in the program, including functions that are not a part of the class.

2. **Structs**

   - Structs are usually used to store very basic structures, often with just variable data and no functions.

```
struct Coordinate
{
  float x, y;
};
```

   - Struct declarations should go in their own .h file, usually the filename will match the name of the class, such as "Coordinate.h".
   - **All .h files need file guards!**
   - Also note that at the closing curly brace `}` of the struct declaration there is a semicolon - this is required!
   - Members of a struct are `public` level accessibility by default.

3. **Classes**

   - Classes are meant for more complex structures.

```
class CLASSNAME
{
    public:
    // Public members

    protected:
    // Protected members

    private:
    // Private members
};
```

- **Class declarations should go in their own .h file, and class function definitions should go in a corresponding .cpp file!**
- It is best practice to make **member variables** of a class `private`, and only provide indirect access to this data via functions.
- **Example: Product.h**

```
#ifndef _PRODUCT_H  // File guards
#define _PRODUCT_H  // File guards

#include <string>
using namespace std;

class Product
{
    public:
    // Constructors:
    Product();
    Product( string newName, float newPrice );

    // Destructor:
    ~Product();

    // Setters:
    void SetName( string newName );
    void SetPrice( float newPrice );

    // Getters:
    string GetName() const;
    float GetPrice() const;

    private:
    string m_name;
    float m_price;
}

#endif
```

- **Example: Product.cpp**

```
#include "Product.h"

// Constructors:
Product::Product()
{
  m_name = "unset";
  m_price = 0;
}

Product::Product( string newName, float newPrice )
{
  SetName( newName );
```

```
  SetPrice( newPrice );
}

// Destructor:
Product::~Product()
{
  cout << "Bye." << endl;
}

// Setters:
void Product::SetName( string newName )
{
  m_name = newName;
}

void Product::SetPrice( float newPrice )
{
  if ( newPrice >= 0 )
  {
    m_price = newPrice;
  }
}

// Getters:
string Product::GetName() const
{
  return m_name;
}

float Product::GetPrice() const
{
  return m_price;
}
```

- **Accessor/Getter functions:**
  - Don't take in any input data (no parameters).
  - Return the value of a private member variable (has a return).
  - Should be marked `const` to prevent data from changing within the function. (This is "read only")
- **Mutator/Setter functions:**
  - Take in an input value of the new data to be stored (has a parameter).
  - Generally doesn't return any data (no return, `void` return type).
- **Constructor functions:**
  - Called automatically when a new object of that class type is created.
  - Can overload.
- **Destructor functions:**

- Called automatically when an object of that class type is destroyed/loses scope.
- Cannot overload.

4. **Class objects / instantiating a class object**

   - Once we've declared a class we can then declare variables whose data types *are that class*:

     ```
     PlayerCharacter bario;
     NonPlayerCharacter boomba;
     ```

   - In this example, `bario` is a **PlayerCharacter object**, aka an "instantiation of the PlayerCharacter object".

5. **Class inheritance**

   - A class (called a **subclass** or a **child class**) can inherit from another class (called a **superclass** or a **parent class**). Doing this means that any `protected` and `public` members are *inherited* by the child class. This can be useful for creating more "specialized" versions of something, storing the shared attributes of a set of items in a common "parent" class.
   - **Example:**

     ```
     class Character
     {
       void SetPosition( float x, float y );
       void Move( float xAmount, float yAmount );

       protected:
       float x, y;
     };

     class PlayerCharacter : Public Character
     {
       public:
       void GetKeyboardInput();

       private:
       int totalLives;
     };

     class NonPlayerCharacter : public Character
     {
       public:
       void ComputerDecideMove();

       private:
       bool attackPlayer;
     };
     ```

6. **Class composition**

   - Class composition is where a class contains *class objects* as member variables. This is another form of object oriented design where a class might "contain" traits that could be inherited, but instead encapsulates them into a sub-object.
   - **Example:**

```
class MovableObject
{
  void SetPosition( float x, float y );
  void Move( float xAmount, float yAmount );

  private:
  float x, y;
};

class PlayerCharacter
{
  public:
  void GetKeyboardInput();

  private:
  int totalLives;
  MovableObject mover;
};
```

1.8 Lab: CS 200 review

1.8.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

1.8.2 Included files:

```
wk01_CS200Review
 graded_program
    image1.txt
    image2.txt
    image3.txt
    Image.cpp
    Image.h
    main.cpp
 instructions.org
 practice1_basics
    welcome.cpp
 practice2_branching
    battery.cpp
 practice3_whileloops
    numguess.cpp
 practice4_forloops
    forloops.cpp
 practice5_vectors
    stlvector.cpp
```

```
practice6_functions
   funcprog.cpp
practice7_classes
   Ingredient.cpp
   Ingredient.h
   instructions.md
   main.cpp
practice8_fileio
    file1.txt
    saveload.cpp
```

1.8.3 Practice programs

Within your repository folder in the `wk01_CS200Review` folder you'll see a set of practice programs to work on. Their instructions are either in the code itself as a comment, or as a separate `instructions.md` file.

1.8.4 Graded program - Image loader

1. Instructions

   - Start with the Image.h file and follow the `// TODO` items.
   - Work on Image.cpp second.
   - Work on main.cpp third.
   - If you have trouble we can go over it during class.
   - Build the program with: `g++ *.h *.cpp`
   - Run the program with:
     - `./a.out FILENAME` (Mac/Linux)
     - `./a.exe FILENAME` (Windows)
     - Pass in either `image1.txt`, `image2.txt`, or `image3.txt` to see a different image.

2. Example output:

   share/student-repo/wk01_CS200Review/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs235-cs250-share/student-repo/wk01_CS200Review/screenshot.png?ref_type=heads

   A screenshot of the program running three times, taking in a different input file each time.

---

2 Week 2: Testing, debugging, friends, and templates

2.1 Intro: Testing

2.1.1 How do we test?

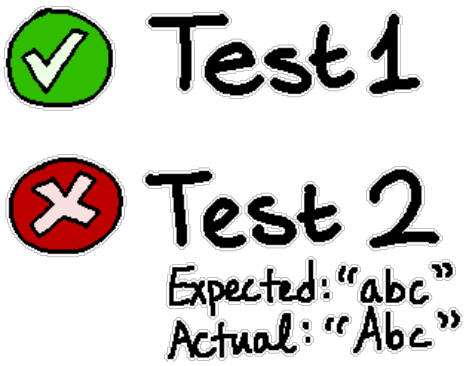
How do you know that your program actually works? Especially as it gets more complex?

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**?"

~

2.1.2 Writing tests first?

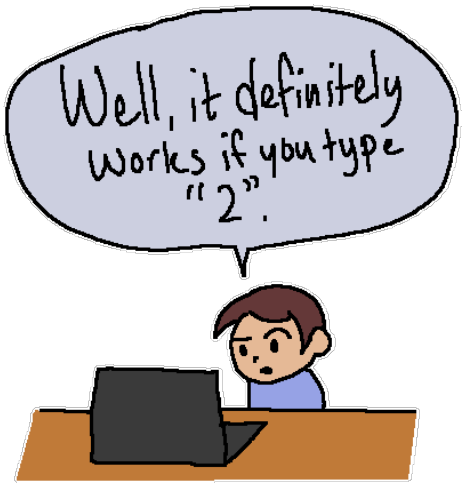


Notice that you don't actually need to know *what* is in the *function* before you write tests for it. Knowing what the program is *supposed to do*, you can figure out a set of inputs and outputs that it ought to have.

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).

~

2.1.3 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!

~

2.1.4 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**.)

~

2.1.5 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.

~

2.1.6 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.

2.2 Intro: General debugging

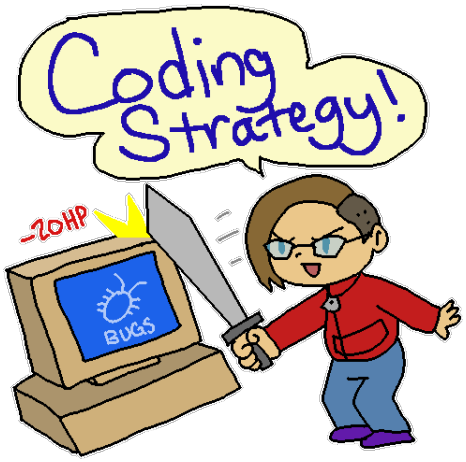
Being able to read and interpret error messages is really important in programming. Especially when just starting off, you will be making a lot of **syntax errors** until you learn the language better.

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!

~

2.2.1 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`.

2.2.2 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";
     ```

2.2.3 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;

2.2.4 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 »

~

2.2.5 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.

2.2.6 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?

2.3 Intro: Debugging with gdb

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

2.3.1 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
```

2.3.2 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>`

2.3.3 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::operato
```

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::operator<< <c
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 = {...}) at prog
#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`.

2.3.4 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
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
27      {
    (gdb) n
    28          if ( argCount != 2 )
    (gdb) n
    30              cout << endl << "Expected form: " << args[0] << " zipcode" << endl;
(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 |
(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#`.

2.3.5 Leaving gdb

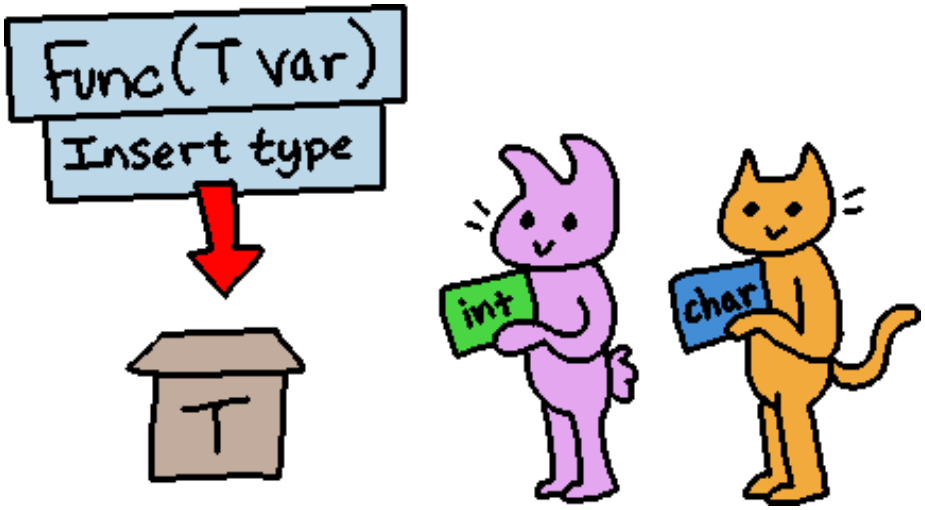
You can exit gdb by pressing CTRL+D or typing `exit`.

2.3.6 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).

2.4 Intro: Templates



2.4.1 Before Templates

Templates don't exist in C++'s precursor, C. Because of this, if you had a function like - for example - `SumTwoNumbers` - that you wanted to work with different data types, you would have to define different functions for each version. C also doesn't have **function overloading**, so they would have to have different names as well.

As a real-world example, OpenGL is a cross-platform graphics library that can be used to create 3D graphics. OpenGL was written in C, and you could tell it a set of vertices to draw in order to create one polygon or quad or other shape. There were different functions you could use to define points (vertices) in a shape, like:

- `glVertex2f( 0, 0 );`
- `glVertex3f( 0, 0, 0 );`

And, in particular, there are a bunch of "glVertex" functions: glVertex2d, glVertex2dv, glVertex2f, glVertex2fv, glVertex2i, and so on... (Don't you wish you were programming in C?)

2.4.2 What are Templates?

With C++ and other languages like C# and Java, we can now use **Templates** with our functions and classes. A Template allows us to specify a **placeholder** for a data type which will be filled in later.

In the C++ Standard Template Library, there are objects like the **vector** that is essentially a dynamic array, but it can store any data type - we just have to tell it what it's storing when we declare a vector object:

C++ source: Declaring templated vectors

```
vector<int> listOfQuantities;
vector<float> listOfPrices;
vector<string> listOfNames;
```

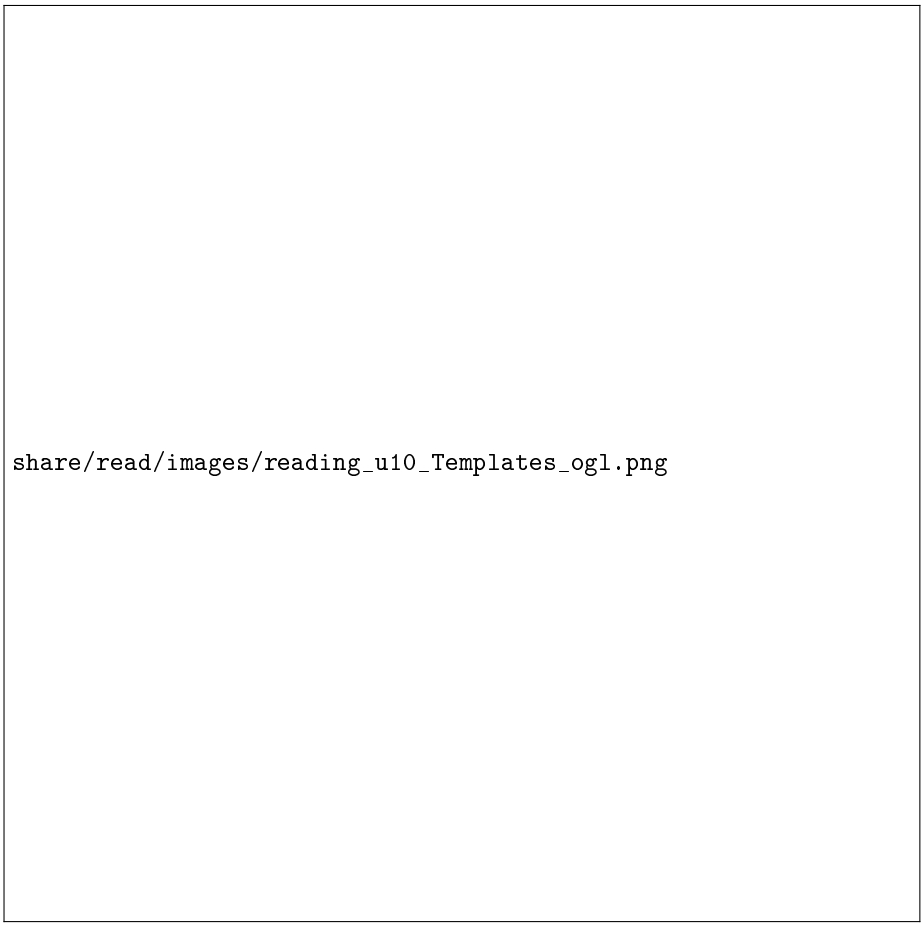


Figure 1: A rendering of a simple 3D scene built from triangles and quads, rendered with OpenGL.

We can also define our own functions and even classes with templated functions and member variables ourselves, leading to much more reusable code.

1. Templated functions

   We can write a standalone function with templated parameters or a templated return type or both. For example, here's a simple function to add two items together:

   C++ source: Templated function form

```
template <typename T>
T Sum( T numA, T numB )
{
  return numA + numB;
}
```

   This function can be called with **any data type**, so long as the data type has the `+` operator defined for it - so, if it were a custom class you wrote, you would have to overload the `operator+` function.

   What this means is that we can call `Sum` with integers and floats, but also with someting like a string, since strings use the + operator to combine two strings together.

   C++ source: Calling the templated function

```
int main()
{
  int intA = 4, intB = 6;
  float floatA = 3.9, floatB = 2.5;
  string strA = "alpha", strB = "bet";

  cout << intA << " + " << intB
       << " = " << Sum( intA, intB ) << endl;

  cout << floatA << " + " << floatB
       << " = " << Sum( floatA, floatB ) << endl;

  cout << strA << " + " << strB
       << " = " << Sum( strA, strB ) << endl;
}
```

   Program output:

```
4 + 6 = 10
3.9 + 2.5 = 6.4
alpha + bet = alphabet
```

2. Templated classes

   More frequently, you will be using templates to create classes for data structures that can store **any kind of data**. The C++ Standard Template

Library has data structures like **vector**, **list**, and **map**, but we can also write our own.

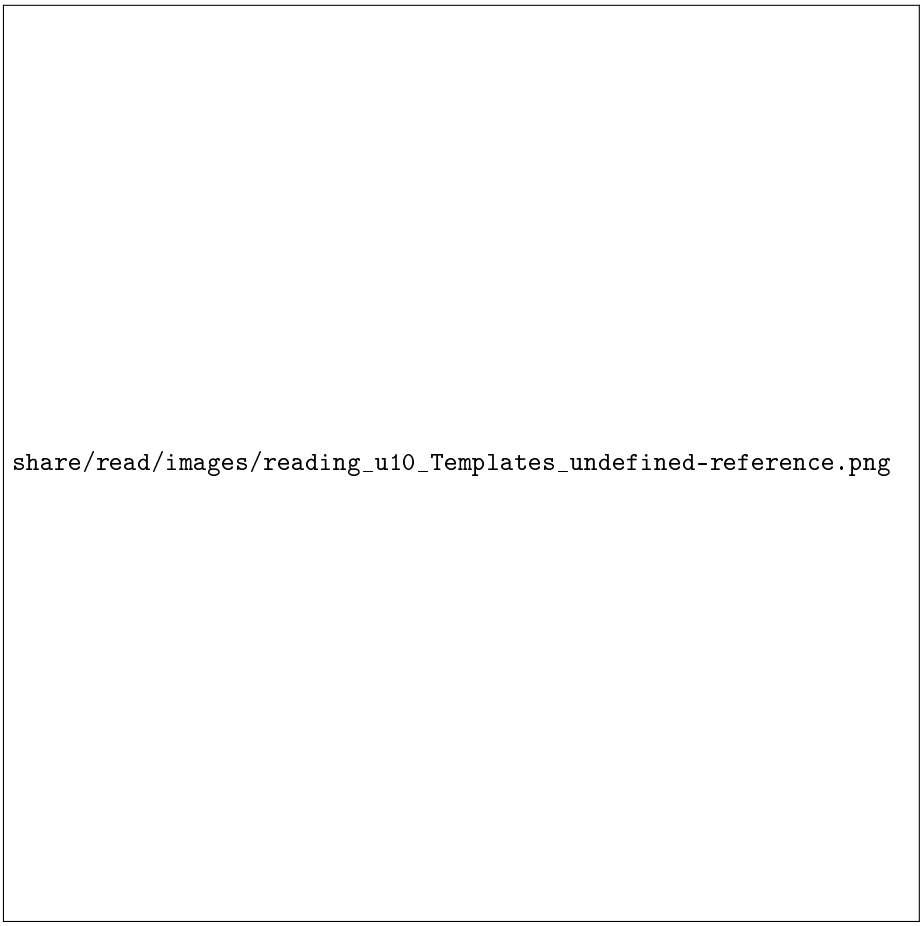
When creating our templated class, there are a few things to keep in mind:

(a) We need to use `template <typename T>` at the beginning of the class declaration.

(b) Method definitions **must be in the header file** - in this case, we won't be putting the method definitions in a separate .cpp file. You can either define the functions *inside* the class declaration, or immediately after it.

(c) Method definitions also need to be prefixed with `template <typename T>`.

If you try to create a "TemplatedArray.h" file and a "TemplatedArray.cpp" file and put your method definitions in the .cpp file, then you're going to get compile errors:



You might think, "Well, that's weird." - and yes, it is. C++ is a strange language with weird behaviors. In this case in particular, you can read

about why this is for templates here: https://isocpp.org/wiki/faq/templates\#templates-defn-vs-decl

In short, the template command is used to generate classes, and while our class declaration looks normal, this is actually special code that is just telling the compiler how it's going to generate a family of classes. Because of this, the compiler needs to see the function definitions as well.

2.4.3 Example templated class:

Here's a small example of a templated class declaration.

C++ source: Class declaration (.h file):

```
template <typename T>
class CLASSNAME
{
  public:
  void Setup( T PARAM1, int PARAM2 );

  private:
  T VARNAME1;
  int VARNAME2;
};
```

If the class uses a templated type somewhere within it, it needs to have the `template <typename T>` specified above the class name. Some templated classes may have member variables with `T` as their data types, but this isn't a requirement.

C++ source: Function definitions (also in .h file)

```
template <typename T>
void CLASSNAME<T>::Setup( T PARAM1, int PARAM2 )
{
  this->VARNAME1 = PARAM1;
  this->VARNAME2 = PARAM2;
}
```

The templated functions' definitions must also go in the header (.h) file. This is how the definition would look if you're defining it *beneath* the class declaration.

C++ source: Instantiating the templated class (main.cpp example)

```
int main()
{
  CLASSNAME<string> obj1; // PARAM1 will be a string
  CLASSNAME<float> obj2; // PARAM1 will be a float
```

```
  // Calling the templated function:
  obj1.Setup( "Test", 100 );
  obj2.Setup( 2.99, 50 );

  return 0;
}
```

When declaring an object variable whose data type is a templated class you need to specify the **data type** to fill into the `T` placeholder. This part goes within angle brackets `< >`.

2.5 Intro: Friends

Remember that when member variables and functions of a class are set to **public** they can be accessed by anything and when they are set to **private** these members can *only* be accessed from within the class itself.

We can make an exception to this rule by declaring some external function or other class as a **friend** of the class we're creating. A friend function or class has access to any private or protected members of the class.

~

2.5.1 Friend function:

The friend function will be declared within the class' declaration:

```
class MyClass
{
  public:
  void Hi();

  private:
  int name;

  friend void PrintMyClass( const MyClass& item );
};
```

~

And then it can be defined in a source (.cpp) file elseware:

~

```
void PrintMyClass( const MyClass& item )
{
  // name is private, but this function can access it.
  cout << item.name << endl;
}
```

~

2.5.2 Friend class:

A friend class is the same sort of thing except that any member function of our friend class has access to any private members of the other class.

```
class ClassWithAFriend
{
  private:
  int name;

  friend class FriendlyClass;
};
```

~

That other class would be declared elseware, and any functions it has can access our `ClassWithAFriend`'s members.

~

```
class FriendlyClass
{
public:
    void Display( const ClassWithAFriend& myFriend )
    {
        cout << myFriend.name << endl;
    }
};
```

~

However - it doesn't go both ways.

Keep in mind that if `classA` declares that `classB` is its friend, this means that `classB` has access to `classA`'s members. However, this **does not** mean that `classA` has access to `classB`'s members - we would have to explicitly state "classA is a friend" within the classB class.

2.5.3 Application of friends

Usually the best design is to keep member variables private - no exceptions. By exposing member data externally, that means the data can be accessed or changed without error checks, and updates in the codebase will be more difficult.

I tend to ONLY use the `friend` feature for one of two things:

1. A **tester** class that contains unit tests for a given class; the tester will have access to the private member variables to confirm that proper changes are made after a function call.

2. A **manager** class that handles the class it is a friend of, so it can manage everything about that class. However, design-wise, a manager class can still be designed without "friend" status just fine using the class' Get and Set functions.

2.6 Lab: Testing, Debugging, Friends, and Templates

2.6.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

2.6.2 Included files:

```
wk02_TestDebugFriendTemplate
 graded_program
    FixedArray.h
    FixedArrayTester.cpp
    FixedArrayTester.h
    main.cpp
 instructions.org
 practice1_testing
    Functions.cpp
    Functions.h
    main.cpp
 practice2_debugging
    crash.cpp
    debug-questions.txt
    logicerror.cpp
 practice3_friends
    main.cpp
 practice4_templates
     Functions.h
```

```
Products.cpp
Products.h
main.cpp
```

2.6.3 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 - Testing

   Within the **Functions.h** file you'll see the following functions declared, as well as a "Test" function for each:

   - `bool IsOverdrawn( float balance );`
   - `float AdjustIngredient( float original, float batches );`
   - `bool IsValidInput( int choice, int min, int max );`
   - `float Average( vector<float> arr );`
   - `int Factorial( int n );`

   The functions themselves aren't implemented, and each of the test functions are incomplete. Only having *one test* generally isn't enough to check all reasonable outcomes for each of these functions. For example, if you had a "add X and Y" tester, only checking "2 + 3 = 5?" isn't enough, because someone might have programmed the function to just have `return 5;`.

   Before fixing the functions themselves, implement the tests. This is part of **Test Driven Development:** tests come first, to help you verify that the to-be-implemented functionality works, and once those tests are up and running (and probably failing), then you build out the function and use the tests to verify your work.
   ~

   (a) IsOverdrawn:
   `bool IsOverdrawn( float balance )`
   - Parameter: `float balance`, a bank account balance.
   - Return: `bool`, true if overdrawn, false otherwise.

   Here we need three test cases to get complete coverage: One for overdrawn, but also checking a positive value and zero (zero isn't overdrawn, just empty). One test is given:

```
{
    float input_balance = 10;
    bool  exp_out = false;
    bool  act_out = IsOverdrawn( input_balance );
    cout << "Test: IsOverdrawn(" << input_balance << ") = " << exp_out << "? ";
    if ( act_out == exp_out ) { cout << GREEN << " PASS" << endl; }
    else                      { cout << RED << " FAIL; got " << act_out << " instead!"
    cout << CLEAR;
}
```

Use this as reference and implement two additional tests...

Input	Expected output
0	false (not overdrawn)
NEGATIVE NUMBER	true (overdrawn)

Build and run your program and run the tests. Some will fail:

```
$ g++ *.h *.cpp
$ ./a.out test

Test: IsOverdrawn(10) = 0?  PASS
Test: IsOverdrawn(0) = 0?  PASS
Test: IsOverdrawn(-5) = 1?  FAIL; got 0 instead!
```

Finally, fix up the `IsOverdrawn` function to have the correct functionality, then build and run tests again:

```
$ g++ *.h *.cpp
$ ./a.out test

Test: IsOverdrawn(10) = 0?  PASS
Test: IsOverdrawn(0) = 0?  PASS
Test: IsOverdrawn(-5) = 1?  PASS
```

You can also run the command directly in the command line:

```
$ ./a.out IsOverdrawn 10
IsOverdrawn(10) = false

$ ./a.out IsOverdrawn 0
IsOverdrawn(0) = false

$ ./a.out IsOverdrawn -5
IsOverdrawn(-5) = true
```

~

(b) AdjustIngredient

`float AdjustIngredient( float original, float batches )`

- Parameter: `float original`, the original amount for the ingredient, and `float batches`, the adjusted amount of batches to bake.
- Return: `float`, the adjusted amount.

Again the first test is given, you just need to implement at least one more.

Input - original	Input - batches	Expected output
5	1.5	7.5
?	?	?

Make your test to use different input values and check the output value. The calculation is *original * batches*, punch your numbers into a calculator and the number value is your result. Put this hard-coded number as your test expected output.

Implement the test, then the function, and run the automated tests and also test manually using the AdjustIngredient command.

```
$ ./a.out test
(...)
Test: AdjustIngredient(5, 2) = 10?  PASS
Test: AdjustIngredient(7, 0.5) = 3.5?  PASS
(...)

$ ./a.out AdjustIngredient 5 1.5
AdjustIngredient(5, 1.5) = 7.5

$ ./a.out AdjustIngredient 7 0.5
AdjustIngredient(7, 0.5) = 3.5
~
```

(c) IsValidInput

`bool IsValidInput( int choice, int min, int max )`

- Parameter: `int choice` the user's choice. `int min` the minimum valid value (inclusive). `int max` the maximum valid value (inclusive).
- Return: `true` if choice is within min and max, or `false` otherwise.

Input - choice	Input - min	Input - max	Expected output
5	1	5	true
1	2	6	false

With tests like these, you should make sure to have enough test cases to make sure that the "is equal to" case is also met... so if the minimum is 1 and the choice is 1, make sure that returns true.

(There's a difference between $1 <= x <= 5$ and $1 < x < 5$!)

- NOTE: To test two separate boolean expressions together, use AND `&&` or OR `||`. Each sub-expression should be complete.
  - DOESN'T WORK: if (1 <= x <= 5)
  - DOESN'T WORK: if (x < 1 || > 5)
  - OK: if (x < 1 || x > 5) ... (Not the solution)

```
$ ./a.out test
(...)
Test: IsValidInput(5, 1, 5) = 1?  PASS
Test: IsValidInput(1, 1, 5) = 1?  PASS
Test: IsValidInput(1, 2, 6) = 0?  PASS
Test: IsValidInput(7, 2, 6) = 0?  PASS
(...)
```

```
$ ./a.out IsValidInput 5 1 10
IsValidInput(5, 1, 10) = true

$ ./a.out IsValidInput 10 1 5
IsValidInput(10, 1, 5) = false
~
```

(d) Average

`float Average( vector<float> arr )`

- Parameter: `vector<float> arr`, a dynamic array of floats.
- Return: the average - all elements of the `arr` summed together, then divided by `arr.size()`.

Input - arr	Expected output
1.5, 2.5, 3.5	2.5
5, 3, 6, 7	5.25

For a good test, not only use different values for the array elements, but also use different *amounts* of elements. This way we can ensure that the programmer (pretending it's someone else :) didn't hard-code the array size into their implementation!

```
$ ./a.out test
(...)
Test: Average({ 1.5, 2.5, 3.5 }) = 2.5?  PASS
Test: Average({ 5, 3, 6, 7 }) = 5.25?  PASS
(...)

$ ./a.out Average 1 3 5 7
Average({ 1, 3, 5, 7 }) = 4

$ ./a.out Average 3.0 4.0 3.5 2.5
Average({ 3, 4, 3.5, 2.5 }) = 3.25
~
```

(e) Factorial

`int Factorial( int n )`

- Parameter: `int n`, the n value.
- Return: The result of $n!$

Input - n	Expected output
0	1
1	1
2	2
3	6
4	24

Besides having two tests for something like $3!$ and $4!$, make sure to also check for special cases, like $0!$ being 1!

```
$ ./a.out test
(...)
Test: Factorial(3) = 6?  PASS
Test: Factorial(5) = 120?  PASS
Test: Factorial(0) = 1?  PASS
Test: Factorial(1) = 1?  PASS
(...)

$ ./a.out Factorial 5
Factorial(5) = 120

$ ./a.out Factorial 7
Factorial(7) = 5040
~
```

(f) Example output: No arguments given:

```
$ ./a.out

Expected form:
./a.out test - run all tests
./a.out IsOverdrawn amount - check if amount given is overdrawn
./a.out AdjustIngredient original batches - calculate adjusted ingredient amount
./a.out IsValidInput choice min max - check if choice is within the valid range
./a.out Factorial n - calculate n!
./a.out Average num1 num2 num3 num4... - Calculate the average of all numbers given
```

(g) Example output: Running the automated tests (starter code):

```
$ ./a.out test

Test: IsOverdrawn(10) = 0?  PASS
Test: AdjustIngredient(5, 2) = 10?  FAIL; got 0 instead!
Test: IsValidInput(5, 1, 5) = 1?  FAIL; got 0 instead!
Test: Average({ 1.5, 2.5, 3.5 }) = 2.5?  FAIL; got 0 instead!
Test: Factorial(3) = 6?  FAIL; got 0 instead!
```

(h) Example output: Running the automated tests (functions fixed, tests implemented):
Your tests might have different test values!

```
$ ./a.out test

Test: IsOverdrawn(10) = 0?  PASS
Test: IsOverdrawn(0) = 0?  PASS
Test: IsOverdrawn(-5) = 1?  PASS
Test: AdjustIngredient(5, 2) = 10?  PASS
Test: AdjustIngredient(7, 0.5) = 3.5?  PASS
Test: IsValidInput(5, 1, 5) = 1?  PASS
Test: IsValidInput(1, 1, 5) = 1?  PASS
Test: IsValidInput(1, 2, 6) = 0?  PASS
```

```
Test: IsValidInput(7, 2, 6) = 0?  PASS
Test: Average({ 1.5, 2.5, 3.5 }) = 2.5?  PASS
Test: Average({ 5, 3, 6, 7 }) = 5.25?  PASS
Test: Factorial(3) = 6?  PASS
Test: Factorial(5) = 120?  PASS
Test: Factorial(0) = 1?  PASS
Test: Factorial(1) = 1?  PASS
```

---

2. Practice 2 - Debugging

   Within the `practice2_debugging` folder there are two separate programs. We will see how to use the `gdb` (GNU Debugger) in order to locate errors. (You can also view the reference page here: https://gitlab.com/moosadee/courses/-/blob/main/reference/gdb.org?ref_type=heads.)

   Fill out the `debug-questions.txt` file as you go. You do not need to update the source code files!

   ~

   (a) Using the backtrace with crash.cpp

   - Build the crash.cpp file using `-g` to enable debug symbols:

     `g++ -g crash.cpp -o crash.exe`
   - Windows/Linux: Then load it into gdb:

     `gdb crash.exe`
   - Mac/Linux: Or load it into lldb:

     `lldb crash.exe`
   - Use the `run` command to begin the program execution. It will go until it crashes.
     - GDB view:

       ```
       Dereference pointers! What could POSSIBLY go wrong...?
       0. A
       1. B
       2. C

       Program received signal SIGSEGV, Segmentation fault.
       ```
     - LLDB view:

       ```
             Process 76841 stopped
           * thread #1, name = 'crash.exe', stop reason = signal SIGSEGV: in
           frame #0: 0x00007ffff7ebd4c4 libstdc++.so.6`std::basic_ostream<cha
       libstdc++.so.6`std::operator<<<char, std::char_traits<char>, std::allo
       ->  0x7ffff7ebd4c4 <+4>:  movq   0x8(%rsi), %rdx
           0x7ffff7ebd4c8 <+8>:  movq   (%rsi), %rsi
           0x7ffff7ebd4cb <+11>: jmp    0x7ffff7e0e9f0            ; ___lldb_u

       libstdc++.so.6`std::getline<char, std::char_traits<char>, std::alloca
           0x7ffff7ebd4d0 <+0>:  endbr64
       ```

- Use the `bt` command to view the backtrace. Paste the backtrace into the `debug-questions.txt` document. Identify which function was being executed when the crash happened, as well as which file and line number.
  - GDB Backtrace:

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

  - LLDB Backtrace:

    ```
    frame #1: 0x000055555555654f crash.exe`DisplayValue(ptr=<parent failed to evalu
    frame #2: 0x00005555555565c7 crash.exe`DisplayAll(ptrs=size=1) at crash.cpp:16:
    frame #3: 0x0000555555556790 crash.exe`main at crash.cpp:25:15
    frame #4: 0x00007ffff7b4ed90 libc.so.6`__libc_start_call_main(main=(crash.exe`m
    frame #5: 0x00007ffff7b4ee40 libc.so.6`__libc_start_main_impl(main=(crash.exe`m
    frame #6: 0x0000555555556465 crash.exe`_start + 37
    ```

~

(b) Using breakpoints in logicerror.cpp
Build the logicerror.cpp program with debug symbols:

```
g++ -g logicerror.cpp -o logic.exe
```

- Windows/Linux: Then load it into gdb:
  `gdb logic.exe`
- Mac/Linux: Or load it into lldb:
  `lldb logic.exe`
- Set up a breakpoint to start at the `main()` function.
  - GDB: `break main`
  - LLDB: `breakpoint set --name main`
- Begin running the program. For this program you have to require an argument, like a zip code: 66047.
  - GDB: `run 66047`
  - LLDB: `run 66047`
- Use `n` to step to the next line of code, and `s` to step INTO a function call (do this at `string city = GetCity( zipcode );`)
- You can also use `print VARNAME` to view the current value of a variable.
- GDB stepthrough:

  ```
  (gdb) start 66044
  Temporary breakpoint 1 at 0x26d7: file logicerror.cpp, line 27.
  Starting program: /(...)/practice2_debugging/logic.exe 66044
  [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=2, args=0x7fffffffdb48) at logicerror.cpp
  ```

```
  27      {
(gdb) n
28          if ( argCount != 2 )
(gdb) n
34          int zipcode = stoi( args[1] );
(gdb) n
35          string city = GetCity( zipcode );
(gdb) s
GetCity[abi:cxx11](int) (zipcode=66044) at logicerror.cpp:7
7       {
    (gdb) n
    8           if ( zipcode == 66002 )
    (gdb) n
    12          else if ( zipcode == 66044 || zipcode == 66045 || zipcode
    (gdb) n
    14              return "Lawrence";
    (gdb) n
    24      }
(gdb) n
main (argCount=2, args=0x7fffffffdb48) at logicerror.cpp:36
36          cout << endl
  (gdb) n

  37                << "Zipcode: " << zipcode
  (gdb) n
  38                << ", city: " << city << endl << endl;
  (gdb) print zipcode
  $1 = 66044
  (gdb) print city
  $2 = "Lawrence"
  (gdb) n
  Zipcode: 66044, city: Lawrence

  40          return 0;
  (gdb) n
  41      }
```

- LLDB Stepthrough:

```
Process 77292 stopped
* thread #1, name = 'logic.exe', stop reason = step over
frame #0: 0x00005555555567cc logic.exe`main(argCount=2, args=0x00007ffff
33
34       int zipcode = stoi( args[1] );
35       string city = GetCity( zipcode );
-> 36       cout << endl
      37                << "Zipcode: " << zipcode
      38              << ", city: " << city << endl << endl;
```

After stepping through the program once, start it again with the zipcode 66047. This should return "Lawrence", but it doesn't. Step through the program to locate what instead gets returned, and iden-

tify which line of code has the logic error. Type your answers in the `debug-questions.txt` file.

---

3. Practice 3 - Friends

   This one is small - at the top of main.cpp there is a `Pet` class declared with a constructor function to set values of the object's *private member variables*. Besides that, there are no **accessor** functions to retrieve the private member data.

   Within the class declaration, declare that a new function (which we will define in a moment) is a `friend`:

   ```
   friend void Display( const Pet& pet );
   ```

   Below the function declaration, use this same function header to define the function. This function is a **friend** of the Pet class, so it will have access to the **Pet** class member variables.

   Within the Display function, write a `cout` statement that displays the pet's name, animal, personality, and age.

   (a) Example output

   ```
   $ a.exe
   Kabe the Cat (Sleepy), 11 years old.
   Luna the Cat (Troublemaker), 8 years old.
   Daisy the Dog (Grumpy), 14 years old.
   ```

---

4. Practice 4 - Templates

   Within the Products.h file there are three structs declared: `FoodProduct`, `SongProduct`, and `BookProduct`. They have some variables in common, but other variables that are not shared:

	name	price	quantity	calories	minutes	author	year
FoodProduct	y	y	y	y			
SongProduct	y	y	y		y		
BookProduct	y	y	y			y	y

   Each struct also contains **Constructors**, a `Setup` function, and a `Display` function.

   Within **Functions.h** you'll implement two templated functions:
   ~

   (a) float TotalValue(vector<T>& arr)

   This function will iterate through all of the elements of `arr` and add the worth of each item (`price`) multiplied by the amount of that item in stock (`quantity`). At the end of the function return the sum.
   ~

(b) void Display(const vector<T>& arr)

Within this function, use a loop to iterate over all of the elements of `arr` and call each item's `Display()` function.

~

(c) Updates to `main()`

Within main, do the following:

i. Create vectors and initialize data:
   - Create a vector of FoodProduct, a vector of SongProduct, and a vector of BookProduct.
   - Initialize each one with at least 2 products each.

ii. Call the `TotalValue` function on each of your 3 vectors, displaying the result for each one.

iii. After the table header given later in `main()`, call the `Display` function on each of the three vectors.

~

(d) Example output

Once done your output should look something like this:

```
$ a.exe
Total food value: $61.75
Total song value: $16.43
Total book value: $175.68

NAME                PRICE     QUANTITY  *CALORIES *MINUTES  *AUTHOR   *YEAR
-------------------------------------------------------------------------------
burrito             2.49      10        200       -         -         -
burger              4.99      5         400       -         -         -
banana              1.19      10        10        -         -         -
wonderwall          1.99      2         -         3         -         -
freebird            2.49      5         -         9         -         -
xenogenesis         15.99     5         -         -         butler    1987
sabriel             7.99      7         -         -         nix       1995
frankenstein        1.99      20        -         -         shelley   1818
```

2.6.4 Graded programs

1. Graded program - Fixed Array structure

(a) Getting started The graded program contains the following:

- **FixedArray.h** - Where the templated fixed array structure is created.
- **FixedArrayTester.h** - A tester class and its functions are declared here.
- **FixedArrayTester.cpp** - The tester functions are defined here.
- **main.cpp** - Program starting point.

You can build the program with debug symbols like this:

```
g++ -g *.h *.cpp -o arr.exe
```

When you run the program, you can either run with the tester:

```
./arr.exe test
FixedArrayTester - RunAll
(etc)
```

Or don't include any arguments to get the basic program to run:

```
./arr.exe
Display array:
0.
1.
2.
3.
4.
5.
6.
7.
8.
9.
10. Segmentation fault
```

At the moment it crashes. Once you implement the core functionality, *well written code* will avoid crashes... however, I've hidden two errors within `main()`. When you find the erroneous code, comment out that line.

The best way to approach this assignment is to:

i. Read through the function behaviors (in this document).

ii. Implement the **tests** in **FixedArrayTester.cpp**. Build and run the tests to check that most of them fail.

iii. Implement the **functions** in **FixedArray.h**. Build and run the tests to check that their implementation is correct.

iv. Use gdb to run the main program, step through to find what lines of code are causing the program to crash. Comment out the bad lines.

- Introduction to Graded Program (video)

~

(b) FixedArray functionality

FixedArray() - constructor

- **Inputs:** None
- **Expected result:**
  - `ARRAY_SIZE` should be initialized to 10.
  - `m_itemCount` should be initialized to 0.
- **Test case:**

Action	Expected result
FixedArray<int> test	test `m_itemCount` is 0 test `ARRAY_SIZE` is 10

Only one test is required for this function since there is no input and every call to the constructor will result in the same thing.
~

`size_t` **Size() const;** • **Inputs:** The member variable `m_itemCount` may have different values during the lifetime of FixedArray.

- **Expected result:**
  - The Size function should return whatever the current amount of items in the array is - this is represented with `m_itemCount`.
- **Example tests cases:**

State	Expected result
itemCount is 5	Size() returns 5
itemCount is 2	Size() returns 2

~

void Clear(); • **Inputs:** None

- **Expected result:**
  - After call, the array is considered "empty"; we use lazy deletion by setting `m_itemCount` to 0, which ensures that elements will be erased next time we insert new items.
- **Example tests cases:**

State	Function call	Expected result
itemCount is 5	Clear()	itemCount is 0
itemCount is 2	Clear()	itemCount is 0

~

bool IsEmpty() const; • **Inputs:** None

- **Expected result:** Returns `true` if the array is empty, or `false` otherwise.
- **Example tests cases:**

State	Expected result
itemCount is 5	IsEmpty() returns false
itemCount is 0	IsEmpty() returns true

~

bool IsFull() const; • **Inputs:** None

- **Expected result:** Returns `true` if the array is full, or `false` otherwise.
- **Example tests cases:**

State	Expected result
itemCount is 0	IsFull() returns false
itemCount is 5	IsFull() returns false
itemCount is ARRAYSIZE	IsFull() returns true

~

void PushBack(T newItem) • **Inputs:** A new item to store in the array.

- **Expected result:** If there is space in the array, the new item will be added at the next available space.

- **Example tests cases:**

State	Function call	Expected result
array is empty	PushBack("A")	array[0] is "A", itemCount is 1
array has 1 item	PushBack("B")	array[1] is "B", itemCount is 2
array is full	PushBack("X")	**TEMPORARY:** cout error and return early; (later:

~

void PopBack() • **Inputs:** None

- **Expected result:** The item at the "end" of the list is lazy deleted.
- **Example tests cases:**

State	Function call	Expected result
array has 2 items	PopBack()	itemCount is 1
array has 1 item	PopBack()	itemCount is 0
array is empty	PopBack()	**TEMPORARY:** cout error and return early; (later: exc

~

T& GetBack() • **Inputs:** None

- **Expected result:** The item at the "end" of the list is returned.
- **Example tests cases:**

State	Function call	Expected result
array has 3 items	GetBack()	arr[2] is returned
array has 2 items	GetBack()	arr[1] is returned
array is empty	GetBack()	**TEMPORARY:** cout error, return `m_array[0]` for now

~

T& GetFront() • **Inputs:** None

- **Expected result:** Returns the item at the "front" of the list.
- **Example tests cases:**

State	Function call	Expected result
array has 3 items	GetFront()	arr[0] is returned
array has 2 items	GetFront()	arr[0] is returned
array is empty	GetBack()	**TEMPORARY:** cout error, return `m_array[0]` for now

~

T& GetAt(`size_t` index) • **Inputs:** The index of an item in the array.

- **Expected result:** The item at that index.
- **Example tests cases:**

State	Function call	Expected result
array has 3 items	GetAt(1)	arr[1] is returned
array has 4 items	GetAt(5)	**TEMPORARY:** cout error, return `m_array[0]` for now
array is empty	GetBack()	**TEMPORARY:** cout error, return `m_array[0]` for now

~

`size_t` **Search(T item) const**

- **Inputs:** An item to look for in the array.
- **Expected result:** Returns the index where the item was found.
- **Example tests cases:**

State	Function call	Expected result
array has "A", "B", "C"	Search("B")	1 is returned
array has "W", "X", "Y", "Z"	Search("Z")	3 is returned
array has "C", "A", "T"	Search("S")	**TEMPORARY:** cout error, re

~

(c) Implementing the tests

You should begin by implementing the tests within **FixedArrayTester.cpp**. This will help you understand the functionality of the FixedArray, and give you a way to verify your work.

Keep an eye on the FixedArray class declaration so you know what functions and variables are part of it. Its private member variables are:

```
T m_array[10];
const size_t ARRAY_SIZE;
size_t m_itemCount;
```

And I'll step through the functionality and suggested test cases below.
~

(d) Implementing the FixedArray functionality

Now implement the functions in **FixedArray.h**. Remember that this is a templated class, so all of its declarations *and* definitions must go in the .h file.
~

(e) Discovering the errors

You can use the `gdb` program to look at the **backtrace** of functions called and line number where a crash occurs, and use **breakpoints** to pause the program execution and investigate program flow and variable values.

(You can also view the reference page here: https://gitlab.com/moosadee/courses/-/blob/main/reference/gdb.org?ref_type=heads.)

Once you find the lines of code in `main()` causing the errors, comment those lines out.

i. gdb quick reference:

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

- Within gdb:
  - `run` - Run a program normally
  - `start` - Start the program, pausing at the first executed line.
  - `print VARNAME` - Prints out the value of a variable in scope at the breakpoint.
  - `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).

~

(f) Output

Once you've implemented the tests and the FixedArray functionality, all tests should pass. At minimum, I've provided a few tests, so here's a look at them passing:

```
$ .\a.exe test

FixedArrayTester - RunAll
Set m_itemCount to 5, Size() should return 5... PASS
Set m_itemCount to 5, IsEmpty() should return false... PASS
Set m_itemCount to 2, call PopBack(). m_itemCount should now be 1... PASS
Put 10, 20, 30 in array. Call GetBack(). Should return 30... PASS
Put 10, 20, 30 in array. Call GetFront(). Should return 10... PASS
Put 10, 20, 30 in array. Call GetAt(1). Should return 20... PASS
Create empty array. Call PushBack(A). itemCount should be 1, m_array[0] should be A. P
Add A B C to array. Search for B. Should return 1. PASS
```

But you should also implement additional tests to check more cases. And assuming your input/output logic is right, the tests *should* pass. :)

~

After the tests pass, run the main program itself.

```
$ g++ -g *.h *.cpp -o arr.exe
$ gdb arr.exe
```

Utilize the debugger to find the lines of code in `main()` that are causing the program to crash.

I'm not going to show you the full working program because that would spoil the bugs ;)

```
$ .\a.exe
Display array:
0. CS 134
1. CS 200
2. CS 235
```

```
3. CS 250

Back:  CS 250
(and more...)
```

Once you've commented out the bugs, you should be good to backup your changes to the GitLab server and turn in your work!

3 Week 3: Intro to data structures, Array-based structures

3.1 Intro: Exploring data

For a moment, let's forget about computers and think about an old timey shop ran by one fella. Let's call this enthusiastic entrepreneur "Stan", and he sells snacks. What kind of data does he have access to, as he sells items to customers?

He probably keeps a log of inventory - how much of each item he's bought. He can see from how much has sold, which items are popular and what he needs to order more of. Perhaps the Butterscotch Bars are very popular so he needs to make sure to order double the stock to keep up with demand.

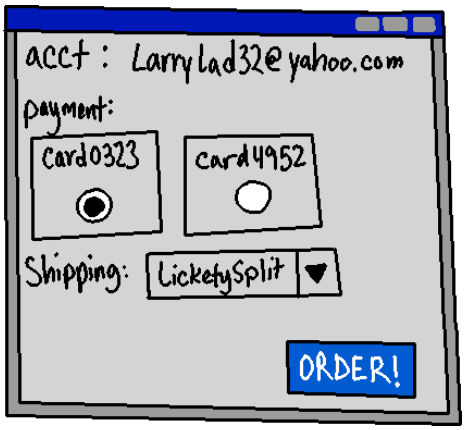
If it's just ole' Stan here manning the shoppe, perhaps he notices patterns as well - little Sally tends to buy Nickel Candies after school, so whenever he sees her he knows what she's going to order.

Larry the Lad likes buying Cinnamon Firecrackers, but he buys Lemondrop Llamas on Fridays. Good ole' Stan asks Larry the Lad, "So when do you have a hankerin' for these Llamas?" and Larry the Lad tells Stan, "I visit my grand pappy's house on Fridays, and he sure likes them Lemondrops!". With or without that context, Stan can still see the pattern of Larry the Lady purchasing

Cinnamon Firecrackers on Monday and Wednesday, and Lemondrop Llamas on Friday.

The longer he runs the shoppe and the more regular customers he gets, he keeps a mental note of the patterns of his customers. And for a small olde timey shoppe like his, this can create good customer service, perhaps he offers a more customized experience than his competitor, Big City Candy Co.

Back to the modern day...



Virtually everything we do on the internet generates data, whether we are actively providing data or not. Merely just *browsing* websites generates data that companies use to optimize sales or ad revenue. Some pages track where your mouse cursor sits while you visit a page, because people often point their mouse cursor to what they're reading - this data creates a heat map that companies can use to figure out the best layout for their pages to funnel potential customers to where they want them.

Even in-person systems we use are digitized and recording data. When you purchase something with a card at a store, they can keep a record for you as a customer and relate your purchases to that card - even better if you have a "super saver" type membership card as well.

Scrolling through social media? Which ads do you pause over? Maybe you got distracted by your dog, but the data shows potential interest - you spent a few seconds longer looking at *this* ad over the *other* ads. (And if you click the ad, then they really have a lead!) And of course your order history can be used to figure out what to market to you and when.

As you carry your phone around with you, location data is often tracked. When you get together with a friend, you might start seeing ads influenced by your friend - you're hanging out, so perhaps *their ads* will appeal to *you*.

With the speed of the internet and relatively cheap cost of storage, data is being tracked all the time, and in large quantities. "Big Data", as well as using

Machine Learning to do what Good Ole' Stan does, are part of our modern tech landscape.

~

3.1.1 OK but how does that relate to *this* class?

Data needs to be stored, and while long-term storage means putting it in some kind of database, receiving data to save, or pulling data to crunch, requires software at some level. We need to be able to store the data in a **data structure**, and we need to choose the structure that is the most **efficient** for the goals we have - do we prioritize **access speed** or **insert speed**? Do we want a structure that **auto-sorts** incoming data so we can find records more quickly? What kind of **sorting** algorithm do we use to sort the unsorted data?

In this class we'll be focusing on common data structures you'll see as a software developer - array-based structures, link-based structures, binary search trees, hash tables, stacks, and queues. We're going to learn *how* they work, and the efficiency of their functionality. While you probably won't be writing these structures from scratch *after* this course, knowing how they work helps you make an informed design decision on what is the best tool for the job.

~

3.1.2 Example: Spotify

We can get some kind of idea of the underlying structure of a system by looking at the **API** (Application Programming Interface) that it exposes to the outside world. Many modern websites offer APIs in order to allow third parties to create helper programs and services that work with the data stored by the original service.

Looking at the Spotify API documentation (https://developer.spotify.com/documentation/web-api), we can see several main "objects" it provides:

```
1. Albums        6. Episodes     11. Search
2. Artists       7. Genres       12. Shows
3. Audiobooks    8. Markets      13. Tracks
4. Categories    9. Player       14. Users
5. Chapters     10. Playlists
```

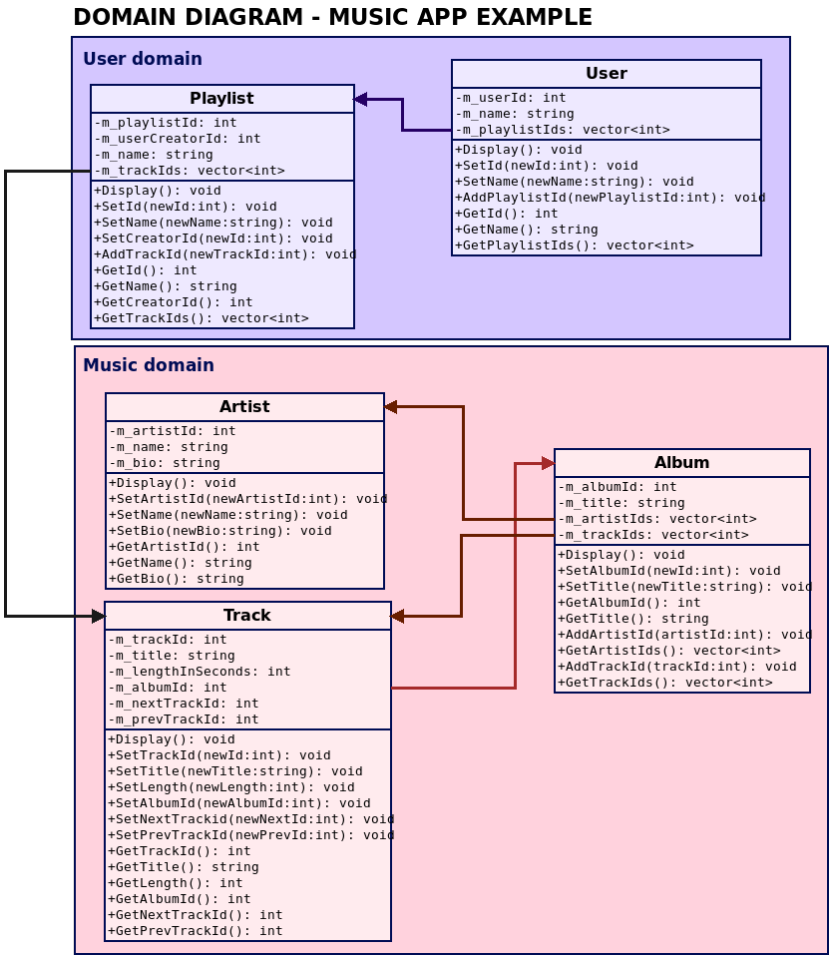
If you click on one of the objects, it will show an example of requests that an external program can call, such as **Get Album**, **Get Album Tracks**, **Save Albums for Current User**, and so on.

Clicking on a request type will also show a **Response Sample**, which shows what kind of information may be returned with a call, such as **Get Album**:

```
{
  "album_type": "compilation",
  "total_tracks": 9,
  "available_markets": [ "CA", "BR", "IT" ],
  "id": "2up3OPMp9Tb4dAKM2erWXQ",
  "images": [
    {
      "url": "https://i.scdn.co/image/ab676...",
      "height": 300, "width": 300
    }
  ],
  "release_date": "1981-12",
  "type": "album",
  "artists": [
    { "id": "string", "name": "string", }
  ],
  "tracks": {
    "total": 4,
    "href": "https://api.spotify.com/v1/me/shows?offset=0&limit=20",
    "next": "https://api.spotify.com/v1/me/shows?offset=1&limit=1",
    "previous": "https://api.spotify.com/v1/me/shows?offset=1&limit=1"
  },
  "copyrights": [
    { "text": "string", "type": "string" }
  ],
  "genres": [ "Egg punk", "Noise rock" ],
  "popularity": 0
}
```


Figure 2: Example response data from the Spotify API documentation

The data and the form it's returned in via API response isn't necessarily how the data is stored in the database itself, or how the data is stored in objects when worked on by the Spotify servers themselves. Usually, API systems pulls data from multiple sources and bundles it together to return as a response. But, this can at least give us some idea of how the actual system may be organized.



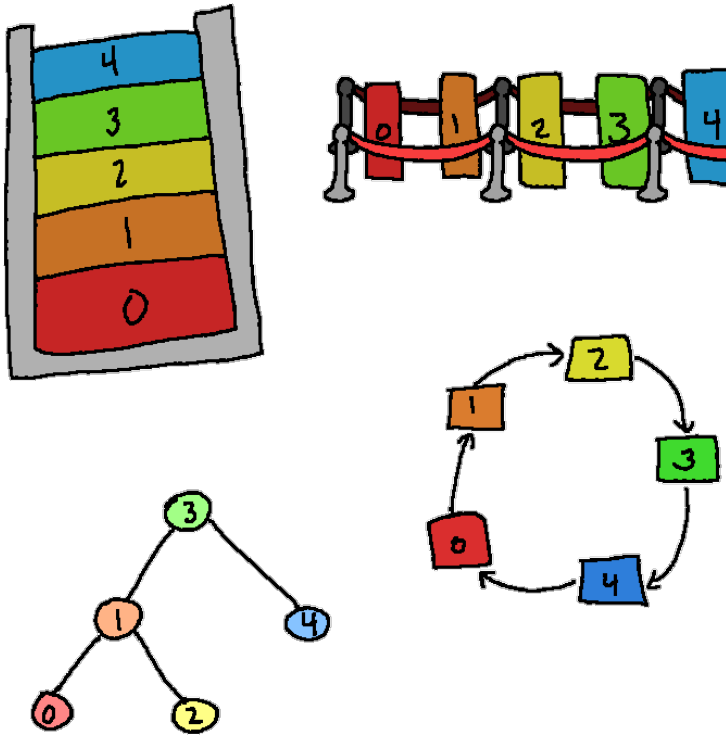
This is just a small example, containing a "Music" section (Artist, Album, Track) and a "User" section (User, Playlist). The arrows show relationships *between* the objects. But this diagram is also lacking any of the back-end functionality that helps everything actually *do* something. On their own, it's just data stored somewhere. A program has to be written around it.

3.1.3 Additional APIs

In the Tech Literacy assignment you will brainstorm about data used for specific types of software. You may want to reference the API pages for these common apps for ideas:

- Instagram: https://developers.facebook.com/docs/instagram-api/reference
- Canvas LMS: https://canvas.instructure.com/doc/api/all_resources.html
- Grubhub: https://developer.grubhub.com/

3.2 Intro: About Data Structures



3.2.1 What are data structures?

A **data structure** is an object (a class, a struct - some type of **structured** thing) that holds **data**. In particular, the entire job of a data structure object is to store data and provide an interface for **adding, removing,** and **accessing** that data.

"Don't all the classes we write hold data? How is this different from other objects I've defined?"

In the past, you may have written classes to represent objects, like perhaps a book. A book could have member variables like its title, isbn, and year published, and some methods that help us interface with a book.

Book
- title : string - isbn : string - year : int
+ Setup() : void + Display() : void

However - this is not a data structure. We *could*, however, use a data structure to *store* a list of books.

A data structure will store a series of some sort of data. Often, it will use an **array** or a **linked list** as a base structure and functionality will be built on top.

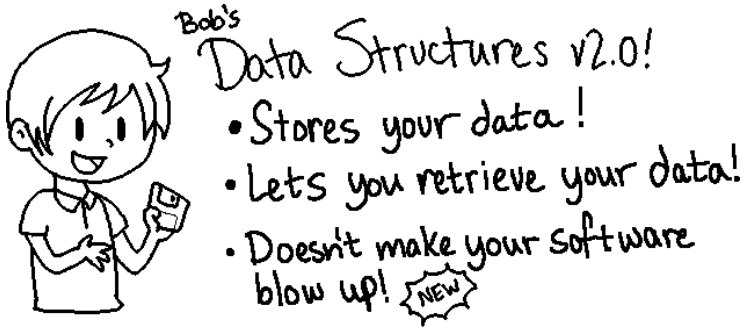
Ideally, the user doesn't care about *how* the data structure works, they just care that they can **add, remove,** and **access** data they want to store.

Ideally, when we are writing a data structure, it should be:

- Generic, so you could store **any data type** in it.
- Reusable, so that the data structure can be used in many different programs.
- Robust, offering exception handling to prevent the program from crashing when something goes wrong with it.
- Encapsulated, handling the inner-workings of dealing with the data, without the user (or other programmers working outside the data structure) having to write special code to perform certain operations.

SmartBookArray
- bookArray : `Book[]` - arraySize : `int` - bookCount : `int`
+ AddBook(newBook : Book) : `void` + RemoveBook(index : int) : `void` + RemoveBook(title : string) : `void` + GetBook(index : int) : `Book&` + FindBookIndex(title : string) : `int` + DisplayAll() : `void` + Size() : `int` + IsEmpty() : `bool`

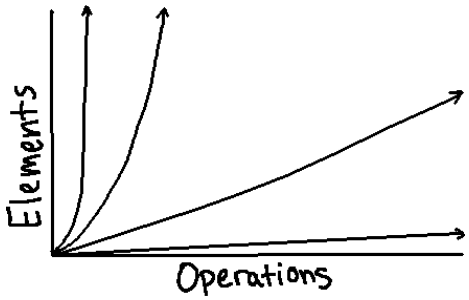
The way I try to conceptualize the work I'm doing on a data structure is to pretend that I'm a developer that is going to create and sell a C++ library of data structures that *other developers at other companies* can use in their own, completely separate, software projects. If I'm selling my data structures package to other businesses, my code should be dependable, stable, efficient, and relatively easy to use.



~

3.2.2 What is algorithm analysis?

Algorithm Analysis is the process of figuring out how **efficient** a function is and how it scales over time, given more and more data to operate on.

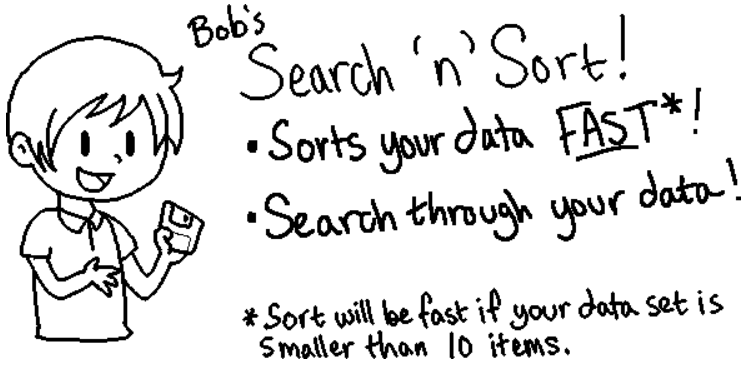


This is another important part of dealing with data structures, as different structures will offer different trade offs when it comes to the efficiency of **data access** functions, **data searching** functions, **data adding** functions, and **data removal** functions. Every operation takes a little bit of processing time, and as you iterate over data, that processing time is multiplied by the amount of times you go through a loop.

Example: Scalability

Let's say we have a sorting algorithm that, for every n records, it takes n^2 program units to find an object. If we had 100 records, then it would take $100^2 = 10,000$ time-units to sort the set of data.

What the time-units are could vary - an older machine might take longer to execute one instruction, and a newer computer might process an instruction much more quickly, but we think of algorithm complexity in this sort of generic form.



Example: Efficiency of an array-based structure

value:	kansas	missouri	arkansas	ohio	oklahoma
index:	0	1	2	3	4

In an array-based structure, we have a series of **elements** in a row, and each element is accessible via its **index** (its position in the array). Arrays allow for random-access, so **accessing** element #2 is instant:

```
cout << arr[2];
```

No matter how many elements there are (n), we can access the element at index 2 without doing any looping. We state that this is $O(1)$ ("Big-O of 1") time complexity for an **access** operation on an **array**.

However, if we were **searching** through the unsorted array for an item, we would have to start at the beginning and look at each item, one at a time, until we either found what we're looking for, or hit the end of the array:

```
for ( int i = 0; i < ARR_SIZE; i++ )
{
    if ( arr[i] == searchTerm )
    {
            return i; // found at this position
    }
}
return -1;      // not found
```

For **search**, the *worst-case scenario* is having to look at *all elements of the array* to ensure what we're looking for isn't there. Given n items in the array, we have to iterate through the loop n times. This would end up being $O(n)$ ("Big-O of n") time complexity for a **search** operation on an **array**.

We can build our data structures on top of an array, but there is also a type of structure called a **linked** structure, which offers its own pros and cons to go with it. We will learn more about algorithm analysis and types of structures later on.

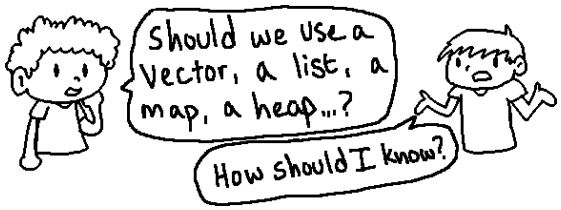
3.2.3 The point of a Data Structures class

Of course, plenty of data structures have already been written and are available out there for you to use. C++ even has the **Standard Template Library** full of structures already built and optimized!

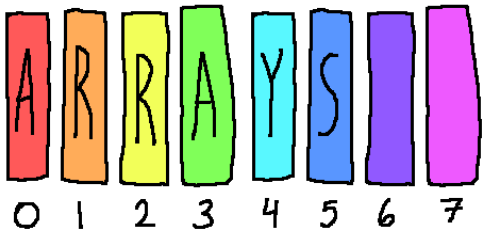
vector	http://www.cplusplus.com/reference/vector/vector/
list	http://www.cplusplus.com/reference/list/list/
map	http://www.cplusplus.com/reference/map/map/
stack	https://cplusplus.com/reference/stack/stack/
queue	https://cplusplus.com/reference/queue/queue/

"So why are we learning to write data structures if they've already been written for us?"

While you generally *won't* be rolling your own linked list for projects or on the job, it *is* important to know how the inner-workings of these structures operate. Knowing how each structure works, its tradeoffs in efficiency, and how it structures its data will help you **choose** what structures to use when faced with a **design decision** for your software.



3.3 Intro: Fixed-array structure



3.3.1 Introduction

First we're going to start off by building a "Smart" fixed-length array, taking a traditional C-style array:

```
int myArray[100];
```

and **wrapping** it within a class, so that we can create functions and helper variables to provide a nice **interface** for a user, so they don't have to worry about the specifics of *how* the array is implemented, they just know that they can **add**, **remove**, and **access** their data. (In this context the "user" would be another programmer using your data structure.)

With the Smart Fixed-Length Array, we'll work through the steps together. Then, for a little more challenge, you'll work on a Smart Dynamic Array, which uses pointers to allow "resizing" of the array. There will be a lot of functionality in common between these two "Smart" Arrays, but also some differences.

3.3.2 Smart fixed array structure

In previous programming courses you've probably worked with **arrays** to store data. You've probably encountered out-of-bounds errors and had to deal with array indices, moving items around, and generally micro-managing your array's data as the program goes. With our traditional array **wrapped** inside of a class, we can look out for these error states and how to manage our array, and do it within the functions - the "programmer-user" doesn't have to think about how it's implemented.

So, what will our array actually *do*?

1. **SmartFixedArray - Functionality**

   Pretend you're licensing out a special Data Structures library that other developers use. Our "programmer-user" is going to want to do some basic tasks:

   - Store data (PUSH)
   - Access data (GET)
   - Discard data (POP)

There could also be other features like Search. We'll be implementing these features and seeing how the implementations are different for each of our different types of data structures.

So how do we achieve this core functionality?

(a) **Creating the array:**

If we weren't storing our array inside a class, we might declare it in a program like this:

```
// Create an array + helper variables
const int ARR_SIZE = 100;
int totalItems = 0;
string data[ARR_SIZE];
```

With a fixed-length array, we have to keep track of the array size `ARRAY_SIZE` to ensure we don't go out-of-bounds in the array... valid indices will be `0` until `ARRAY_SIZE - 1`.

We are also keeping track of the `totalItems` in the array, because even though we have an array with some size, it could be useful to know that currently no data is stored in it. Then, when new data is added, we could increment `totalItems` by 1.

~

Our empty array (`totalItems = 0`):

index	0	1	2	3	4	5
value	""	""	""	""	""	""

(b) **Adding onto the array:**

Adding on to the array would mean choosing an index to put new data. If we were wanting to fill the array from **left-to-right**, we would start at index 0, then 1, then 2, then 3, and so on.

The `totalItems` variable begins at 0. Once the first item is added, it is then 1. Once a second item is added, it is then 2. Basically, the `totalItems` corresponds to the *next index to store new data at...*

```
// Add new item
data[totalItems] = myNewData;
totalItems++;
```

~

One item added (`totalItems = 1`):

index	0	1	2	3	4	5
value	"Cats"	""	""	""	""	""

(c) **Removing items from the array:**

We may also want to remove an item at a specific index from the array. There's not really a way to "delete" an item from an array, but we can overwrite it, if we wanted to...

```
// Remove an item
int index;
cout << "Remove which index? ";
cin >> index;

data[index] = "";
totalItems--;
```

But if we did it this way we would create an array with gaps in it. Additionally now `totalItems` doesn't align to the *next index to insert at*. While we *could* design our SmartArray this way (I have this type of array as the base for our eventual HashTable data structure), but the design we're going for here, we **don't want gaps in the array** and we're **filling it up from left-to-right**.

Before removing an item (`totalItems = 6`):

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	"Snakes"	""

After removing an item (`totalItems = 5`):

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	""	"Snakes"	""

If we designed our array to allow gaps, we would then have to add code to find a new *available* place to add an item we'd have to use a for loop to look for an empty spot to use - this means every time we add a new item it would be *less efficient* because of the "search" operation.

Instead, we could design our arrays so that after a remove, we shift elements backwards to "overwrite" the empty spot. This also solves our issue with `totalItems` not aligning to the *next-index-to-insert-at*.

After removing an item with a "Shift Left" (`totalItems = 5`):

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Snakes"	""	""

Here, we've shifted everything after index 3 (where we removed "Birds" from) and shifted everything backwards by 1, so now "Snakes" is at position 3 instead of 4.

(d) **Searching for an item:**

Another thing a user of this data structure might want to do is search for an item - is this thing stored in the array? If so, what is the index? If the array is **unsorted** (we're not going to do sorting yet), then really the only way to search for the item is to start at the beginning and keep searching until the end. If we find a match, we return that **index number** to let the user know *where* in the array that data was found.

If we get to the end of the loop and nothing has been returned, that means it **isn't in the array** so we would have to return something to symbolize "it's not in the array – such as returning `-1` as the index (negative numbers are invalid indices), or perhaps throwing an **exception**.

Populated array (`totalItems = 5`):

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	"Snakes	""

Searching through the array would look like this:

```
// Search for an item
string findme;
cout << "Find what? ";
getline( cin, findme );

int foundIndex;

for ( int i = 0; i < totalItems; i++ )
{
    if ( data[i] == findme )
    {
        foundIndex = i;   // Found it
        break;
    }
}

// Did not find it
foundIndex = -1;
```

Once we implement this structure within a class, we will turn this into a function that returns data - this is just an example as if we were writing a program in main().

(e) **Retrieving an item at some index:**
Besides adding, removing, and searching for items, users will want to retrieve the data located at some index. This would be a simple return once we're writing our class, but generally accessing *item at index* looks like:

```
// Display element at index
int index;
cout << "Which index? ";
cin >> index;
cout << data[index] << endl;
```

(f) **Visualizing the array:**
Another feature could be to display all elements of the array, which might look something like this:

```
// Display all items
for ( int i = 0; i < totalItems; i++ )
{
    cout << i << ". " << data[i] << endl;
}
```

2. **SmartFixedArray - Predicting errors/exceptions**

(a) **Invalid indices:**
When working with arrays one of the most common errors encountered are **out-of-bounds** errors: When we have an index that is less than 0, or equal to or greater than the size of the array...

```
// Out of bounds!
cout << data[-1]        << endl;    // error!
cout << data[ARR_SIZE]  << endl;    // error!
```

Additionally, if the user tries to access an index $\geq$ `totalItems` then they would retrieve data that is invalid, though this wouldn't crash the program if it is still less than `ARR_SIZE`.

(b) **Array is full:**
With the fixed-length array, we could also run out of space, and we would have to design a way to deal with it. For example, if `totalItems` was equal to `ARR_SIZE`, then we are out of space and doing this would also cause an out-of-bounds error.

```
data[totalItems] = "Ferrets";       // error if full!
totalItems++;
```

As part of designing our data structure, we need to make sure we account for reasonable scenarios that would cause the program to crash and guard against logic errors. This would all be part of how we implement the functionality.

3. **SmartFixedArray - Creating a class to wrap the array**

   For our wrapper class we will need the array and array size / item count integers as part of the private member variables. The public functions would include the "interface" of what the user will want to do with the array, and we could also add private/protected helper functions.

SmartFixedArray	
- m_array	: TYPE[100]
- m_itemCount	: int
- m_arraySize	: const int
+ SmartFixedArray()	
+ PushFront(newData : TYPE)	: void
+ PushBack(newData : TYPE)	: void
+ PushAtIndex(index : int, newData : TYPE)	: void
+ PopFront()	: void
+ PopBack()	: void
+ PopAt(index : int)	: void
+ GetFront()	: TYPE&
+ GetBack()	: TYPE&
+ GetAtIndex(index : int)	: TYPE&
+ Search(item : TYPE)	: int
+ Clear()	: void
+ IsEmpty()	: bool
+ IsFull()	: bool
+ Size()	: int
- ShiftLeft(index : int)	: int
- ShiftRight(index : int)	: int

The class declaration could look like this:

```
template <typename T>
class SmartFixedArray
{
public:
  SmartFixedArray();

  void PushBack( T newItem );
  void PushFront( T newItem );
  void PushAt( T newItem, int index );

  void PopBack();
  void PopFront();
  void PopAt( int index );
```

```
  T GetBack() const;
  T GetFront() const;
  T GetAt( int index ) const;

  int Search( T item ) const;
  void Display() const;
  int Size() const;
  bool IsEmpty() const;
  bool IsFull() const;

private:
  const int m_arraySize;
  T m_array[100];
  int m_itemCount;

  void ShiftLeft( int index );
  void ShiftRight( int index );
};
```

For this example data structure, we are hard-coding the array size to 100 elements. It is unlikely that anyone would use this data structure for anything, but I wanted to start off with a basic fixed-length array. Next, we will move to a dynamic array, which *will* be more useful.

With this class structure set up, lets look into how to actually implement the functionality.

(a) Constructor - Preparing the array to be used

When the constructor is called, we need to set the value of `m_arraySize` as part of the initializer list since it is a const member and needs to be initialized right away. Otherwise, we can initialize `m_itemCount` to 0 as well, since are creating an empty array.

```
template <typename T>
SmartFixedArray<T>::SmartFixedArray()
    : m_arraySize( 100 )
{
  m_itemCount = 0;
}
```

(b) **int Size() const**

This function returns the value from `m_itemCount` - the current amount of items in the array.

(c) **bool IsEmpty() const**

Returns `true` if the array is empty and `false` if not. The array is empty if `m_itemCount` is 0.

(d) **bool IsFull() const**

Returns `true` if the array is full and `false` if not. The array is full if `m_itemCount` is equal to `m_arraySize`.

(e) **void Display() const**

Iterates over all the elements of the array displaying each item.

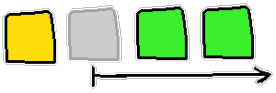
```
void SmartFixedArray<T>::Display() const
{
  for ( int i = 0; i < m_itemCount; i++ )
  {
    cout << i << ". " << m_array[i] << endl;
  }
}
```

(f) **int Search(T item) const**

Iterates over all the elements of the array searching for the *item*. If a match is found, it returns the index of that element. Otherwise, if it finishes looping over the array and has not yet returned, it means that nothing was found. Return -1 in this case, or throw an exception (this is a design decision).

```
int SmartFixedArray<T>::Search( T item ) const {
  for ( int i = 0; i < m_itemCount; i++ ) {
    if ( m_array[i] == item ) {
      return i;
    }
  }
  throw runtime_error( "Could not find item!" );
}
```

(g) **void ShiftRight(int index)**



Error checks:

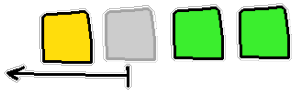
- If the array is full, then throw an exception.
- If the index is invalid, then throw an exception.

Functionality:

- Iterate over the array, starting at the index of the first empty space. Moving backwards, copy each item over from the previous index. Loop until hitting the *index*.

```
void SmartFixedArray<T>::ShiftRight(int index) {
    if ( index < 0 || index >= m_itemCount ) {
        throw out_of_range( "Index is out of bounds!" );
    }
    else if ( IsFull() ) {
        throw length_error( "Array is full!" );
    }

    for ( int i = m_itemCount; i > index; i-- ) {
        m_array[i] = m_array[i-1];
    }
}
```

(h) **void ShiftLeft(int index)**



Error checks:

- If the index is invalid, then throw an exception.

Functionality:

- Iterate over the array, starting at the index given and going until the last element. Copy each neighbor from the right to the current index until we get to the end.

```
void SmartFixedArray<T>::ShiftLeft(int index) {
{
  if ( index < 0 || index >= m_itemCount ) {
    throw out_of_range( "Index is out of bounds!" );
  }

  for ( int i = index; i < m_itemCount-1; i++ ) {
    m_array[i] = m_array[i+1];
  }
}
```

(Continued...)

(i) **void PushBack(T newItem)**

We have three different Push functions in order to add an item to different parts of the array, which will each require slightly different logic. `PushBack` adds a new item to the **end** of the list of elements.

Functionality:

i. ERROR CHECK:
   A. Use the `IsFull()` function to check if the array is full... if it is, then throw an exception, because we can't resize this type of array!

ii. All good to go:
   A. The `m_itemCount` is also the index of the next available space in the array... Add the `newItem` to the `m_array` at the index `m_itemCount`.
   B. Increment `m_itemCount` by 1.

BEFORE:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	""	""

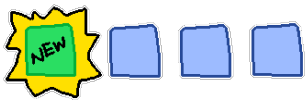
`m_itemCount` is 4.

AFTER:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	**"Ferrets"**	""

`m_itemCount` is now 5.

(j) **void PushFront(T newItem)**



`PushFront` adds a new item to the beginning of the list of elements. This requires shifting everything forward to make space for the new item...

Functionality:

i. ERROR CHECK:
   A. Use the `IsFull()` function to check if the array is full... if it is, then throw an exception, because we can't resize this type of array!

ii. All good to go:
   A. The front of the array is always index `0`. To make space for our new item, we need to call `ShiftRight( 0 )` to scoot everything over by 1 space.
   B. Add the `newItem` to the `m_array` at the index `0`.
   C. Increment `m_itemCount` by 1.

BEFORE:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	""	""

`m_itemCount` is 4.

MAKE ROOM by calling `ShiftRight(0)`

index	0	1	2	3	4	5
value	""	"Cats"	"Dogs"	"Mice"	"Birds"	""

`m_itemCount` is now 5.

AFTER:

index	0	1	2	3	4	5
value	**"Ferrets"**	"Cats"	"Dogs"	"Mice"	"Birds"	""

(k) **void PushAt(T newItem, int index)**

`PushAt` adds a new item to the index given. This requires shifting everything forward to make space for the new item...

Functionality:

i. ERROR CHECKS:
   A. Use the `IsFull()` function to check if the array is full... if it is, then throw an exception, because we can't resize this type of array!
   B. If `index` is out of bounds (less than 0 or greater than `m_itemCount`!) then throw an exception - can't add at an invalid index!

ii. DRY (Don't Repeat Yourself) checks:
   A. If the index is 0, call `PushFront(newItem)` instead.
   B. If the index is `m_itemCount`, call `PushBack(newItem)` instead.

iii. ELSE:
   A. The `index` for our new item has been passed in. To make space for our new item, we need to call `ShiftRight(index)` to scoot everything over by 1 space.
   B. Add the `newItem` to the `m_array` at the index `index`.
   C. Increment `m_itemCount` by 1.

BEFORE:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	""	""

`m_itemCount` is 4.

MAKE ROOM by calling `ShiftRight( 2 )`

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	""	"Mice"	"Birds"	""

AFTER `PushAt("Ferrets", 2)`

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	**"Ferrets"**	"Mice"	"Birds"	""

`m_itemCount` is now 5.

(l) **T GetBack() const**

The Get functions just return data from the array. The "Back" of the array will be at `m_itemCount-1` (the last valid index), the "Front" of the array is at `0`. Note that with the Get functions, we need to check `IsEmpty()` first - if the array is empty, we can't retrieve any data!

Functionality:

i. ERROR CHECK:
   A. Use the `IsEmpty()` function to check if the array is empty... if it is, then throw an exception, because we can't retrieve from an empty array!

ii. All good to go:
   A. Return the item from `m_array` at the index `m_itemCount-1`.

(m) **T GetFront() const**

Functionality:

i. ERROR CHECK:
   A. Use the `IsEmpty()` function to check if the array is empty... if it is, then throw an exception, because we can't retrieve from an empty array!

ii. All good to go:
   A. Return the item from `m_array` at the index `0`.

(n) **T GetAt(int index) const**

Functionality:

i. ERROR CHECK:

A. Use the `IsEmpty()` function to check if the array is empty... if it is, then throw an exception, because we can't retrieve from an empty array!

B. If `index` is out of bounds (less than 0 or greater than or equal to `m_itemCount`!) then throw an exception - can't retrieve from an invalid index!

ii. All good to go:

A. Return the item from `m_array` at the index `index`.

(o) **void PopBack()**

Functionality:

i. ERROR CHECK:

A. Use the `IsEmpty()` function to check if the array is empty... if it is, then throw an exception, because we can't remove data from an empty array!

ii. All good to go:

A. Just decrement `m_itemCount` by 1. (Next time something is added, it will overwrite this location!)

BEFORE:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Ferrets"	"Mice"	"Birds"	""

`m_itemCount` is 5.

AFTER `PopBack()`

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Ferrets"	"Mice"	""	""

`m_itemCount` is now 4.

(p) **void PopFront()**

Functionality:

i. ERROR CHECK:

   A. Use the `IsEmpty()` function to check if the array is empty... if it is, then throw an exception, because we can't remove data from an empty array!

ii. All good to go:

   A. Call `ShiftLeft(0)`, which will essentially "erase" the item by overwriting it.

   B. Decrement `m_itemCount` by 1.

BEFORE:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Ferrets"	"Mice"	"Birds"	""

`m_itemCount` is 5.

AFTER `PopFront()`

index	0	1	2	3	4	5
value	"Dogs"	"Ferrets"	"Mice"	"Birds"	""	""

`m_itemCount` is now 4.

(Continued...)

(q) **void PopAt(int index)**

Functionality:

i. ERROR CHECK:

A. Use the `IsEmpty()` function to check if the array is empty... if it is, then throw an exception, because we can't remove data from an empty array!

B. If `index` is out of bounds (less than 0 or greater than or equal to `m_itemCount`!) then throw an exception - can't remove data from an invalid index!

ii. All good to go:

A. Call `ShiftLeft(index)`, which will essentially "erase" the item by overwriting it.

B. Decrement `m_itemCount` by 1.

BEFORE:

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Ferrets"	"Mice"	"Birds"	""

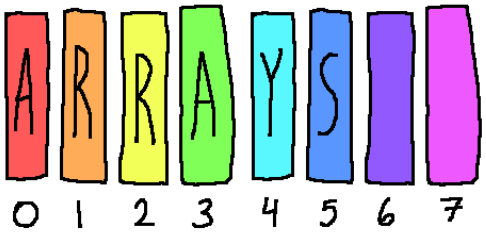
`m_itemCount` is 5.

AFTER `PopAt(2)` **and** `ShiftLeft(2)`

index	0	1	2	3	4	5
value	"Cats"	"Dogs"	"Mice"	"Birds"	""	""

`m_itemCount` is now 4.

3.4 Intro: Dynamic-array structure



3.4.1 Smart dynamic array structure

With the dynamic array structure, most of it will be similar to the Fixed Array Structure, except we now have to deal with memory management and pointers.

Instead of throwing exceptions if the array is full, with a dynamic array we can resize it instead. However, we also now need to check for the array pointer (`m_array` in this example) pointing to `nullptr`.

The class declaration could look like this:

```
class SmartDynamicArray
{
public:
  SmartDynamicArray();
  ~SmartDynamicArray();

  void PushBack( T newItem );
  void PushFront( T newItem );
  void PushAt( T newItem, int index );

  void PopBack();
  void PopFront();
  void PopAt( int index );

  T GetBack() const;
  T GetFront() const;
  T GetAt( int index ) const;

  int Search( T item ) const;

  void Display() const;
  int Size() const;
  bool IsEmpty() const;
  void Clear();

private:
  T* m_array;
  int m_arraySize;
  int m_itemCount;
```

```
  void ShiftLeft( int index );
  void ShiftRight( int index );

  void AllocateMemory( int size );
  void Resize( int newSize );

  bool IsFull() const;
};
```

~

1. Constructor - Preparing the array to be used

   With the constructor here, it is important to assign `nullptr` to the pointer so that we don't try to dereference a garbage address. The array size and item count variables should also be assigned to 0.

   ```
   template <typename T>
   SmartDynamicArray<T>::SmartDynamicArray()
   {
     m_array = nullptr;
     m_itemCount = 0;
     m_arraySize = 0;
   }
   ```

   ~

2. Destructor - Cleaning up the array

   Having a destructor is also very important since we're working with pointers. Before our data structure is destroyed, we need to make sure that we free any allocated memory.

   ```
   template <typename T>
   SmartDynamicArray<T>::~SmartDynamicArray()
   {
     if ( m_array != nullptr )
     {
       delete [] m_array;
       m_array = nullptr;
     }
     m_arraySize = 0;
     m_itemCount = 0;
   }
   ```

   We can also put this functionality into the Clear() function, and call Clear() from the destructor.

   ~

3. void AllocateMemory(int size)

When the array is not in use, the `m_array` pointer will be pointing to `nullptr`. In this case, we need to allocate space for a new array before we can begin putting items into it.

Error checks:

- If the `m_array` pointer is NOT pointing to nullptr, throw an exception.

Functionality:

(a) Allocate space for the array via the `m_array` pointer.

(b) Assign `m_arraySize` to the size passed in as a parameter.

(c) Set `m_itemCount` to 0.

```
template <typename T>
void SmartDynamicArray<T>::AllocateMemory( int size )
{
  if ( m_array != nullptr )
  {
    throw logic_error( "Can't allocate memory, m_array is already pointing to a memory add
  }

  m_array = new T[ size ];
  m_arraySize = size;
  m_itemCount = 0;
}
```

For this function, if the array is already pointing somewhere, we don't want to just erase all that data and allocate space for a new array. We want to try to prevent data loss, so it would be better to throw an exception instead so that the caller can decide how they want to handle it.
~

4. void Resize(int newSize)

We can't technically resize an array that has been created. We can, however, allocate more space for a bigger array and copy all the data over and then update the pointer to point to the new array. And that's exactly how we "resize" our dynamic array.

Error checks:

- If the new size is smaller than the old size, throw an exception.

Functionality:

(a) If `m_array` is pointing to nullptr, then just call AllocateMemory instead.

(b) Otherwise:

   i. Allocate space for a new array with the new size using a new pointer.

ii. Iterate over all the elements of the old array, copying each element to the new array.
iii. Free space from the old array.
iv. Update the `m_array` pointer to point to the new array address.
v. Update the `m_arraySize` to the new size.

```
template <typename T>
void SmartDynamicArray<T>::Resize( int newSize )
{
  if ( newSize < m_arraySize )
  {
    throw logic_error( "Invalid size!" );
  }
  else if ( m_array == nullptr )
  {
    AllocateMemory( newSize );
  }
  else
  {
    T* newArray = new T[newSize]; // New array

    // Copy values over
    for ( int i = 0; i < m_itemCount; i++ )
    {
      newArray[i] = m_array[i];
    }

    delete [] m_array;      // Deallocate old space
    m_array = newArray;     // Update pointer
    m_arraySize = newSize;  // Update size
  }
}

~
```

5. Updating other functions
   - **ShiftLeft:** If `m_array` is pointing to `nullptr`, throw an exception.
   - **ShiftRight:**
     - If `m_array` is pointing to `nullptr`, throw an exception.
     - If the array is full, call Resize.
   - **PushBack, PushFront, PushAt:**
     - If `m_array` is pointing to `nullptr` call AllocateMemory.
     - If the array is full, call Resize.

3.5 Lab: Array structures

3.5.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

3.5.2 Assignment information

Getting started:

1. This lab is located in your **repository folder** under `ArrayStructures`.
2. In VS Code use "Open Folder" to open the specific subprogram you wish to work with.
3. BUILD: Since this is a big project, use the `make` command (in the same directory as the Makefile) to build the program.
4. RUN: `./release.exe` for the release version, or `gdb ./debug.exe` for the debug version.

Practice programs & graded programs: We will work on the Fixed Length Array during class, this will be practice. You will then work on the slightly modified Dynamic Array structure as the lab.

Turning in your work:

To turn in the assignment, make sure you've **committed** and **synced (pushed)** your changes to your GitLab repository. Double check the file from the GitLab webpage to ensure it's up-to-date. Then, submit the URL to the folder for the week.

Working on this assignment:

After building the program you can run it with the following arguments:

- `./release.exe fixedarray` - Run tests for fixed array
- `./release.exe dynamicarray` - Run tests for dynamic array
- Array Structures reference:
  - Textbook, archived class lecture
  - Lecture sildes
- Quick references:
  - Using Git and VS Code
  - Program arguments
- Assignment direct link

3.5.3 Included files:

```
ArrayStructures
 DataStructures
    ArrayQueue
       ArrayQueue.h
       ArrayQueueTester.cpp
       ArrayQueueTester.h
    ArrayStack
       ArrayStack.h
       ArrayStackTester.cpp
       ArrayStackTester.h
    DynamicArray
       DynamicArray.h
       DynamicArrayTester.cpp
       DynamicArrayTester.h
    FixedArray
        FixedArray.h
        FixedArrayTester.cpp
        FixedArrayTester.h
 Exceptions
    InvalidIndexException.h
    ItemNotFoundException.h
    NotImplementedException.h
    NullptrException.h
    StructureEmptyException.h
    StructureFullException.h
 instructions.org
 main.cpp
 Makefile
 Utilities
     StringHelper.cpp
     StringHelper.h
     Style.cpp
     Style.h
```

4 Week 4: Exceptions and The Standard Template Library

4.1 Intro: The Standard Template Library

C++ comes with a bunch of data structures we can use to store sets of data. Each of these have different uses and work better for different designs. They all have documentation available online so you can learn more about how these structures work there as well.

4.1.1 Without the STL: Traditional fixed-length array

One of the first ways you may have learned to store a sequence of data is with a **traditional array** in C++. These aren't "smart" at all, cannot be resized, and you either have to manually keep track of the size of the array or do a calculation whenever you want to find the size.

1. Declaration

   They come in this form:

   ```
   const int ARRAY_SIZE = 100;
   int itemCount = 0;
   int myArray[ ARRAY_SIZE ];
   ```

   The `ARRAY_SIZE` named constant and the `itemCount` variable are optional, however, we have to manually keep track of the size of the array and how many items are stored within it - it won't take care of that for us.

2. Accessing elements

   Reading and writing values to this array is simple:

   ```
   cout << "Enter a number: ";
   cin >> myArray[ 0 ];
   cout << "Item #0 is " << myArray[0] << endl;
   ```

3. Iterating over elements

   And iterating through it requires, again, keeping track of the amount of items we've stored in the array with some kind of variable:

   ```
   for ( int i = 0; i < itemCount; i++ )
   {
     cout << i << " = " << myArray[i] << endl;
   }
   ```

4.1.2 Without the STL: Dynamic array

We can create **dynamic arrays** by using **pointers**. This allows us to define the size of the array at *runtime*, instead of defining the size at *compile-time* like with our traditional fixed-length array. The downside is manually managing the memory - making sure to free whatever we have allocated.

1. Declaration

   ```
   int itemCount = 0;
   int arraySize;

   cout << "Enter an array size: ";
   cin >> arraySize;

   int* myArray = new int[ arraySize ];
   ```

   Reading and writing values to the array is the same as with the traditional array. We just have to make sure to free the memory before the program ends.

2. Iterating over elements

   Just like with a traditional array we iterate over elements using a for loop, and we need to have an `itemCount` variable.

   ```
   for ( int i = 0; i < itemCount; i++ )
   {
     cout << i << " = " << myArray[i] << endl;
   }
   ```

3. Freeing memory

   You need to free the memory you allocate before its pointer loses scope - if this happens, you lose the address and that memory is now taken up and cannot be freed.

   ```
   delete [] myArray;
   ```

4.1.3 STL std::array

Documentation: https://cplusplus.com/reference/array/array/

he `array` from the STL is basically a class wrapping our traditional array. It allows us to use the array as an object with basic **functions**, so that we don't have to keep track of as much.

Declaration:
To create the array you need to specify the *data type* and *size* within the angle brackets.

```
array<int, 100> myArray;
```

1. Accessing the size

   The array object has a `.size()` function.

   ```
   cout << "Size: " << myArray.size();
   ```

2. Accessing elements

   Can access elements just like with a traditional array, using the subscript operator and an index.

   ```
   cout << "Enter item 0: ";
   cin >> myArray[0];
   ```

3. Iterating over elements

   Same sort of for loop, just use the size function to get the array size.

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

4.1.4 STL std::vector

Documentation: https://cplusplus.com/reference/vector/vector/
Vectors are **dynamic arrays** so you can add items to it and it will resize - no worries on your part regarding resizing and managing memory.

1. Declaration

   Specify the *data type* within the angle brackets. No size needed.

   ```
   vector<int> myVector;
   ```

2. Accessing the size

   Vector has a `.size()` function as well.

   ```
   cout << "Size: " << myVector.size();
   ```

3. Accessing elements

   Because you're generally not pre-allocating space for a vector, you need to use the `push_back` function to add additional items to the end of the vector's internal array. After there's an element in the vector, you can use the subscript operator to access an item.

   ```
   cout << "Enter item 0: ";
   int item;
   cin >> item;
   myVector.push_back( item );
   cout << "Item: " << myVector[0] << endl;
   ```

4. Iterating over elements

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

4.1.5 STL std::list

Documentation: https://cplusplus.com/reference/list/list/

The list is *similar* to a vector in that you can store any amount of items in it, however you can only ever access the *front* and the *back* of the list - not random items in the middle. You can still iterate over all the items, though.

1. Declaration

   ```
   list<int> myList;
   ```

2. Accessing the size

   ```
   cout << "Size: " << myList.size();
   ```

3. Adding data

   With a list you can add items to the *front* or the *back* of the list.

```
myList.push_front( 1 ); // Stored at the start of the list
myList.push_back( 10 ); // Stored at the end of the list
```

4. Accessing data

   You can't access an element in the middle of the list, but you can access the front or last elements.

```
cout << "Front item: " << myList.get_front() << endl;
cout << "Back item:  " << myList.get_back() << endl;
```

5. Removing data

   Similarly you can remove items at the front and back.

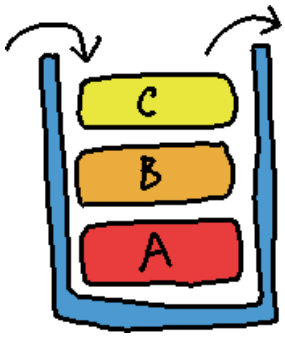
```
myList.pop_front();
myList.pop_back();
```

6. Iterating over elements

   You can use a range-based for loop to iterate over all of the items easily.

```
for ( auto& item : myList )
{
  cout << item << endl;
}
```

4.1.6 STL std::stack



Documentation: https://cplusplus.com/reference/stack/stack/

A stack is a type of restricted-access data type. It can store a series of items, but you can only add and remove items from the *top*.

1. Declaration

   ```
   stack<char> myStack;
   ```

2. Accessing the size

   ```
   cout << "Size: " << myStack.size();
   ```

3. Adding data

   The `push` function will add a new item to the *top* of the stack.

   ```
   myStack.push( 'A' );
   myStack.push( 'B' );
   myStack.push( 'C' );
   ```

4. Accessing data

   Only the *top*-most element of a stack can be accessed.

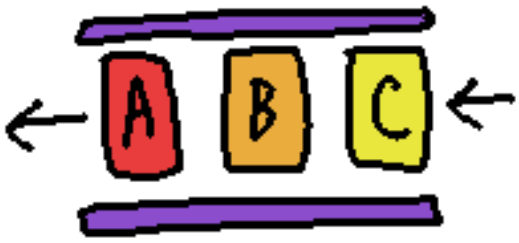
   ```
   cout << "Top item is: " << myStack.top() << endl;
   ```

5. Removing data

   This removes the *top*-most item of the stack.

   ```
   myStack.pop(); // Remove top item
   ```

4.1.7 STL std::queue



Documentation: https://cplusplus.com/reference/queue/queue/

A queue is another type of restricted-access data type. It stores a series of items, with items being added at the *back* of the queue, and being removed from the *front* of the queue, similar to waiting in line at the store.

1. Declaration

   ```
   queue<char> myQueue;
   ```

2. Accessing the size

   ```
   cout << "Size: " << myQueue.size() << endl;
   ```

3. Adding data

   The `push` function will add a new item to the *back* of the queue.

   ```
   myQueue.push( 'A' );
   myQueue.push( 'B' );
   myQueue.push( 'C' );
   ```

4. Accessing data

   Only the *front*-most item can be accessed in a queue.

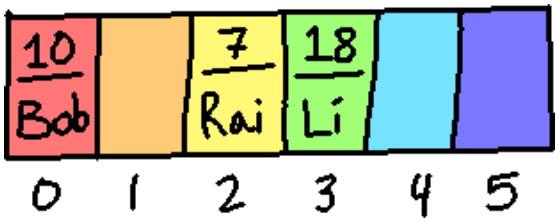
   ```
   cout << "The front item is: " << myQueue.front() << endl;
   ```

5. Removing data

   This removes the *front*-most item in the queue.

   ```
   myQueue.pop(); // Remove front item
   ```

4.1.8 STL std::map



Documentation: https://cplusplus.com/reference/map/map/

Our traditional arrays have **index** numbers (0, 1, 2, ...) and **elements** stored at that position in the array.

With a map, we can have any data type as a **unique identifier** for an **element**. This could be an integer, but it doesn't have to be 0, 1, 2, and so on - it could be an employee ID, a phone number, etc., but the data type of the identifier (called a **key**) can be any data type.

1. Declaration

   Specify the type of the *key* and the type of the *value*.

   ```
   map<int, string> area_codes;
   ```

2. Adding data

   Use the subscript operator with a *key*, and assign it a *value*.

   ```
   area_codes[913] = "Northeastern Kansas";
   area_codes[816] = "Northwestern Missouri";
   ```

3. Accessing the size

   ```
   cout << "Size: " << area_codes.size();
   ```

4. Accessing data

   You can access elements of the map via the *key*, if it exists in the map.

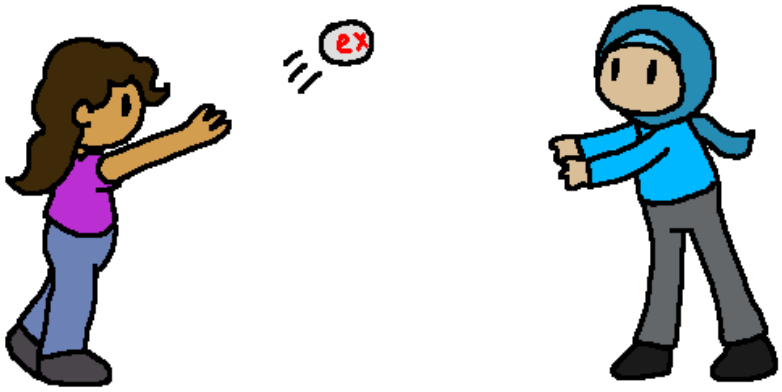
   ```
   int key;
   cout << "Enter an area code: ";
   cin >> key;
   cout << "The region for that is: " << area_codes[ key ];
   ```

5. Iterating over elements

   Use a range-based for loop to easily iterate over all of the elements. Use `item.first` to get the key and `item.second` to get the value.

   ```
   for ( auto& item : area_codes )
   {
     cout << "Key: " << item.first << ", Value: " << item.second << endl;
   }
   ```

4.2 Intro: Exceptions



4.2.1 Writing robust software

As a software developer, you will often be writing software that other developers will be using. This could be other developers at the company working on the same software product, or perhaps you might write a library that gets licensed out to other companies, or anything else. Your code will need to have checks in place for errors and be able to manage those errors gracefully, allowing the software to continue running instead of letting the program crash and restart.

A long time ago, lots of developers used numeric error codes to track errors. If you've ever seen something like "Error 12943" with no other useful information, that is an example of these error codes - useless for the end-user, but meant so that the programmer can search the code for that number and find where it broke. This is also why we use `return 0` at the end of our C++ programs - technically, you could return anything else, but returning 0 is meant to show that there were no errors. If you ran into an error, you *could* return 1 or 2 or 3 instead to mark errors.

Modern languages have **exception handling** built in, usually working with a **try/catch** style. You write in logic to check for errors, and when you find a problem you **throw** an exception, and that exception can be **caught** elseware in the code.

What kind of errors can we listen for?

- Trying to open a file that doesn't exist
- Memory access violations
- Invalid math (dividing by 0)
- Not enough memory
- Receiving unexpected inputs

- Trying to delete from an empty data structure
- Problem converting one data type to another

4.2.2 The C++ Exception object

C++ has an `exception` family of objects that we can use when trying to classify what kind of exception has happened. If none of the existing exception objects is appropriate, you can also create your own exception type.

Exceptions: (From http://www.cplusplus.com/reference/exception/exception/)

Exception class	Description
`bad_alloc`	Exception thrown on failure allocating memory
`bad_cast`	Exception thrown on failure to dynamic cast
`bad_exception`	Exception thrown by unexpected handler
`bad_function_call`	Exception thrown on bad call
`bad_typeid`	Exception thrown on typeid of null pointer
`bad_weak_ptr`	Bad weak pointer
`ios_base::failure`	Base class for stream exceptions
`logic_error`	Logic error exception
`runtime_error`	Runtime error exception
`domain_error`	Domain error exception
`future_error`	Future error exception
`invalid_argument`	Invalid argument exception
`length_error`	Length error exception
`out_of_range`	Out-of-range exception
`overflow_error`	Overflow error exception
`range_error`	Range error exception
`system_error`	System error exception
`underflow_error`	Underflow error exception
`bad_array_new_length`	Exception on bad array length

4.2.3 Detecting errors and throwing exceptions

The first step of dealing with exceptions is identifying a place where an error may occur - such as a place where we might end up dividing by zero. We write an **if statement** to check for the error case, and then **throw** an exception. We choose an exception type and we can also pass an error message as a string.

Example: "Risky" function detecting a divide by 0 error

```
int ShareCookies( int cookies, int kids )
{
  if ( kids == 0 )
  {
    throw runtime_error( "Cannot divide by zero!" );
  }
```

```
  return cookies / kids;
}
```

1. Listening for exceptions with try Now we know that the function `ShareCookies` could possibly throw an exception. Any time we call that function, we need to listen for any thrown exceptions by using **try**.

   Example: Calling the "Risky" function (e.g., from main)

   ```
   try
   {
     // Calling the function
     cookiesPerKid = ShareCookies( c, k );
   }
   ```

2. Dealing with exceptions with catch Immediately following the **try**, we can write one or more **catch** blocks for different types of exceptions and then decide how we want to handle it.

   Example: Calling the "Risky" function, catching an exception and handling it

   ```
   try
   {
     // Calling the function
     cookiesPerKid = ShareCookies( c, k );
   }
   catch( runtime_error ex )
   {
     // Display the error message
     cout << "Error: " << ex.what() << endl;

     // Handling it by setting a default value
     cookiesPerKid = 0;
   }

   cout << "The kids get " << cookiesPerKid << " each" << endl;
   ```

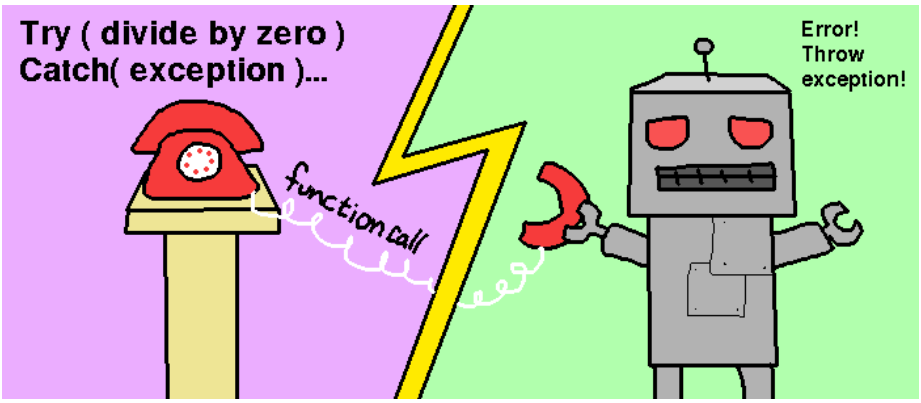
   A function could possibly throw different types of exceptions for different errors, and you can have multiple **catch** blocks for each type.

   Handling the error: Once you catch an error, it's up to you to decide what to do with it. For example, you could...

   - Ignore the error and just keep going
   - Write some different logic for a "plan B" backup
   - End the program because now the data is invalid

Coding with others' code: While writing software and utilizing others' code, you will want to pay attention to which functions you're calling that could throw exceptions. Often code libraries will contain documentation that specify possible exceptions thrown.

4.2.4 Common error in student implementations



A common error I see students make is to put the **try**, **catch**, and **throw** statements *all in one function*. This defeats the point of even using exceptions.

Common error: DON'T DO THIS!

```
float Divide( float num, float denom )
{
  try
  {
    if ( denom == 0 )
    {
      throw Exception;
    }

    return num / denom;
  }
  catch( Exception& ex )
  {
    // ...
  }
}
```

Remember that the **throw** belongs within a function that could cause an error, and the **try**/**catch** belongs with the location that **calls** the potentially exception-causing function.

Example Program

```
// FUNCTION THAT COULD CAUSE ERROR - Responsible for THROW
float Divide( float num, float denom )
{
  if ( denom == 0 ) {
    throw invalid_argument( "Can't divide by 0!" );
  }

  return num / denom;
}


int main()
{
  float n, d;
  cout << "Enter num: ";
  cin >> n;

  cout << "Enter denom: ";
  cin >> d;

  float result = 0;
  try // CALLING "IFFY" FUNCTION - Wrap the CALL in try/catch
  {
    result = Divide( n, d );
  }
  catch( Exception& ex )
  {
    cout << "ERROR OCCURRED!" << endl;
    return 1234;
  }

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

  return 0;
}
```

4.3 Lab: Exceptions and The Standard Template Library

4.3.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

4.3.2 Included files:

```
wk04_STLandExceptions
 graded_program
    FixedArray.h
    FixedArrayTester.cpp
    FixedArrayTester.h
    main.cpp
 instructions.html
 instructions.org
 practice1_stdexcept
    stdexcept.cpp
 practice2_customexcept
    customexcept.cpp
 practice3_array
    array.cpp
 practice4_vector
    vector.cpp
 practice5_list
    list.cpp
 practice6_map
```

```
   map.cpp
practice7_stack
   stack.cpp
practice8_queue
    queue.cpp
```

4.3.3 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 - Standard library exceptions

   With this program we have some risky functions that need us to check for error scenarios and throw an exception if found. We will also update main() to wrap the calls to those risky functions in try/catch statements.

   (a) Risky functions

      i. For the 'Divide' function check if 'denom' is 0. If this is the case, throw an 'invalid$_{\text{argument}}$' exception.

      ii. For the 'Display' function check if 'index' is invalid (less than 0 or greater than/equal to 'arr.size()'). If it's invalid, then throw an 'out$_{\text{ofrange}}$' exception.

      iii. For the 'PtrDisplay' function check if 'arr' is pointing to 'nullptr'. If it is, then throw an 'invalid$_{\text{argument}}$' exception.

   (b) Updating function calls

      i. Wrap the function call 'quotient = Divide(num, denom);' in a try/catch statement. This risky function might throw an 'invalid$_{\text{argument}}$' exception so that is what you'll listen for in the catch portion. When the exception is caught, display what the exception was ('ex.what()').

      ii. Wrap the function call 'Display(my$_{\text{array}}$, index);' in a try/catch statement. The catch should listen for an 'out$_{\text{ofrange}}$' exception. Display the error message when detected.

      iii. There are two calls to 'PtrDisplay', each should have **their own** try/catch blocks surrounding them. For both, listen for an 'invalid$_{\text{argument}}$' exception and display the error when detected.

   (c) Example output

```
--------------------------------------------------------------------------------
DIVISION EXAMPLE
Enter a numerator and denominator, separated by a space: 5 0
invalid_argument Exception: Division by 0 not allowed!

--------------------------------------------------------------------------------
DISPLAY EXAMPLE
Enter an index between 0 and 4: 10
out_of_range Exception: Invalid index!
```

```
--------------------------------------------------------------------------------
POINTER EXAMPLE
Display good_ptr...
Item being pointed to is: 10

Display bad_pointer...
invalid_argument Exception: ptr is null!
```

(d) Reference Exception family documentation: https://en.cppreference.com/w/cpp/error/exception

THROWing exceptions - Detect problem states and THROW as a result within the function that could cause problems.

```
void SOME_RISKY_FUNCTION( PARAMS )
{
  if ( BAD_SCENARIO )
  {
    throw SOME_EXCEPTION( "Message!" );
  }
  // ...
}
```

TRY/CATCH exceptions - Wrap the function call to the risky function in try, catch any possible exception types with catch.

```
try
{
  SOME_RISKY_FUNCTION( XYZ );
}
catch( const SOME_EXCEPTION& ex )
{
  cout << "Exception: " << ex.what() << endl;
}
```

---

2. Practice 2 - Custom exceptions

   At the top of the file declare two new exception classes - `NotEnoughFriendsException` and `NotEnoughPizzaException`. You can have these inherit from the `runtime_error` exception class (see reference for a template).

   Within these exceptions' constructors you will pass the `message` to the parent constructor, and within the constructor function body you can add additional instructions. In this case, we're just going to `cout` an extra message, though usually this would be used to clean up the program or log errors.

   - `NotEnoughFriendsException`: Display "You should really make more friends."
   - `NotEnoughPizzaException`: Display "You can't have a pizza party without pizza!"

(a) Risky function definition - 'int SlicesPerPerson(int friends, int pizzaSlices)'

This function does the math to figure out how many slices of pizza per person. However, since there is division, it's possible that we could end up with **divide by zero**. In this case (if `friends` is 0) we need to throw the `NotEnoughFriendsException`. You can also add an error message at this point, like "Zero friends at party!" to give more context as to why the exception was thrown.

Secondly, if there are 0 `pizzaSlices` at the party, while it won't cause an arithmetic error, it is a logic error for this pizza party planning program. If we detect this state, then we throw the `NotEnoughPizzaException` with an error message like "Zero pizza at party!".

(b) Calling the risky function in main()

Finally we need to make sure our program actually detects the exceptions and handles them when found. Currently, the function call is just: `slices = SlicesPerPerson(friendCount, pizzaSliceCount);`. We need to **wrap** this function call in a `try {}` statement.

The two exceptions that could be thrown are `NotEnoughFriendsException` or `NotEnoughPizzaException`, so we need two `catch` cases following the try statement. Within the catch code blocks I usually just `cout` the error message, which you can also do here. This would also be a good place to write the exceptions out to a log file if this were the real world.

(c) Example output

OK amount of friends / pizza:

```
PIZZA PARTY
How many pizza slices at pizza party? 50
How many friends at party? 4

Give each friend 12 slices of pizza
```

0 friends:

```
PIZZA PARTY
How many pizza slices at pizza party? 10
How many friends at party? 0

You should really make more friends.
Exception: Zero friends at party!
```

0 pizza:

```
PIZZA PARTY
How many pizza slices at pizza party? 0
How many friends at party? 10

You can't have a pizza party without pizza!
Exception: Zero pizza at party!
```

(d) Reference

Example of inheriting from 'runtime$_{\text{error}}$', with a constructor that calls the 'runtime$_{\text{error}}$' constructor:

```
class NEWEXCEPTION : public std::runtime_error
{
public:
  NEWEXCEPTION(std::string message)  // Constructor definition
    : std::runtime_error(message)    // This calls the parent class' constru
  {
    // Additional code
  }
};
```

THROWing exceptions - Detect problem states and THROW as a result within the function that could cause problems.

```
void SOME_RISKY_FUNCTION( PARAMS )
{
  if ( BAD_SCENARIO )
  {
    throw SOME_EXCEPTION( "Message!" );
  }
  // ...
}
```

TRY/CATCH exceptions - Wrap the function call to the risky function in try, catch any possible exception types with catch.

```
try
{
  SOME_RISKY_FUNCTION( XYZ );
}
catch( const SOME_EXCEPTION& ex )
{
  cout << "Exception: " << ex.what() << endl;
}
```

---

3. Practice 3 - Array

   For this program we're going to look at using the STL 'array' object.

   (a) Declare an array of strings of size 4 named 'course$_{\text{list}}$'.

   (b) Hard-code the items at index 0 through 3 with some course names (e.g., "CS 200").

   (c) Use a for loop to iterate over all the elements of the array, displaying each element's index and value.

   (a) Example output

```
ARRAY PROGRAM

1. Declare an array of strings of size 4...

2. Hard coding data into the array...

Iterating over the array to display each element index and value...
0: CS 134
1: CS 200
2: CS 235
3: CS 250

THE END
```

(b) Reference

array documentation: https://cplusplus.com/reference/array/array/

Declare an array object:

```
array<TYPE, SIZE> ARRAYNAME;
```

Access an element of the array:

```
ARRAYNAME[INDEX]
```

Get an array's size:

```
ARRAYNAME.size()
```

Iterate over an array:

```
for ( size_t i = 0; i < ARRAY.size(); i++ )
{
    // i is the index, ARRAY[i] is the value
}
```

---

4. Practice 4 - Vector

For this program we are using the STL vector which is basically a dynamic array.

(a) At the beginning of the program declare a vector of strings named `course_list`.

(b) Within the while loop after the if statement the user has entered a course name in the `input` variable. Push this item into the `course_list` using the vector's `push_back` functionality.

(c) Near the end of the program use a for loop to iterate over all the elements of the vector. Display each element's index and value.

(a) Example output

```
VECTOR PROGRAM

Declare a vector of strings...
Enter a new course, or STOP to finish: CS 134

Pushing new item into the vector...
Enter a new course, or STOP to finish: CS 200

Pushing new item into the vector...
Enter a new course, or STOP to finish: CS 235

Pushing new item into the vector...
Enter a new course, or STOP to finish: STOP

Iterating over the vector to display each element's index and value...
0: CS 134
1: CS 200
2: CS 235

THE END
```

(b) Reference vector documentation: https://cplusplus.com/reference/vector/vector/

Declare a vector object:

```
vector<TYPE> VECNAME;
```

Add an item to the vector:

```
VECNAME.push_back( VALUE );
```

Get the size of a vector:

```
VECNAME.size()
```

---

5. Practice 5 - List

For this program we will be using the STL list object.

(a) Near the start of the program declare a list of strings named `course_list`.

(b) After the user enters a `position` if they selected "F" then use the list's `push_front` function to add the `input` to the front of the `course_list`.

(c) Otherwise, if the user selected "B" then use the list's `push_back` function to add the `input` to the back of the `course_list`.

(d) Before the program ends, iterate through all the elements of the list, displaying each value.

(a) Example output

```
LIST PROGRAM

Declare a list of strings...
Enter a new course, or STOP to finish: CS 134
Insert at (F) FRONT or (B) BACK? F

Pushing new item to front of list...
Enter a new course, or STOP to finish: CS 200
Insert at (F) FRONT or (B) BACK? F

Pushing new item to front of list...
Enter a new course, or STOP to finish: CS 235
Insert at (F) FRONT or (B) BACK? B

Pushing new item to back of list...
Enter a new course, or STOP to finish: STOP

Iterating over the list to display each element's  value...
CS 200
CS 134
CS 235

THE END
```

(b) Reference list documentation: https://cplusplus.com/reference/list/

Declare a list:

```
list<TYPE> LISTNAME;
```

Add a new item to the front of the list:

```
LISTNAME.push_front( VALUE );
```

Add a new item to the back of the list:

```
LISTNAME.push_back( VALUE );
```

Get the size of a list:

```
LISTNAME.size()
```

Iterate over all the items in a list: #+BEGIN$_{\text{SRC}}$ form for (TYPE item : LISTNAME) { // item is the current element }

---

6. Practice 6 - Map

For this program we will be using the STL map.

(a) Near the top of the program declare a map with string keys and float values, the variable name should be `product_prices`.

(b) Afterwards, hard-code 3 products, use the product name as the KEY and the price of the product as a VALUE. (e.g., "burrito" and 1.29).

(c) Use a for loop to iterate over all of the elements of `product_prices`, display each element's key and value.

(d) After the user has entered a food, display the price of the item by accessing the element of `product_prices` with `choice` as the key.

(a) Example output

```
MAP PROGRAM

Declare a map with string keys and float values...

Set up 3 product names and their prices in the map...

Iterating over the map to display each element's key and value...
Item: burrito, Price: $1.29
Item: quesadilla, Price: $2.36
Item: taco, Price: $1.29

Enter a food: burrito
The price of this item is... $1.29

THE END
```

(b) Reference map documentation: https://cplusplus.com/reference/map/map/

Declare a map with TYPE1 key and TYPE2 value:

```
map<TYPE1, TYPE2> MAPNAME;
```

Add a (key, value) pair to the map:

```
MAPNAME[KEY] = VALUE;
```

Get the amount of items in the map:

```
MAPNAME.size()
```

Iterate over all of the pairs in a map:

```
for ( auto item : product_prices )
{
    // KEY: item.first
    // VALUE: item.second
}
```

Get the value of an item from a map given some key:

```
MAPNAME[ KEY ]
```

---

7. Practice 7 - Stack

   For this program we will use the STL stack. A stack structure is a "first-in, last-out" structure, where the first item ends up at the bottom of the stack and all newer items are stacked on top of it.

(a) Near the start of the program declare a stack of strings named `my_stack`.

(b) Within the while loop, check if the stack is empty. If it is, display "STACK IS EMPTY". Otherwise, display the TOP item of the stack. Use the `top` function to do this.

(c) During case 1 of the switch statement the user wants to add `text` to the stack. Use the stack's `push` function to do this.

(d) During case 2 of the switch statement the user wants to pop the top-most item off the stack. Use the stack's `pop` function to do this.

(e) After the while loop is over we will display each item of the stack, popping each item after we've seen it. While the stack is not empty, do the following: 5a. Display the item at the top of the stack. 5b. Pop the item at the top of the stack.

(a) Example output

```
STACK PROGRAM

Declaring a stack of strings...
-------------------------------------------------
STACK IS EMPTY
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 1
Enter new text to push on stack: C

Pushing the new item onto the stack...
-------------------------------------------------
TOP ITEM IN STACK: C
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 1
Enter new text to push on stack: A

Pushing the new item onto the stack...
-------------------------------------------------
TOP ITEM IN STACK: A
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 1
Enter new text to push on stack: T

Pushing the new item onto the stack...
-------------------------------------------------
```

```
TOP ITEM IN STACK: T
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 2

Popping the top item off the stack...
-------------------------------------------------
TOP ITEM IN STACK: A
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 0

-------------------------------------------------

Iterate over stack until it's empty... show the top item then pop it...
TOP OF STACK: A
TOP OF STACK: C

THE END
```

(b) Reference stack documentation: https://cplusplus.com/reference/stack/stack/

Declare a stack:

```
stack<TYPE> STACKNAME;
```

Push an item to the top of the stack:

```
STACKNAME.push( VALUE );
```

Access the item at the top of the stack:

```
STACKNAME.top()
```

Pop an off the top of the stack:

```
STACKNAME.pop();
```

Get the total amount of items stored in the stack:

```
STACKNAME.size()
```

Check if a stack is empty:

```
STACKNAME.empty()
```

---

8. Practice 8 - Queue

   For this program we will use the STL queue. A queue structure is a "first-in, first-out" structure, where the first item will be at the front of the queue and items that enter the queue after it must wait for the front-most item to get finished before they can be accessed.

(a) Near the start of the program declare a queue of strings named `my_queue`.

(b) Within the while loop, check if the queue is empty. If it is, display "QUEUE IS EMPTY". Otherwise, display the FRONT item of the queue. Use the `front` function to do this.

(c) During case 1 of the switch statement the user wants to add `text` to the queue. Use the queue's `push` function to do this.

(d) During case 2 of the switch statement the user wants to pop the front-most item out of the queue. Use the queue's `pop` function to do this.

(e) After the while loop is over we will display each item of the queue, popping each item after we've seen it. While the queue is not empty, do the following: 5a. Display the item at the front of the queue. 5b. Pop the item at the front of the queue.

(a) Example output

```
QUEUE PROGRAM

Declaring a queue of strings...
-------------------------------------------------
QUEUE IS EMPTY
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 1
Enter new text to push on queue: C

Pushing the new item into the queue...
-------------------------------------------------
FRONT ITEM IN QUEUE: C
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 1
Enter new text to push on queue: A

Pushing the new item into the queue...
-------------------------------------------------
FRONT ITEM IN QUEUE: C
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 1
Enter new text to push on queue: T
```

```
Pushing the new item into the queue...
-------------------------------------------------
FRONT ITEM IN QUEUE: C
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 2

Popping the front item out of the queue...
-------------------------------------------------
FRONT ITEM IN QUEUE: A
-------------------------------------------------
0. Quit
1. PUSH item
2. POP item
>> 0

-------------------------------------------------

Iterate over queue until it's empty... show the front item then pop it...
FRONT OF QUEUE: A
FRONT OF QUEUE: T

THE END
```

(b) Reference queue documentation: https://cplusplus.com/reference/queue/queue/

Declare a queue:

```
queue<TYPE> QUEUENAME;
```

Push an item to the back of the queue:

```
QUEUENAME.push( VALUE );
```

Access the item at the front of the queue:

```
QUEUENAME.front()
```

Pop an from the front of the queue:

```
QUEUENAME.pop();
```

Get the total amount of items stored in the queue:

```
QUEUENAME.size()
```

Check if a queue is empty:

```
QUEUENAME.empty()
```

4.3.4 Graded programs

1. Graded program - Fixed Array structure

   This program is a continuation of your `wk02_TestDebugFriendTemplate/graded_program`. You can copy/paste your function implementations into `FixedArray.h`.

   (a) Adding error checks and exceptions

   Previously, without exceptions, if we encountered an error state we would have to just ignore it, return some bad "default" data, or let the program crash. Now we can fix up the functions to use exceptions when an error state occurs.

   i. `void FixedArray<T>::PushBack( T newItem )` - If the array is full, then throw a `length_error`.
   ii. `void FixedArray<T>::PopBack()` - If the array is empty, then throw a `runtime_error`.
   iii. `T& FixedArray<T>::GetBack()` - If the array is empty, then throw a `runtime_error`.
   iv. `T& FixedArray<T>::GetFront()` - If the array is empty, then throw a `runtime_error`.
   v. `T& FixedArray<T>::GetAt( size_t index )`
      A. If the array is empty, then throw a `runtime_error`.
      B. If the `index` is out of range, then throw an `out_of_range` exception.
   vi. `size_t FixedArray<T>::Search( T item ) const` - If the item isn't found, then throw a `runtime_error`.

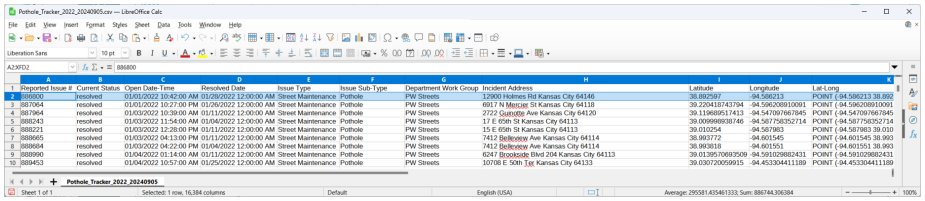
   (b) Bonus points - Updating the tests

   If you would like to udpate your tests from the wk02 version, you can now add checks to make sure exceptions are thrown when an error state occurs. You would surround the "risky" function calls in `try` and listen for the appropriate exception, or `...` to catch all exceptions...

```
FixedArray<int> testArray;
testArray.m_itemCount = 0;

bool expectOut_exception = true;
bool actualOut_exception = false;

try
{
    testArray.PopBack();
}
catch( const runtime_error& ex )
{
    actualOut_exception = true;
    cout << ex.what() << endl;
}

// If expected out matches actual out...
```

4.4 Project: Part 1

4.4.1 Introduction



For the project this semester you will be working with some kind of data set. For this first little assignment you will need to find a data file you'd like to work with and upload it for review and approval. Your data file should be in CSV (comma-separated-value) format that has at least 10,000 records (rows) in it. Make sure to look at what fields (columns) are present and think about how you might filter and sort the data, and what kind of reports might be generated (e.g., "all potholes in XYZ zipcode", "all potholes NOT in resolved status").

4.4.2 Finding data

Some suggestions for open data sources are:

- U.S. data.gov: https://data.gov/
- KC open data: https://data.kcmo.org/

You can also look in other locations for data.

4.4.3 Turn-in

- Provide the URL where you found the data file (I should be able to also download the CSV file and review it.)
- In the comments:
  - Tell me which **columns** from the CSV file you are going to analyze.
  - What do you find interesting about this data set? What might you want to analyze?

As an example, given the **potholes** data set, the analysis could be something like this:

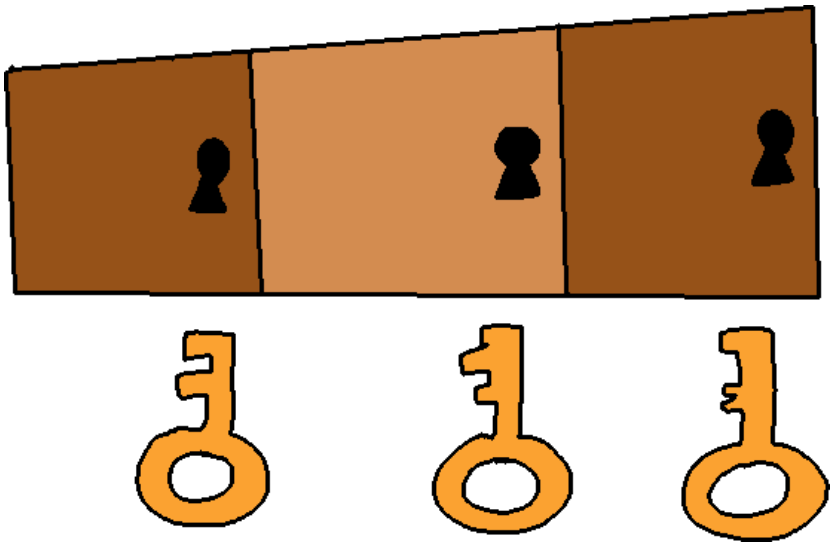
1. Comparing the total amount of reported issues per area - which location has the highest amount of potholes?
2. Comparing the total amount of RESOLVED issues vs. RECEIVED issues and their locations - which location has the highest % of reported but not fixed potholes, and which location has the highest % of reported AND fixed potholes?

After analyzing location statistics related to road repair, we could also look at a map of historically redlined districts in the KC Metro and look at how these may correspond, or look at some other data on income taxes (lots of income for fixing roads?), total amount of people (lots of people driving?), total amount of businesses in the area (heavy commuting?), etc.

5 Week 5: Hash Table structure

5.1 Intro: Hash Table structures

5.1.1 Hash tables, or Dictionary, or Map, or Associative Array...

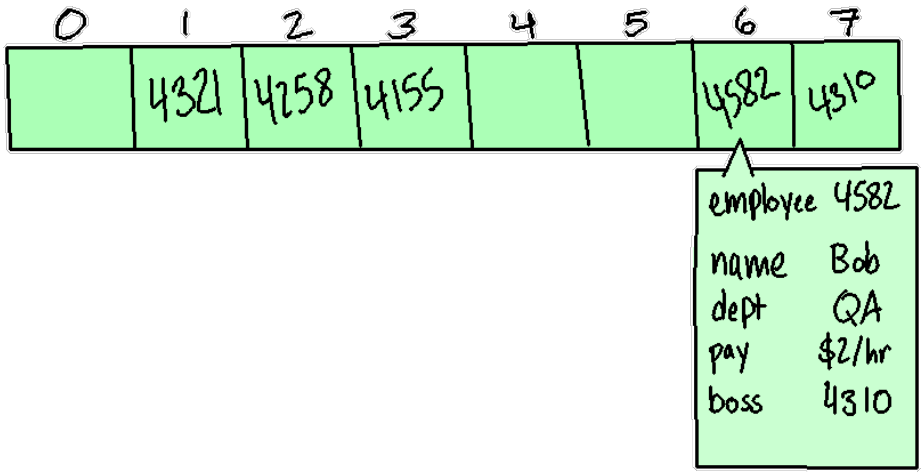


As you might remember with `std::map` in C++ or something like a **Dictionary** in Python, sometimes we don't just want a series of elements in an array with indices like 0, 1, 2, 3, ... etc. Why use positional indices when we can also use the "index" to store useful data?

Instead, we can store data as **key/value** pairs, where some key (unique identifier) is mapped to some value.

The Key and the Value can each be any data type, and they can be different data types from each other. Often, the key will be a basic data type like an integer or string, while the value will be a class that contains more data.

Hash Tables are how we can implement our own Map or Dictionary type.



With this, we utilize a Hashing Function to convert a unique key into an index of an array.

Because we're taking a key, doing a simple math operation, and getting an index from that, the best case access-time for a Hash Table is O(1)... However, sometimes different keys might map to the same index - that's where collision strategies come into play.

1. Key-value pairs

   Dictionaries are often thought of as "key-value pairs," where we have some data to store (this is the value), and we can look it up by its key (some searchable identifier).

   When designing a class around this, we can reuse some piece of information as the **key**, as long as it's going to be unique in the table.

Employee ID (key)	**Employee object (value)**
1024	Name: Amina, Employee ID: 1024, Department: Dev
2048	Name: Benita, Employee ID: 2048, Department: Sales
1280	Name: Caiyun, Employee ID: 1280, Department: QA

   The **key** should be a datatype that can be easily put into a mathematical function - such as an integer. Strings can also be used, but a complex data type like a class wouldn't work very well as a key.

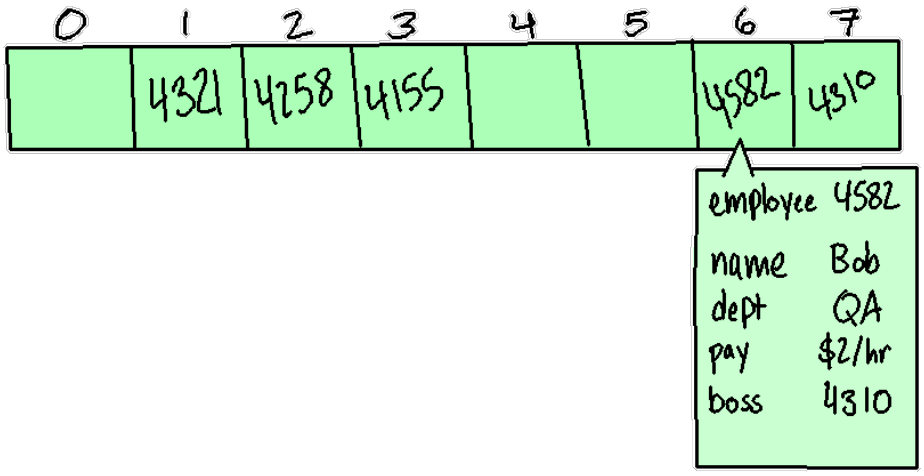
   The **value** can be any data type, usually a specific class that stores more data.

   Some examples of key-value uses are:

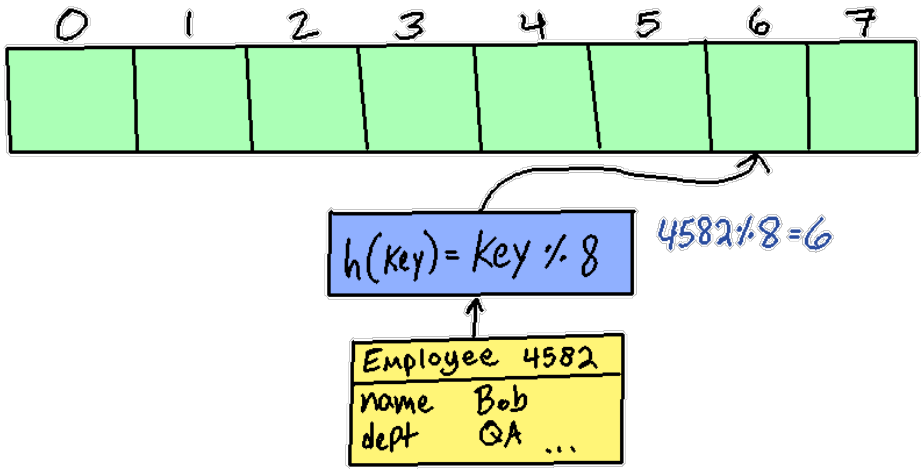
   - People lookup - Associating an employee ID, student ID, etc. (key) to an Employee or Student object (value).

- Address book - Associating a phone number (key) with a contact entry (value).
- Dictionaries - searching for a word (key), finding the definition (value).
- Book lookup - Associating an ISBN (key) to a Book object (value).
- Word counter - Associating each word (key) with the amount of times it has shown up in a document (value).

2. **Hashing the Key to get the index** With our Hash Tables, each **element** in the array will be associated with a **key** - similar to our Binary Search Trees. The **key** is a unique identifier, allowing us to access a set of data mapped to the key.



The way we do this is we use this **key** as the input of a function that converts the key into an **index** - somewhere from 0 (the beginning of the array) to *(n-1)* (the end of the array). This is known as the **hash function**. A simple hash function could be taking the key and using the modulus operator with the size of the array to get an index.



Because of this hash function, the efficiency of both **inserting** and **searching** is *(O(1))* - unless there are **collisions**, where multiple keys map to the same index. This also means that, with minimal collisions, the speed to insert/search is going to be near instantaneous whether we have 10 items or 10,000 items.

(a) Hashing functions

The simplest hashing function will just take the **key** and the **size of the array** and use modulus to find a resulting index:

```
Hash(key) = key % ARRAY_SIZE
```

Using the modulus operator restricts the resulting indices so that they will be between `0` and `ARRAY_SIZE-1`. For example, if we had an array of size 5 and were generating indices, everything would be between 0 and 4:

index	0	1	2	3	4
key	0	1	2	3	4
	5	6	7	8	9
	10	11	12	13	14
	15	16	17	18	19

Looking at the table above, if we had something with the key 5 and the key 10, this would cause a **collision** - they would both map to the index position 0. If we do encounter a collision, we will need a strategy to find a new index – more on that later.

As an example of a hash function, let's say we're going to store employees in a Hash Table by their unique employee IDs. Employee IDs begin at 1000 (not 0 like the array index) and can be any 4-digit number. They're not sequential. We want to assign an array index to each employee ID so we can quickly search for an employee by their ID. For this, we need a hash function. The simple hash function would look like this:

```
// Input: employee ID integer (key)
// Output: array index
int Hash(int employeeId) {
    return employeeId % ARRAY_SIZE;
}
```

(b) Collision strategies

Sooner or later, a collision will occur, and you will need a way to take care of it - basically, to generate a different index than the one that our hash function gives us. There are several ways we can design our hash tables and collision strategies. We're going to cover 3 in this course:

- **Linear Probing:** Just move forward by 1 index until an available space is found. The form here would generally be:
  `newIndex = originalIndex + collisionCount`
- **Quadratic Probing:** Move forward by collisions2 until an available space is found. The form here would generally be:
  `newIndex = originalIndex + ( collisionCount * collisionCount )`
- **Double Hashing:** Use a secondary hash function and the collision count to compute a new index. The form here would generally be:
  `newIndex = Hash1( key ) + collisionCount * Hash2( key )`

i. Linear Probing (Open Addressing) With linear probing, the solution to a collision is simply to go forward by one index and see if that position is free. If not, continue stepping forward by 1 until an available space is found.

A. Initial hash table:

index	0	1	2	3	4
value		6	12		

B. Insert new employee ID 16...
`H(16) = 16 % 5 = 1`
COLLISION: Index 1 is already taken!

C. Move forward by 1...
`H(16) + 1 = (16 % 5) + 1 = (1) + 1 = 2`
COLLISION: Index 2 is already taken!

D. Move forward by 1...
`H(16) + 2 = (16 % 5) + 2 = (1) + 2 = 3`
Index 3 is available!

index	0	1	2	3	4
value		6	12	16	

As an example of a hash function, let's say we're going to store employees in a Hash Table by their unique employee IDs. Employee IDs begin at 1000 (not 0 like the array index) and can be any 4 digit number. They're not sequential. We want to assign an array index to each employee ID so we can quickly search for an employee by their ID. For this, we need a hash function. The simple hash function would look like this:

```
// Input: employee ID integer (key)
// Output: array index
int Hash( int employeeId )
{
    return employeeId % ARRAY_SIZE;
}
```

ii. Quadratic Probing (Open Addressing) With quadratic probing, we will keep stepping forward until we find an available spot, just like with linear probing. However, the amount we step forward by changes each time we hit a collision on an insert.

A. Initial hash table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value		14	41			18							

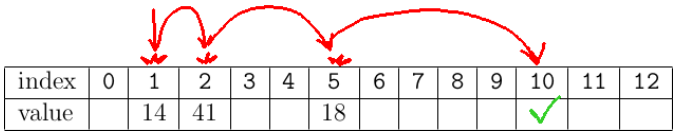
B. Insert new employee ID 27...
`H(27) = 27 % 13 = 1`
COLLISION: Index 1 is already taken!

C. Collisions = 1, shift over by $collisions^2$...
`H(27) + 1^2 = (27 % 13) + 1^2 = (1) + 1^2 = 2`
COLLISION: Index 2 is already taken!

D. Collisions = 2, shift over by $collisions^2$...
`H(27) + 2^2 = (27 % 13) + 2^2 = (1) + 2^2 = 5`
COLLISION: Index 5 is already taken!

E. Collisions = 3, shift over by $collisions^2$...
`H(27) + 3^2 = (27 % 13) + 3^2 = (1) + 3^2 = 10`
Index 10 is available!

Each time there's a collision, we search further from the original hashed index: +1 space, then +4 spaces, then +9 spaces.



The probing function would look something like this:

```
int QuadraticProbe( int originalIndex,
   int collisionCount )
{
  return originalIndex + pow( collisionCount, 2 );
}
```

iii. Double Hashing (Open Addressing) With double hashing, we have two hash functions. If the first hash function returns an index that is already taken, then we use the second hash function on the key to generate an offset, instead of just using a linear or quadratic formula.
With this method, our new index after a collision will be:
$newIndex = Hash(key) + collisionCountHash2(key)$

It is up to our design to figure out a second hash function, but generally it should also work with prime numbers. For example:

```
int Hash( int key )
{
  return key % ARRAY_SIZE;
}

int Hash2( int key )
{
  return 7 - ( key % 7 );
}
```

iv. Additional considerations
Index wrap-around Note that with all of these operations, we will also do an additional operation of `index % ARRAY_SIZE` to make sure the index stays within the bounds of the array.

The problem of clustering Clustering is a problem that arises, particularly when using linear probing. If we're only stepping forward by 1 index every time there's a collision, then we tend to get a Dictionary table where all the elements are clustered together in groups.

index	0	1	2	3	4	5	6	7	8	9
value		A	B	C			X	Y	Z	

Because of this, the more full the Hash Table gets, the less efficient it becomes when using linear probing.

Sizing a Hash Table Array When we're writing a hash function, we will want to take into account how likely a collision will be. A general design rule is that making your array size a prime

number helps reduce collisions.

Pushing to a Hash Table When Pushing a new { key : value } pair to a hash table first we try to use the Hash function to turn key into an index. If there is no collision, then we're good and we insert the new item to the table with that index.
However, while there is a collision, we need to add 1 to a collision counter and generate a new index. As long as that position in the table is taken, we encounter another collision and repeat.

Getting from a Hash Table
Similarly for the Get function we continue looping to find the index for our key. In this case, we're looping while we see a taken index but that item's key doesn't match. This means our loop ends if: (1) We've found an empty spot in the array (so our key isn't in the hash table at all), or (2) We've found an index where our key matches.
In the case where we find our desired key, then we return the value at that position, the array[index].

(c) Hash Table Efficiency In an ideal world with no collisions, a Hash Table would have an $O(1)$ growth rate for its Search, Insert, and Deletion no matter how many items it stores. However, since collisions may occur, the efficiency of the Hash Table can vary.

Pros

- **Searching and Inserting is relatively quick:** Since we use hash functions and collision functions to find the index to insert at and to access, it is just one math operation (at best) to find an index, and with collisions, hopefully just a few more calculations.
- **Mapping a key to a value is useful:** With a normal Array vs. Linked List, there's not really additional logic that goes into the storage - we just need to store data and it is thrown into a structure. With a Hash Table, we can have a more intentional design for our data.

Cons

- **Many things affect Hash Table efficiency:** The efficiency of finding an index can vary based on the size of the array, how full it is, the quality of the hashing function and collision strategy, and even the data being stored in it.
- **A bad hash function can cause more collisions:** Different hash functions can affect how many collisions occur, so different functions can cause different speeds.
- **They are difficult to resize:** If we were to resize a hash table, not only would we have to copy all the data over (which would be $O(n)$), we would also need to rehash all of the keys to get

new indices based on the new table size. Because resizing a hash table is expensive, if we can reasonably estimate the needed size of the hash table ahead of time, that only helps us out. If we under-estimate the size, that can be costly in having to resize things.

3. See Also

   You can find out about more hashing techniques by looking on Wikipedia.

   - Separate chaining
   - Coalesced hashing
   - Robin-hood hashing

5.2 Lab: Hash Table structure

5.2.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

5.2.2 Assignment information

Getting started:

1. This lab is located in your **repository folder** under `HashTableStructure`.
2. In VS Code use "Open Folder" to open the specific subprogram you wish to work with.
3. BUILD: Since this is a big project, use the `make` command (in the same directory as the Makefile) to build the program.
4. RUN: `./debug.exe` or `gdb ./debug.exe` for the debug version.

~

Turning in your work:
To turn in the assignment, make sure you've **committed** and **synced (pushed)** your changes to your GitLab repository. Double check the file from the GitLab webpage to ensure it's up-to-date. Then, submit the URL to the folder for the week.

~

Working on this assignment:
After building the program you can run it with the following arguments:

- `./debug.exe hashtable` - Run tests for hash table

- Array Structures reference:
  - Textbook, archived class lecture
  - Lecture sildes
- Quick references:
  - Using Git and VS Code
  - Program arguments
- Assignment direct link

5.2.3 Included files:

```
HashTableStructure
 DataStructures
    HashTable
       HashItem.h
       HashTable.h
       HashTableTester.cpp
       HashTableTester.h
    SmartTable
        SmartTable.h
        SmartTableNode.h
        SmartTableTester.cpp
        SmartTableTester.h
 Exceptions
    InvalidIndexException.h
    ItemNotFoundException.h
    NotImplementedException.h
    NullptrException.h
    StructureEmptyException.h
    StructureFullException.h
 Makefile
 Utilities
    StringHelper.cpp
    StringHelper.h
    Style.cpp
    Style.h
 instructions.org
 main.cpp
```

5.2.4 Instructions

1. int HashTable<T>::Hash1(int key)

   This function takes in a `key`, runs it through a hash function, and returns an `index` that corresponds to the array.

The hash function we're using here is: `key % m_table.ArraySize()` (where `%` means modulus). Return the result.
~

2. int HashTable<T>::LinearProbe(int originalIndex, int collisionCount)

   If a collision has happened, this takes the `originalIndex` assigned the key and the `collisionCount` - the amount of collisions that has happened. It will then generate a new index, offset using the Linear Probe strategy.

   Return: The `originalIndex` plus the `collisionCount`.
   ~

3. int HashTable<T>::QuadraticProbe(int originalIndex, int collisionCount)

   If a collision has happened, this takes the `originalIndex` assigned the key and the `collisionCount` - the amount of collisions that has happened. It will then generate a new index, offset using the Quadratic Probe strategy.

   Return: The `originalIndex` plus the value of `collisionCount` squared (you can use `collisionCount * collisionCount` instead of having to include any math libraries.).
   ~

4. int HashTable<T>::Hash2(int key, int collisionCount)

   If a collision has happened, this takes the item's `key` and the `collisionCount` - the amount of collisions that has happened. It will then generate a new index using the main Hash function `Hash1` and the secondary hash function `Hash2` to generate an offset.

   Here, our second hash is `3 - key % 3`

   Return: original index (`Hash1(key)`) plus the step forward (`collisionCount * (3 - key % 3)`).
   ~

5. void HashTable<T>::Push(`size_t` key, T data)

   For this function we have received a new piece of `data` to store in our array, as well as a `key` for that data. Our HashTable structure uses an underlying `SmartTable<HashItem<T>> m_table` to store its data (I've already implemented the SmartTable). The SmartTable allows gaps between indices in the array. The `m_method` member variable should have already been set to some collision method as well: `CollisionMethod::LINEAR`, `CollisionMethod::QUADRATIC`, and `CollisionMethod::DOUBLE_HASH`.

   CUSTOM EXCEPTION TYPES: I have a custom exception type `Exception::StructureFullException( string, string )`. Notice that it takes TWO strings instead of one like the standard library's exceptions. The first string should be the FUNCTION NAME and the second string should be the ERROR MESSAGE.

   - **Error check:** If the table is full (use the table's `IsFull()` function), then throw a `StructureFullException`.

- Declare an integer variable named `collisionCount` and initialize it to 0.
- Declare an integer variable named `originalIndex` and set it equal to the result of `Hash1`, passing in the `key`. (If a collision happens, we will use this as the starting point, and then find an offset.)
- Declare an integer named `newIndex` and set it to `originalIndex`. (We'll use this if there is a collision; we don't want to overwrite the original index.)
- When we call `m_table.ItemAtIndex( newIndex )`, it should return `true` if something is already there (COLLISION!) or `false` otherwise. If it's false, we can skip the collision loop and insert our new item at the index found. If there is a collision, then we're going to need to find a new index based on our collision strategy.
- WHILE COLLISION ON `newIndex`, DO THE FOLLOWING:
  - Increment `collisionCount` by 1.
  - If the collision method is `LINEAR`, then set `newIndex` to the result of the LinearProbe, passing in `originalIndex` and `collisionCount`. Also modulus this by the `m_table.ArraySize()` to keep it within bounds of the array.
  - Otherwise if the collision method is `QUADRATIC`, then set `newIndex` to the result of the QuadraticProbe, passing in `originalIndex` and `collisionCount`. Also modulus this by the `m_table.ArraySize()` to keep it within bounds of the array.
  - Otherwise if the collision method is `DOUBLE_HASH`, then set `newIndex` to the result of calling Hash2, passing in `key` and `collisionCount`. Also modulus this by the `m_table.ArraySize()` to keep it within bounds of the array.
- After the WHILE LOOP is over, then we can add the new item:
- Create a new `HashItem<T>`.
- Set your item's `data` to the data passed in as a parameter.
- Set your item's `key` to the key passed in as a parameter.
- Call `m_table.PushAt`, passing in your new item and the `newIndex`.

~

6. T& HashTable<T>::Get(int key)

   This function works similarly to the Push function, we are still using hashing to find the position of an item we're searching for, and if there's something at the found position but it doesn't match, then we have a collision and need to keep looking for our item.

   - Declare an integer variable named `collisionCount` and initialize it to 0.
   - Declare an integer variable named `originalIndex` and set it equal to the result of `Hash1`, passing in the `key`. (If a collision happens, we will use this as the starting point, and then find an offset.)

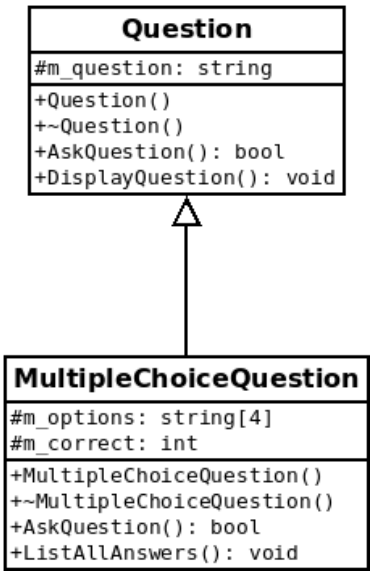
- Declare an integer named `newIndex` and set it to `originalIndex`. (We'll use this if there is a collision; we don't want to overwrite the original index.)
- If the `Size()` is 0, then throw a `StructureEmptyException`. (Remember to pass in two strings - funciton name and error message!)
- A collision here counts if there is an ITEM AT THE INDEX `newIndex`, AND its KEY DOESN'T MATCH.
- WHILE `m_table.ItemAtIndex( newIndex )` is true and `m_table.GetAt( newIndex ).key` doesn't match our key parameter, do the following:
  - Increment `collisionCount` by 1.
  - If the collision method is `LINEAR`, then set `newIndex` to the result of the LinearProbe, passing in `originalIndex` and `collisionCount`. Also modulus this by the `m_table.ArraySize()` to keep it within bounds of the array.
  - Otherwise if the collision method is `QUADRATIC`, then set `newIndex` to the result of the QuadraticProbe, passing in `originalIndex` and `collisionCount`. Also modulus this by the `m_table.ArraySize()` to keep it within bounds of the array.
  - Otherwise if the collision method is `DOUBLE_HASH`, then set `newIndex` to the result of calling Hash2, passing in `key` and `collisionCount`. Also modulus this by the `m_table.ArraySize()` to keep it within bounds of the array.
- After the WHILE LOOP is over:
- If there is an ItemAtIndex `newIndex` and the GetAt `newIndex` item's key matches our key parameter, then return `m_table.GetAt( newIndex ).data`.
- Otherwise, throw an `Exception::ItemNotFoundException`. (Make sure to pass two strings!)

6 Week 6: Polymorphism and static members

6.1 Intro: Polymorphism

6.1.1 Reviewing classes and pointers

1. Review: Class inheritance and function overriding



   Some things to remember about inheritance with classes:

   - Any public or protected members (functions and variables) are inherited by the child class. (e.g., `m_question`, `DisplayQuestion()`, and `AskQuestion()`).
   - A child class can override the a parent's function by declaring and defining a function with the same signature. (e.g., `AskQuestion()`).
   - If the child class doesn't override a parent's function, then when that function is called via the child object it will call the parent's version of that function. (e.g., `DisplayQuestion()`).

2. Review: Pointers to class objects

   You can declare a pointer to point to the address of an existing object, or use the pointer to allocate memory for one or more new instances of that class...

   - Pointer to existing address:

```
myPtr = &existingQuestion;
```

- Pointer to allocate memory:

```
myPtr = new Question;
```

Then, to access a member of that object via the pointer, we use the `->` operator, which is equivalent to dereferencing the pointer and then accessing a member:

- Arrow operator:

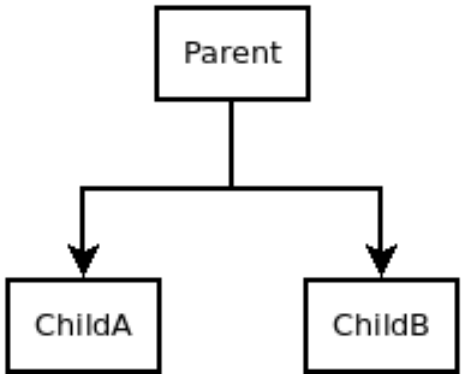
```
myPtr->DisplayQuestion();
```

- Dereference and access:

```
(*myPtr).DisplayQuestion();
```

6.1.2 Design and polymorphism

So much of the design tricks and features we utilize in C++ and other object-oriented programming languages all stem from the concept of "do not repeat yourself". If you're writing the same set of code in multiple places, there is a chance that we could design the program so that we only need to write that code once.



Polymorphism is a way that we can utilize pointers and something called **vtables** to have a family of classes (related by inheritance) and be able to write one set of code to handle interfacing with *all of those family members*. We have a family tree of classes, and we can write our program to treat all the objects as the **parent class**, but the program will decide which set of functions to call at run time.

Example: Using Polymorphism to treat ChildA and ChildB objects as their Parent object

```
Parent* myPtr = nullptr;
if      ( type == 1 ) { myPtr = new ChildA; }
else if ( type == 2 ) { myPtr = new ChildB; }
```

```
myPtr->Display();
delete myPtr;
```

1. Example: Quizzer and multiple question types

   Let's say we are writing a quiz program and there are different types of questions: True/false questions, multiple choice, and fill-in-the-blank. They all have a common question string, but how they store their answers is different...

   Class diagrams:

Question		**TrueFalseQuestion**
# `m_question` : string		# `m_question` : string # `m_answer` : bool
+ bool AskQuestion() + void DisplayQuestion()		+ bool AskQuestion()

MultipleChoiceQuestion		**FillInQuestion**
# `m_question` : string # `m_options` : string[4] # `m_correct` : int		# `m_question` : string # `m_answer` : string
+ bool AskQuestion() + void ListAllAnswers()		+ bool AskQuestion()

   How would you store a series of inter-mixed quiz questions in a program? Without polymorphism, you might think to just have separate vectors or arrays for all the questions:

   Example: Not using Polymorphism - Having to create *separate* vectors for each Question type

```
vector<TrueFalseQuestion>       tfQuestions;
vector<MultipleChoiceQuestion>  mcQuestions;
vector<FillInQuestion>          fiQuestions;
```

   Utilizing polymorphism in C++, we could simply store an array of pointers of the parent type:

   Example: Creating a vector of parent class pointers

```
vector<Question*> questions;
```

   And then initialize the question as the type we want during creation:

   Example: Storing different Question types in one vector

```
questions.push_back(new TrueFalseQuestion);
questions.push_back(new MultipleChoiceQuestion);
questions.push_back(new FillInQuestion);
```

Since we are using the `new` keyword here, we would also need to make sure to `delete` these items at the end of the program:

Example: Iterating over all the Question objects in the vector

```
for (auto& question : questions)
{
    delete question;
}
```

Other design considerations

When we're working with polymorphism in this way, we need to be able to treat each child as its parent, from a "calling functions" perspective. Each child can have its own unique member functions and variables, but when we're making calls to functions via a pointer to the parent type, the parent only knows about functions that it, itself, has.

Let's say that the `Question` class has a `DisplayQuestion()` function. Since all its children use `m_question` in the same way and inherit this function, it will be fine to call it via the pointer.

Example: Calling a common function (DisplayQuestion) that is part of the Question family tree

```
ptrQuestion->DisplayQuestion(); // ok
```

But with a function that belongs to a child - not the parent's interface - we wouldn't be able to call that function via the pointer without casting.

Example: Calling a function that doesn't belong to the Question family - just one of the subclasses

```
ptrQuestion->ListAllAnswers();  // not ok

(static_cast<MultipleChoiceQuestion*>(ptrQuestion))
   ->ListAllAnswers(); // ok
```

You could, however, still call that `ListAllAnswers` function from within `MultipleChoiceQuestion`'s `DisplayQuestion` function, and that would still work fine...

Example: Calling the specialized function from within the general function

```
bool MultipleChoiceQuestion::AskQuestion()
{
    DisplayQuestion();
    ListAllAnswers();
    // etc.
}
```

Still fuzzy? That's OK, this is just an overview; we're going to step into how all this works more in-depth next.

6.1.3 Which version of the method is called?

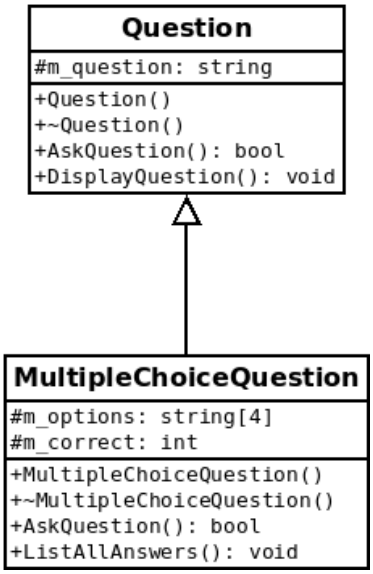
Let's say we have several objects already declared:

Example: Declaring multiple Question family objects

```
Question q1, q2;
MultipleChoiceQuestion mc1;
```

We could create a `Question*` ptr that points to `q1` or `q2` or even `mc1`...

```
Question* ptr;
ptr = &q1;  // Question* pointing to a Question address
ptr = &q2;  // Question* pointing to a Question address
ptr = &mc1; // Question* pointing to a MultipleChoiceQuestion address...?
```



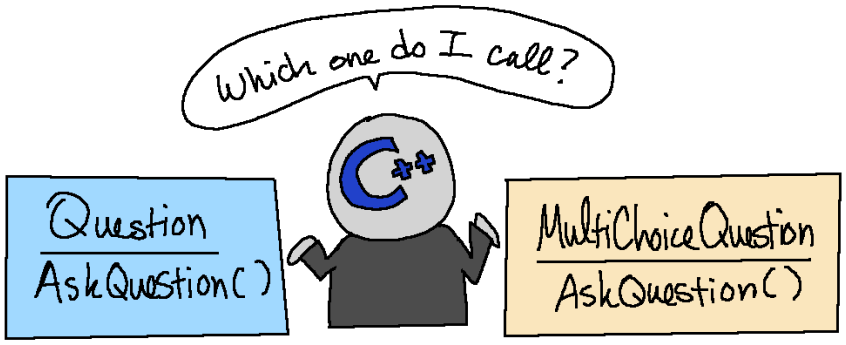
And, any functions that the `Question` class and the `MultipleChoiceQuestion` class could be called from this pointer...

```
// Every Question child just calls the Question::DisplayQuestion() function.
ptr->DisplayQuestion();
```

This is fine for any member methods **not overridden** by the child class. But, which version of the function is called if we used an **overridden** method?

```
// Every Question family class has its own AskQuestion() function.
ptr->AskQuestion();
```

So which is called - `Question::AskQuestion()`, or `MultipleChoiceQuestion::AskQuestion()`?



1. No virtual methods - Which `AskQuestion()` is called?
   Let's say our class declarations look like this:

   Example: Question parent class declaration

   ```
   class Question
   {
       public:
       bool AskQuestion();
       // etc.
   };
   ```

   Example: MultipleChoiceQuestion declaration

   ```
   class MultipleChoiceQuestion : public Question
   {
       public:
       bool AskQuestion();
       // etc.
   };
   ```

Here are the outputs we could have from using pointers in different ways:

A. `Question*` **pointer,** `Question`**'s** `AskQuestion()` **is called:**Example:**

```
Question* ptr = new Question;
bool result = ptr->AskQuestion();
```

B. `MultipleChoiceQuestion*` **pointer,** `MultipleChoiceQuestion`**'s** `AskQuestion()` **is called:**

```
MultipleChoiceQuestion* ptr = new MultipleChoiceQuestion;
bool result = ptr->AskQuestion();
```

C. `Question*` **pointer,** `Question`**'s** `AskQuestion()` **is called:**

```
Question* ptr = new MultipleChoiceQuestion;
bool result = ptr->AskQuestion();
```

"Well, how is that useful at all? The function called matches the pointer data type!" - true, but we're missing one piece that allows us to call **any child's version of the method** from a pointer of the parent type...

2. Virtual methods - Which `AskQuestion()` is called?

   Instead, let's mark our method with the **virtual** keyword:

   Example: Question class with `virtual` function

```
class Question
{
    public:
    virtual bool AskQuestion();
    // etc.
};
```

   Example: MultipleChoiceQuestion class with `virtual` function

```
class MultipleChoiceQuestion : public Question
{
    public:
    virtual bool AskQuestion();
    // etc.
};
```

   Here are the outputs we could have from using pointers in different ways:

   A. `Question*` **pointer,** `Question`**'s** `AskQuestion()` **is called:**

```
Question* ptr = new Question;
bool result = ptr->AskQuestion();
```

B. `MultipleChoiceQuestion*` pointer, `MultipleChoiceQuestion`'s `AskQuestion()` is called:

```
MultipleChoiceQuestion* ptr = new MultipleChoiceQuestion;
bool result = ptr->AskQuestion();
```

C. `Question*` pointer, `MultipleChoiceQuestion`'s `AskQuestion()` is called:

```
Question* ptr = new MultipleChoiceQuestion;
bool result = ptr->AskQuestion();
```

With this, we can now store a list of `Question*` objects, and each question can be a different child class, but we can write one set of code to interact with each one of them.

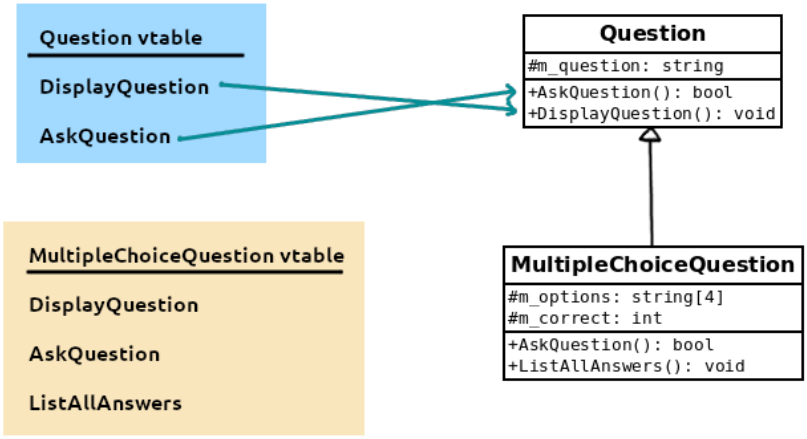
6.1.4 Virtual methods, late binding, and the Virtual Table

By using the **virtual** keyword, something happens with our functions - it allows the pointer-to-the-parent class to figure out *which version* of the method to actually call, instead of just defaulting to the parent class' version. But how does this work?

The `virtual` keyword tells the compiler that the function called will be figured out later. By marking a function as **virtual**, it then is added to something called a **virtual table** - or **vtable**.

The *vtable* stores special *pointers to functions*. If a class contains *at least one virtual function*, then it will have its own vtable.

With the **Question** class, it isn't inheriting any methods from anywhere else so the vtable reflects the same methods it has. But, we also have the child class that inherits **DisplayQuestion()** and overrides **AskQuestion()**.



Because of these **vtables**, we can then have our pointers reference this vtable when figuring out which version of a method to call. Doing this is called **late binding** or **dynamic binding**.

1. When should we use `virtual`?

   Destructors should always be virtual.

   If you're working with inheritance. By making your destructor **virtual** for each class in the family, you are ensuring that the **correct destructor** will be called when the object is destroyed or goes out of scope. If you don't make it virtual and utilize polymorphism, the correct destructor may not be called (i.e., `Question`'s instead of `MultipleChoiceQuestion`'s).

   Constructors cannot be marked `virtual`

   When the object is instantiated (e.g., `ptr = new MultipleChoiceQuestion;`) that class' constructor will be called already.

   Not every function needs to be virtual.

   It's all about design. Though generally, if you always want the parent's version of a method to be called, you wouldn't override that method in the child class anyway.

2. Designing interfaces with pure virtual functions and abstract classes

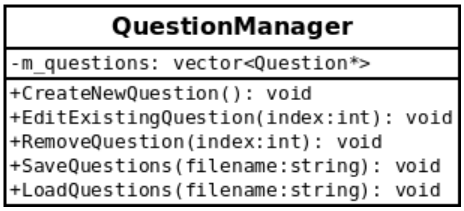
   Polymorphism works best if you're designing a family of classes around some sort of **interface** that they will all share. In the C# language, there is an interface type that is available to you, but that's not here in C++, so we implement it via classes.

What is an Interface?

When we're designing a class to be an interface, the idea is that the user (or other programmers) will just see a set of functions it will interface with - none of the behind-the-scenes, how-it-works stuff.

Most of the devices we use have some sort of **interface**, hiding the more complicated specifics of how it actually works within a case. For example, a calculator has a simple interface of buttons, but if you opened it up you would be able to see its hardware and how everything is hooked up.

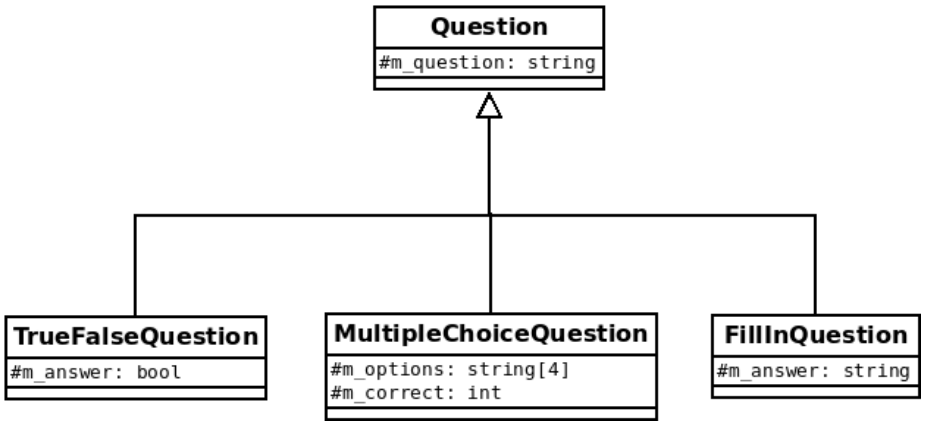
We use the same idea with writing software, where we expose some interface (in the form of the class' **public methods**) as how the "user" interacts with our class.



A common design practice is to write the first **base (parent) class** to be a specification of this sort of **interface** that all its children will adhere to, and to ensure that each child class **must follow the interface** by using something that the compiler will enforce itself: pure virtual functions.

When working with our Quiz program idea, our **base class** is **Question**, which would define the interface for all other types of Questions. Generally, our base interface class **would never be instantiated** - it is not complete in and of itself (i.e., a Question with no types of Answers) - but is merely used to outline a common interface for its family members.

Here is a blank diagram with just the member variables defined, but not yet any functionality, so that we can begin to step through thinking about an interface:



Thinking in terms of implementing a program that could **edit questions** (such as the teacher's view of the quiz), as well as that could **ask questions** (such as the student's view), we can try to think of what kind of functionality we would need from a question...

- Setup the question, answer(s)
- Display the question to the user
- Get the user's answer
- Check if the user's answer was correct

But, the specifics of how each of these question types stores the correct answer (and what data type it is) and validates it differ between each of them...

	User answer	**Stored answer**	**Validate**
True/false	bool	bool answer	**Userinput == answer?**
MultiChoice	int	string options[4]	**Userinput == answer?**
FillIn	int	string answer	**Userinput == answer?**

We could design our Questions so that they have functionality that interacts with the user directly (e.g., a bool function that asks the user to enter their response and returns true if they got it right and false if not) rather than writing functions around returning the actual answer (which would be more difficult because they have different data types).

- Set up question
- Run question

Declarations: We can set up a simple interface for our Questions with these functions. They've been marked as `virtual`, which allows us to use polymorphism, and they've also been marked with `= 0` at the end, marking them as **pure virtual** - this tells the compiler that child classes **must** implement their own version of these methods. A function that contains pure virtual methods is called an **abstract class**.

Example: Question interface with pure virtual functions

```
class Question
{
  public:
  virtual void Setup() = 0;
  virtual bool Run() = 0;

  protected:
  string m_question;
};
```

Now our child classes can inherit from `Question`. They will be required to override `Setup()` and `Run()`, and we can also have additional functions as needed for that implementation:

Example: Inheriting from the Question interface class

```
class MultipleChoiceQuestion : public Question
{
  public:
  virtual void Setup();
  virtual bool Run();
  void ListAllAnswers();

  protected:
  string m_options[4];
  int m_answer;
};
```

Definitions: Each class will have its own implementation of these interface functions, but since they're part of an interface, when we build a program around these classes later we can call all of them the same way.

Example: Question - Defining the Setup function

```
void Question::Setup() {
    cout << "Enter question: ";
    getline( cin, m_question );
}
```

Example: TrueFalseQuestion - Overwriting the Setup function calling the parent class' Setup function

```
void TrueFalseQuestion::Setup() {
    Question::Setup();
    cout << "Enter answer (0 = false, 1 = true): ";
    cin >> m_answer;
}
```

Example: MultipleChoiceQuestion

```
void MultipleChoiceQuestion::Setup() {
  Question::Setup();

  for ( int i = 0; i < 4; i++ )
    {
      cout << "Enter option " << i << ": ";
      getline( cin, m_options[i] );
    }
```

```
  cout << "Which index is correct? ";
  cin >> m_answer;
}
```

Example: FillInQuestion

```
void FillInQuestion::Setup() {
    Question::Setup();
    cout << "Enter answer text: ";
    getline( cin, m_answer );
}
```

Function calls: Now, no matter what *kind* of question subclass we're using, we can utilize the same interface - and the same code.

Example: Using Polymorphism to create the question and calling the common Setup and Run functions

```
// Create the pointer
Question* ptr = nullptr;

// Allocate memory
if ( choice == "true-false" )
{
    ptr = new TrueFalseQuestion();
}
else if ( choice == "multiple-choice" )
{
    ptr = new MultipleChoiceQuestion();
}
else if ( choice == "fill-in" )
{
    ptr = new FillInQuestion();
}

// Set up the question
ptr->Setup();

// Run the question
ptr->Run();

// Free the memory
delete ptr;
```

And, utilizing this interface, we could then store a `vector<Question*>` and set up each question as any question subclass without any duplicate code.

6.1.5 Example usage: Game objects

Let's say we have created a family tree of game objects, starting at the most basic object that has an (x, y) coordinate and dimensions:

GameObject
+ `GameObject()`
+ `Setup( ... )` : void
+ `SetTexture( ... )` : void
+ `GetName()` : string
+ `Update()` : void
+ `Draw()` : void
(etc)
`position` : Vector2f
`name` : string
`sprite` : Sprite
`tx_coord` : IntRect

Then we might have something like an unanimated item in the world, but maybe it has physics so it needs the ability to update, and maybe it has some properties you wouldn't see on a character, like the ability to pick it up, or heal a character.

Item (inherits from GameObject)
+ `Setup( ... )` : void
(etc)
`can_pick_up` : bool
`heal_amount` : int

But then our player and NPC characters also have animated sprites and the ability to move with keyboard input or rudimentary AI:

Character (inherits from GameObject)
+ `Setup( ... )` : void
+ `SetSpeed( float speed )` : void
+ `SetDirection( int dir )` : void
+ `Move( int dir )` : void
+ `Animate()` : void
(etc)
`speed` : float
`direction` : int

If we didn't use polymorphism, we would have to store all objects in their own vector:

Example: Example of storing objects separately

```
vector<Character> m_npcList;
vector<Item> m_pickups;
vector<GameObject> m_decor;
```

But utilizing polymorphism, we can store one vector of `GameObject*` objects initialized on the heap, and any common functionality they have (Update, Draw, etc.) could be accessed via that pointer.

Example: Using Polymorphism for the game entities

```
// Our storage
vector<GameObject*> m_entities;

// Creating a new item (elseware in program)
GameObject* newItem = new Item;

// Adding it to the list
m_entities.push_back( newItem );

// Accessing it later
for ( auto& entity : entities )
{
  entity->Update();
}
```

6.2 Intro: Static

6.2.1 Member variables belong to separate instances

In the context of classes and their variables, the variables we've been working with so far are member variables. When a new object is instantiated, each object has its own version of these variables:

```
Cat cat1;
Cat cat2;

cat1.name = "Kabe";
cat2.name = "Luna";
```

cat1 and cat2 both have variables called name, but cat1's name is different from cat2's name - in memory address, and in value.

We can also declare **static** functions and variables within a class. These static items don't belong to an **instance of an object** - they belong to the class, and all instances share a single one of these static variables.

6.2.2 Static methods (functions)

When a class has a function declared as static, then that function can be called directly via the class itself, though it can also be called via an object. This can be useful for when we need to design a class that we really only need *one of* in the entire program. If we have one class that manages all of the audio in a video game, we shouldn't have to reinstantiate that "manager" over and over and over... we just need it once!

Declaration:

```
class Example
{
  public:
  static void Display();
};
```

Definition:

```
void Example::Display()
{
  cout << "HI" << endl;
}
```

Calls:

```
// Calling via the class
Example::Display();

// Calling via an object
Example e;
e.Display();
```

6.2.3 Static Variables and Functions

Static variables are a special type of variable in a class where **all instances of the class** share the same member. Another term you might hear is a **Class Variable**, whereas a normal member variable of a class would be an **Instance Variable**.

Let's say we are going to declare a **Cat** class, and each cat has its own name, but we also want a counter to keep track of how many Cats there are. The Cat counter could be a static variable and we could write a static method to return that variable's value.

Example: Declaration of a Cat class with a static counter to count the amount of Cat instantiations

```
class Cat
{
public:
  Cat()                        { catCount++; }
  static int GetCount()        { return catCount; }
  void SetName( string name ) { m_name = name; }
  string GetName() const       { return m_name; }

private:
  string m_name;
  static int catCount;
};
```

Within a source file, we will need to initialize this static member. This may go in the class' .cpp file outside of any of the function definitions.

Example: Initializing the Cat's static member variable at the top of Cat.cpp

```
// Initialize static variable
int Cat::catCount = 0;
```

And then any time we create a new Cat object, that variable will automatically add up, and every instance of the Cat will share that variable and its value.

Example: Calling the GetCount function via the objects (catA, catB, catC) or the class (Cat)

```
int main()
{
  Cat catA, catB, catC;

  // These all display 3
  cout << catA.GetCount() << endl;
  cout << catB.GetCount() << endl;
  cout << catC.GetCount() << endl;
  cout << Cat::GetCount() << endl;

  return 0;
}
```

Beyond accessing a static method or member directly through an **instantiated object**, we can also access it through the class name itself, like this: `cout << Cat::GetCount() << endl;`

1. Example usage: Manager class

   In my game engine I use static member variables and functions for my **Manager** classes. These Managers are meant to manage parts of the game, such as the Texture library, Audio library, Inputs, Menus, and so on. Throughout the entire game, I don't create multiple **instances** of the `TextureManager`. Because the functions and data are `static`, I can use this class across the entire project directly.

   Example: Declaring the TextureManager

   ```
   class TextureManager
   {
   public:
     static std::string CLASSNAME;

     static void Add( const std::string& key, const std::string& path );
     static const sf::Texture& AddAndGet( const std::string& key, const std::string
     static void Clear();
     static const sf::Texture& Get( const std::string& key );

   private:
     static std::map<std::string, sf::Texture> m_assets;
   };
   ```

   Example: Defining the member variables (top of TextureManager.cpp):

   ```
   std::string TextureManager::CLASSNAME = "TextureManager";
   std::map<std::string, sf::Texture> TextureManager::m_assets;
   ```

   Example: Function definitions look the same:

   ```
   void TextureManager::Add( const std::string& key, const std::string& path )
   {
     sf::Texture texture;
     if ( !texture.loadFromFile( path ) )
       {
         // Error
         cerr << "Unable to load texture at path \"" << path << "\"", << endl;
         return;
       }

     m_assets[ Helper::ToLower( key ) ] = texture;
   ```

```
}

const sf::Texture& TextureManager::Get( const std::string& key )
{
  if ( m_assets.find( Helper::ToLower( key ) ) == m_assets.end() )
    {
      // Not found
      cerr << "Could not find texture with key " << key << endl;
      throw std::runtime_error( "Could not find texture with key " + key + " - TextureMana
    }

  return m_assets[ Helper::ToLower( key ) ];
}
```

Example: Calling the TextureManager functions:

```
TextureManager::Add( "moose",   "Content/Graphics/Demos/moose.png" );

// ...etc...
sf::Sprite m_player;
m_player.setTexture( chalo::TextureManager::Get( "moose" ) );
```

2. Example usage: Singleton pattern

   Further, we can use the **Singleton pattern** to create a class that can *only* have one instance. You can learn more about the Singleton pattern here: https://en.wikipedia.org/wiki/Singleton_pattern .

   A "Design Pattern" is kind of like a blueprint for a way to implement a structure. These are structures that people have figured out how to build that end up being useful in a lot of scenarios. You can learn more about Design Patterns here: https://en.wikipedia.org/wiki/Design_pattern .

6.3 Lab: Polymorphism and static

6.3.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

6.3.2 Practice programs

1. Practice 1 - Polymorphism
   (a) Reference
   (b) Example output

```
--------------------------------------------------------------------------------
ROUND 1
Gorblin (11/11), AC: 14
Dalfin (9/9), AC: 15

### Gorblin's turn! ###
Gorblin attacks with a Crossbow!
Hit! Dalfin takes 7 points of damage!

### Dalfin's turn! ###
1. Attack with Rapier    2. Attack with Short Bow
CHOICE: 1
Dalfin attacks with a Rapier!
Hit! Gorblin takes 5 points of damage!
```

(c) Instructions

In the CharacterFamily.h there is a base `ICharacter` that contains common functionality for any game character. I've already created the `NonPlayerCharacter` inheriting from `ICharacter`.

In this file, implement a `PlayerCharacter` class. You'll also need the two constructors, a virtual destructor, the virtual DecideAction function, and the two virtual Action functions.

Within CharacterFamily.cpp, for the NPC functions, I used a random roll for the NPC to decide which action to take. I've also implemented the two different attack actions.

You will implement the PlayerCharacter versions:

i. `PlayerCharacter::PlayerCharacter()` You can leave the function body here empty.

ii. `PlayerCharacter::PlayerCharacter( string new_name, int new_armor, int new_hp )` The function body here can be empty, but you should call the `ICharacter` parent class' constructor:

```
PlayerCharacter::PlayerCharacter( string new_name, int new_armor, int new_hp )
    : ICharacter( new_name, new_armor, new_hp )
{
}
```

iii. `PlayerCharacter::~PlayerCharacter()` You can leave the function body here empty.

iv. `void PlayerCharacter::DecideAction( int& attack_roll, int& damage_roll )` In this function display a simple numbered menu, such as:

```
1. Attack with Rapier       2. Attack with Short Bow
```

Ask the user to enter a choice and store it in an integer variable. Based on the player's selection call Action1 or Action2.

v. `void PlayerCharacter::Action1( int& attack_roll, int& damage_roll )` Display a message that the player attacks with the weapon (e.g., Rapier).

You'll also set values for the `attack_roll` and `damage_roll`. You can use the `RollDie` function. Example:

- Attack roll: `RollDie( 20 ) + 3`
- Damage roll: `RollDie( 8 ) + 3`

vi. `void PlayerCharacter::Action2( int& attack_roll, int& damage_roll )` Display a message that the player attack with the weapon (e.g., Short Bow). Same thing as above.

- Attack roll: `RollDie( 20 ) + 4`
- Damage roll: `RollDie( 6 ) + 3`

vii. main() Within the main function I've already created two NonPlayerCharacters:

```
players.push_back( new NonPlayerCharacter( "Gorblin", 14, 11 ) );
players.push_back( new NonPlayerCharacter( "Zeepboop", 13, 14 ) );
```

Go ahead and create a `PlayerCharacter` and push it into the `players` vector.

I have the two NPCs targeting different indices:

```
players[0]->SetOpponentIndex( 1 );
players[1]->SetOpponentIndex( players.size()-1 );
```

Your player will be at `players[2]`, use the same function to set the Player's opponent to either 0 or 1.
After that the rest of the game should work since we're utilizing polymorphism to treat all characters like an `ICharacter*` pointer.

---

2. Practice 2 - Static

   (a) Example output

   ```
   Total students: 0
   Total students: 1
   Total students: 2
   Total students: 3
   Total students: 4
   Total students: 4
   ```

   (b) Instructions

      i. Student.h file:
         - Add a class-level (static) variable to count how many students get instantiated.
         - Declare a static `int` variable named `total_students`.

      ii. Student.cpp file:
         - At the top of the file outside of all of the functions initialize the `Student::total_students` static variable. (See the reference section).
         - In the `Student` constructor add a line of code that increments the `total_students` by 1. This counts up each time a new student object is created.
         - Within the `GetTotalStudents` function return the `total_students` static variable.

6.3.3 Practice 2 - Static manager

1. Instructions

   Within Product.h the `Product` class is declared and `ProductManager` is also specified as a friend class. The Product class has three private member variables: `name`, `price`, and `year`. These normally wouldn't be accessible to functions or classes outside of itself but since `ProductManager` has been specified as a friend the manager class will be able to access these.

   Within ProductManager.h declare the `ProductManager` class with the following:

   - `AddProduct`, a static void function that takes in a `Product` as its parameter.

- `AddProduct`, a static void function that takes in a `name`, `price`, and `year` as its parameters.
- `Display`, a static void function.
- `products`, a static vector of `Product` objects.

Within ProductManager.cpp you will need to initialize the static member vector:

```
vector<Product> ProductManager::products;
```

And define each of its member functions:

(a) `void ProductManager::AddProduct( Product new_product )`
- Push the `new_product` into the `products` vector.

(b) `void ProductManager::AddProduct( string name, float price, int year )`
- Create a new `Product` object, initialize it with the `name`, `price`, and `year`, and push this object into the `products` vector.

2. `void ProductManager::Display()`

- Use this code as the header:

```
cout << left << fixed << setprecision(2);
cout << setw( 5 ) << "ID"
     << setw( 20 ) << "NAME"
     << setw( 10 ) << "PRICE"
     << setw( 10 ) << "YEAR"
     << endl;
cout << string( 80, '-' ) << endl;
```

then iterate through all of the `products` and display each item's index (`i`), name, price, and year.

3. Example output

```
LAUNCH PRICES

ID   NAME                PRICE     YEAR
--------------------------------------------------------------------------------
0    NES                 199.00    1985
1    SNES                199.00    1991
2    Nintendo 64         199.00    1996
3    GameCube            199.00    2001
4    Wii                 249.00    2006
5    Wii U               299.00    2012
6    Nintendo Switch     299.00    2017
```

6.3.4 Graded programs

1. Graded program

   (a) Example output

   You can run the program with either the `run` argument or `test` argument. The test argument runs automated tests.

```
./quiz.exe run
--------------------------------------------------------------------------------
True or false     - Pineapple belongs on pizza.
1. TRUE          2. FALSE
Your answer: 1
Correct!

--------------------------------------------------------------------------------
True or false     - Salmon is a mammal.
1. TRUE          2. FALSE
Your answer: 1
Wrong!

--------------------------------------------------------------------------------
Fill in the blank - Olathe is in which state?
Your answer: Missouri
Wrong!

--------------------------------------------------------------------------------
Fill in the blank - 2+2 = ?
Your answer: 4
Wrong!

--------------------------------------------------------------------------------
Multiple choice   - What is the Hindi word for cat?
1. aap
2. billee
3. kutta
4. kela
Your answer: 2
Correct!

RESULTS:
2 out of 5 correct
```

   (b) Instructions

   - Within **QuestionManager** files we will implement a static manager class that stores and handles the Question types.
   - Within **QuestionFamily** files there is an `IQuestion` abstract interface class and child classes. I've implemented `IQuestion`

and `TrueFalseQuestion` and you will implement an additional 2 types. We will utilize polymorphism to store all of the questions as a vector of `IQuestion*` pointers in the QuestionManager.

i. QuestionFamily.h

Within QuestionFamily.h I've already implemented `IQuestion`. Every type of quiz question has question text (`string question`), but different types of questions store their answers differently. I've already implemented `TrueFalseQuestion` - it stores `bool answer` to store the correct answer here.

In the same file, create two more classes that inherit from `IQuestion`:

- `FillInQuestion`
  - Protected member variable `answer` is a string.
  - Your `IsCorrect` function should take in a `string guess` parameter.
  - Also implement a parameterized constructor (takes in a string for the question and a string for the answer).
  - The default constructor and destructors' function bodies can be left empty.
- `MultipleChoiceQuestion`
  - Protected member variable `vector<string> options` for the multiple choices
  - Protected member variable `int correct` to store the index of the correct answer.
  - Your `IsCorrect` function should take in a `int guess` parameter.
  - Also implement a parameterized constructor (takes in a string for the question, vector of strings for the options, and an integer for the correct answer).
  - The default constructor and destructors' function bodies can be left empty.

ii. QuestionFamily.cpp

Within QuestionFamily.cpp implement the function definitions. You can use the `TrueFalseQuestion` for reference.

iii. QuestionManager.h

The QuestionManager has the following static functions:

- `static void AddQuestion( IQuestion* newQuestion );`
- `static void RunQuiz();`

And the following static variable:

- `static vector<IQuestion*> questions;`

You don't have to update anything in this file, just look at it to see that it's declared static functions/variables.

iv. QuestionManager.cpp

At the top of the file make sure to include this "definition" of the static variable:

```
vector<IQuestion*> QuestionManager::questions;
```

Within the **AddQuestion** function, push the `newQuestion` parameter into the `questions` vector.

Within the **RunQuiz**, create a variable to keep track of the amount of questions answered correctly.

Use a for loop to iterate over all of the question pointers in the `questions` vector. Within the for loop, call the question's `AskQuestion()` function. This function returns `true` if the user answered it correctly and `false` otherwise. If the user answered the question correctly, add 1 to the total correct counter.

After the for loop, display the amount of questions answered correctly and the total amount of questions.

v. main.cpp

Within main, create at least one of each type of question (not IQuestion), for example:

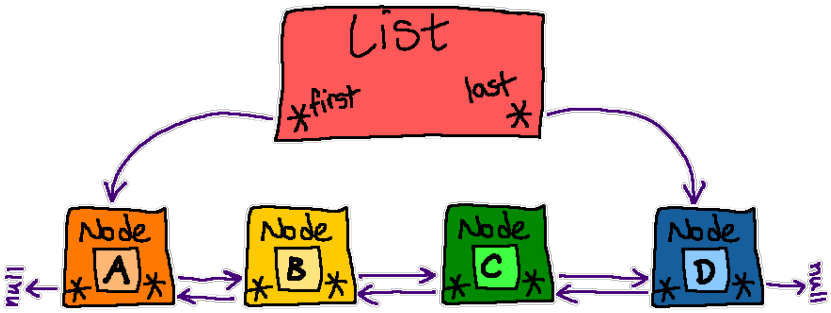
```
QuestionManager::AddQuestion(
    new TrueFalseQuestion( "Pineapple belongs on pizza.",
        true ) );
```

Afterwards, call the manager's `RunQuiz()` function.

```
QuestionManager::RunQuiz();
```

7 Week 7: Linked List structure

7.1 Intro: Linked List structure



7.1.1 Coming from arrays...

Before taking a Data Structures course, you've probably only managed data in your program via **arrays**.

When we declare an array in C++, we must give it a size (whether we're making a dynamic array or a vanilla array). This ensures that memory can be allocated so that each element of the array is side-by-side with other elements.

1. Investigating arrays and memory addresses:

   Let's write a simple program to look at how arrays are stored in memory, in case you don't remember.

   First, how big is one element? We can use the `sizeof` function to find out how big, in bytes, a given data type is:

   ```
   cout << "The size of a int is: "
        << sizeof( int ) << " bytes." << endl;
   ```

   ```
   The size of a int is: 4 bytes.
   ```

   One integer is 4 bytes. Next, we can declare an array of integers and look at how many bytes that takes up:

   ```
   const int SIZE = 10;
   int intArr[SIZE];
   cout << "Size of a int array with "
        << SIZE << " elements: "
        << sizeof( intArr ) << " bytes." << endl;
   ```

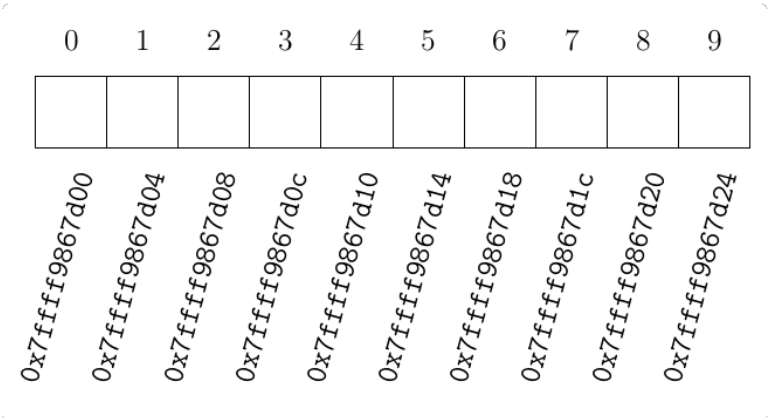
   ```
   Size of a int array with 10 elements: 40 bytes.
   ```

Now, let's look at the memory addresses of each element:

```
cout << "Memory addresses for the int array:" << endl;
for ( int i = 0; i < SIZE; i++ )
{
    cout << "Element " << i << ": " << &(intArr[i]) << endl;
}
```

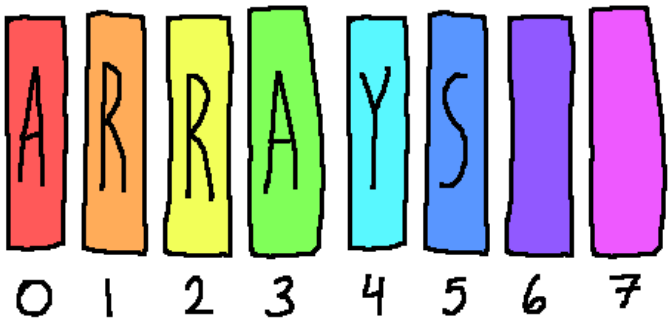
```
Memory addresses for the int array:
Element 0: 0x7ffff9867d00
Element 1: 0x7ffff9867d04
Element 2: 0x7ffff9867d08
Element 3: 0x7ffff9867d0c
Element 4: 0x7ffff9867d10
Element 5: 0x7ffff9867d14
Element 6: 0x7ffff9867d18
Element 7: 0x7ffff9867d1c
Element 8: 0x7ffff9867d20
Element 9: 0x7ffff9867d24
```

Each time this program runs, we will have different memory addresses for the **intArr**, but those addresses **will always be contiguous in memory, 4 bytes apart.**



Sidenote 1: Hexadecimal When we write in hexadecimal, we want each number to only take up one character space. So, we have 0-9 to be 0 through 9, but then 10 is A, 11 is B, 12 is C, 13 is D, 14 is E, and 15 is F.

Sidenote 2: Address of the beginning of the array The address of `intArr` is the same as the address of `intArr[0]`.



2. Pros and Cons of arrays for storing data:

Pros:

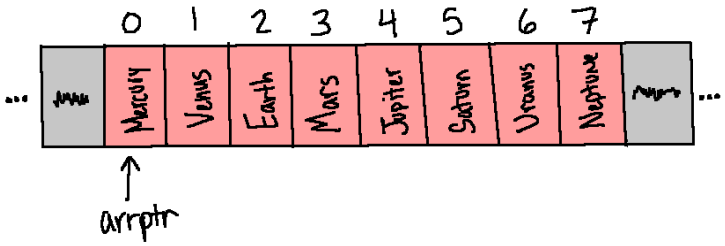
- **Random access is instant:** This means, we can access an element at *any arbitrary index* instantaneously. How? Well, we have the address of the 0th element, and to get an item at position i, we just have to add `i * sizeof(int)`. A simple math operation is virtually instant. `AddressOf( i ) = AddressOf( 0 ) + i * sizeof( dataType )`

Cons:

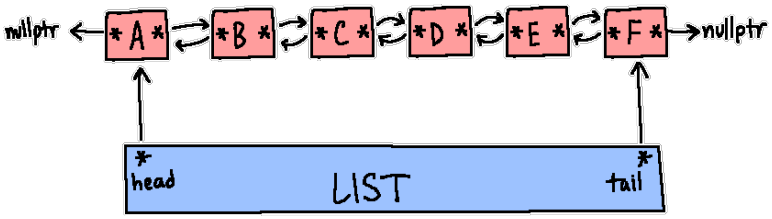
- **Resizing a dynamic array is costly:** Any time our array is full, to resize it we have to allocate memory for a *new array* of a bigger size. Then, we have to copy each element from the old array to the new array. This takes time any time resize occurs.
- **We have to over-estimate the array size:** We don't want to have to resize the entire array every time a new item is inserted, so we generally will resize it by some quantity each time (not just adding 1 to the size of the array). This means we're allocating more space than we're *actually using* if the array isn't full.

The study of Data Structures is all about the trade-offs between different data types: Some are faster to access data but slower to insert new data and vice versa. Instead of storing data in an **array-based structure**, we can make a tradeoff of **less memory used, instant insertion of new data** in exchange for **slower access** using a **linked structure**.

A dynamic array:

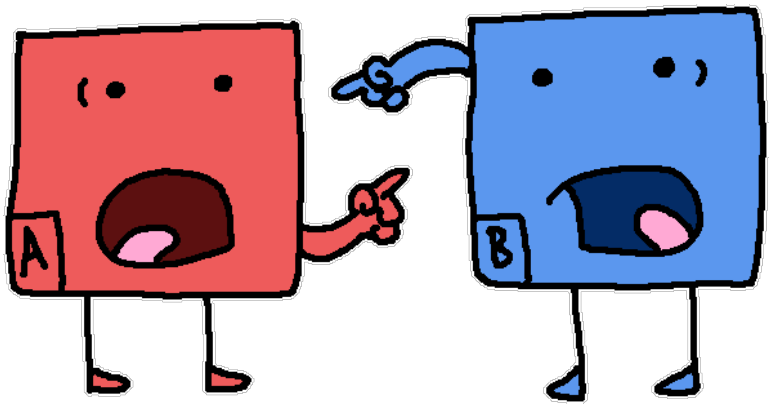


A linked list:



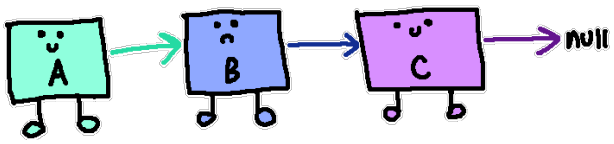
7.1.2 Building a Linked List

1. Anatomy of a Node class



The **Node** is one of the two structures we build when implementing a linked list or other linked type structure. The node is responsible for storing the data itself, and storing one or more pointers.

Singly-linked List's Nodes

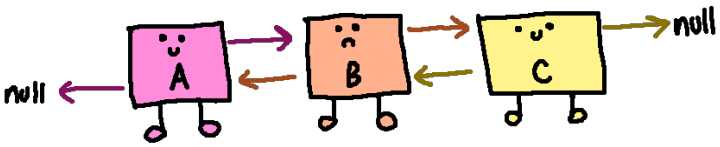


With a **singly-linked list**, the nodes only point to the **next Node** after them. These nodes don't know what comes before each of themselves.

SinglyLinkedNode	
- data	: TYPE
- ptrNext	Node<TYPE>*

```
template <typename T>
struct Node {
public:
  Node();
  Node<T>* ptrNext;
  T data;
};
```

Doubly-linked List's Nodes



With a **doubly-linked list**, the nodes point to both the **next Node** and the **previous Node**.

DoublyLinkedNode	
- data	: TYPE
- ptrPrev	Node<TYPE>*
- ptrNext	Node<TYPE>*

```
template <typename T>
struct Node {
public:
  Node();
  Node<T>* ptrNext;
  Node<T>* ptrPrev;
  T data;
};
```

Warning! Remember that any class with pointers in it should be setting those pointers to `nullptr` in its constructor!

A Node is simple enough that you could implement it as a **struct** instead of a **class**, but it's up to you. A Node is only used by the List (or linked

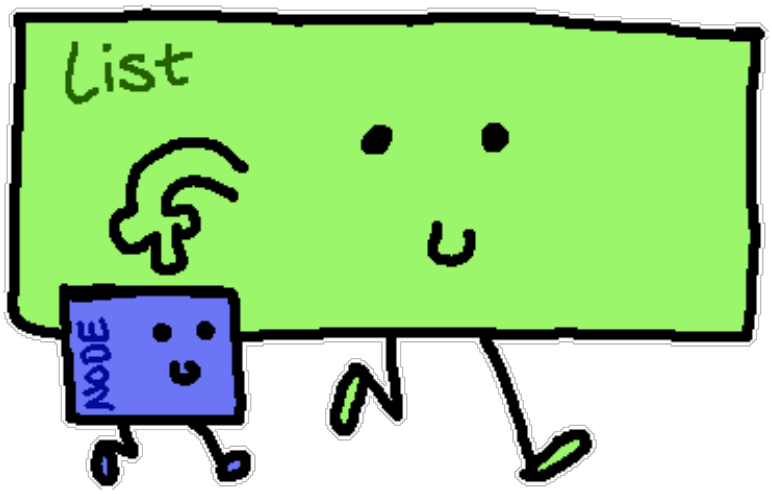
structure) itself, the user wouldn't be accessing the Node or know about it.

Single vs. Double?

I generally prefer to teach how to implement a doubly-linked list first, because the functionality is easier to implement when each Node is aware of what comes before itself.

Once you understand how a doubly-linked list works, you can figure out a singly-linked list; it just means having to iterate through the list more often.

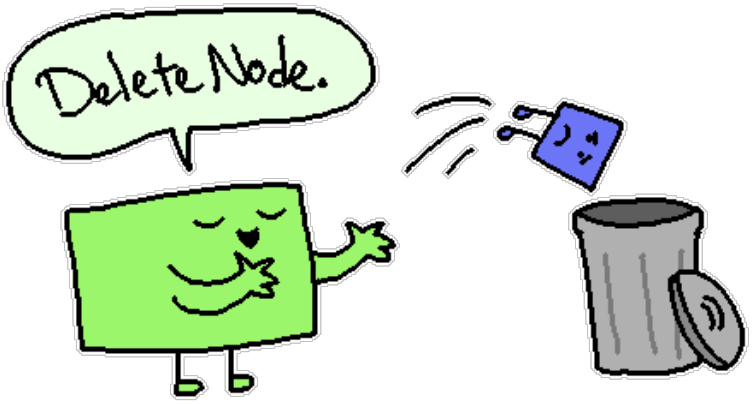
2. Anatomy of a List class



The List class itself is what handles interfacing with the "user" (or other programmers using your List). Data Structures generally will have **Access**, **Add**, **Remove**, and **Search** functionality, and those functions (and helper functions) are stored in our List.

The List will also keep track of the **first Node** in the list, and often also the **last Node** by storing `Node*` pointers. (You could get by without the last Node pointer, but it's just easier, dude.)

The list is also responsible for the **memory management** - it will allocate memory for a new `Node` when it's needed, and free the memory of a `Node` when it's to be removed.



Generally, a List will have functionality like:

DoublyLinkedList	
- ptrFirst	: Node<TYPE>*
- ptrLast	: Node<TYPE>*
- itemCount	: int
+ DoublyLinkedList()	
+ ~DoublyLinkedList()	
+ PushFront(newData: T)	: void
+ PopFront()	: void
+ GetFront()	: TYPE&
+ PushBack(newData: T)	: void
+ PopBack()	: void
+ GetBack()	: TYPE&
+ PushAt(index: int, newData: T)	: void
+ PopAt(index: int)	: void
+ GetAt(index: int)	: TYPE&

The **Push** functions are to add new items. There are three varieties: Add to the *start* of the list (PushFront), Add to the `end` of the list (PushBack), and Add to **somewhere in the middle** of the list (PushAtIndex).

The **Pop** functions are to remove items.

And the **Get** functions are to access data stored in Nodes.

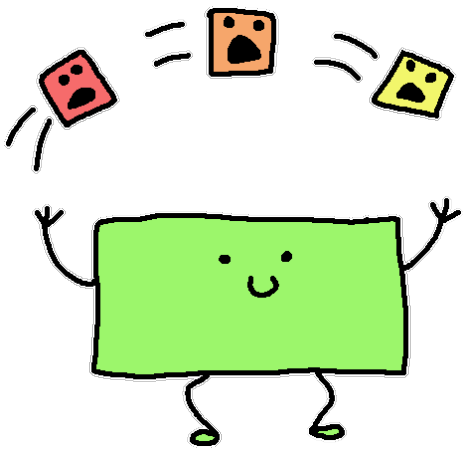
Depending on the design of the List, sometimes the "AtIndex" functions won't be implemented.

Example Linked List declaration:

```
template <typename T>
class LinkedList
{
```

```
private:
  Node<T>* ptrFirst;
  Node<T>* ptrLast;
  int itemCount;

public:
  LinkedList();
  ~LinkedList();

  void PushFront( T newData );
  void PushBack( T newData );
  void PushAtIndex( int index, T newData );

  void PopFront() noexcept;
  void PopBack() noexcept;
  void PopAtIndex( int index );

  T& GetFront();
  T& GetBack();
  T& GetAtIndex( int index );

  void Clear();
  bool IsEmpty();
  int Size();
};
```

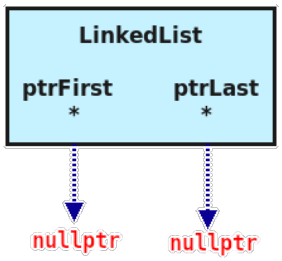
3. Functionality of a Doubly-Linked List



We're going to look at the way functions work, assuming we're using doubly-linked Nodes. This is because it's more straightforward, and once you understand how doubly-linked lists work, you can figure out how a singly-linked list works (it just means more traversing).

(a) Constructor and destructor

i. **Constructor**



When a new LinkedList is created, it is empty. This means setup includes:

- Set your pointer to the first node (`ptrFirst`) and the last node (`ptrLast`) each to `nullptr`!
- Set the amount of items stored in the list (`m_itemCount`) to 0.

ii. **Destructor**

The destructor should make sure that all memory is freed when the LinkedList is destroyed. I've moved this functionality into the `Clear()` function, so just make sure to call `Clear()` within the destructor.

(b) Helper functions

i. **Clear**

```
void Clear();
```

The Clear function will be responsible to freeing the memory of all the nodes. You can implement this function by writing a while loop that continues looping while there are still items in the list, calling a Pop function until it is finally empty.

```
while ( !IsEmpty() )
{
    PopFront();
}
```

ii. **IsEmpty**

```
bool IsEmpty();
```

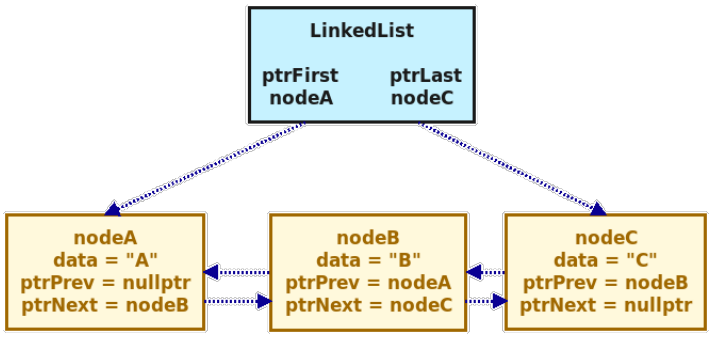
This should simply return `true` if `itemCount` is 0. Otherwise, return `false`.

iii. **Size**

```
int Size();
```

Return the value of `itemCount`.

(c) Walking the list



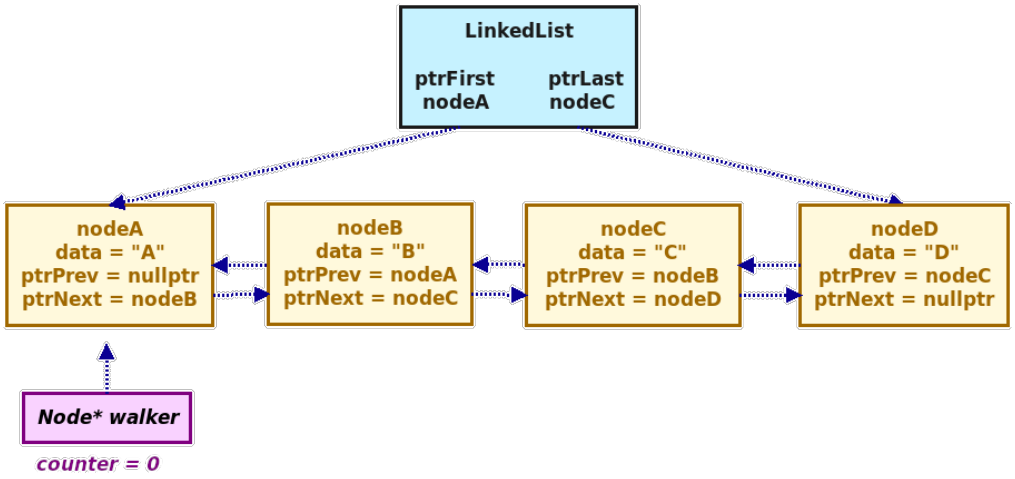
The STL List only allows you to access items at the *front* and *back* of the list, but with our list we are going to also be able to work with the *middle* - but to do so, we need to add some logic to **walk the list**. All of the "At" functions below will use this functionality to get to a certain **index**.

First, we need to create a Node pointer. We aren't going to allocate space for this pointer, we will just use it to point at each item we're currently investigating.

When we want to get *to* somewhere, we will have an **index** position. This will be our goal location.

We will need to create a **counter** variable as well to count how many steps we've taken to that position.

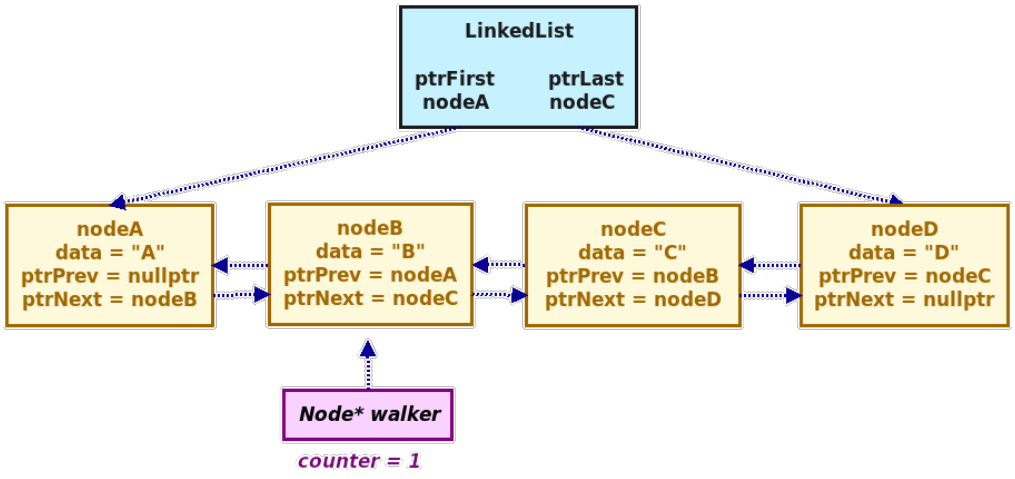
```
// Point to list's first item
Node<TYPE>* walker = this->ptrFirst;
int counter = 0;
```



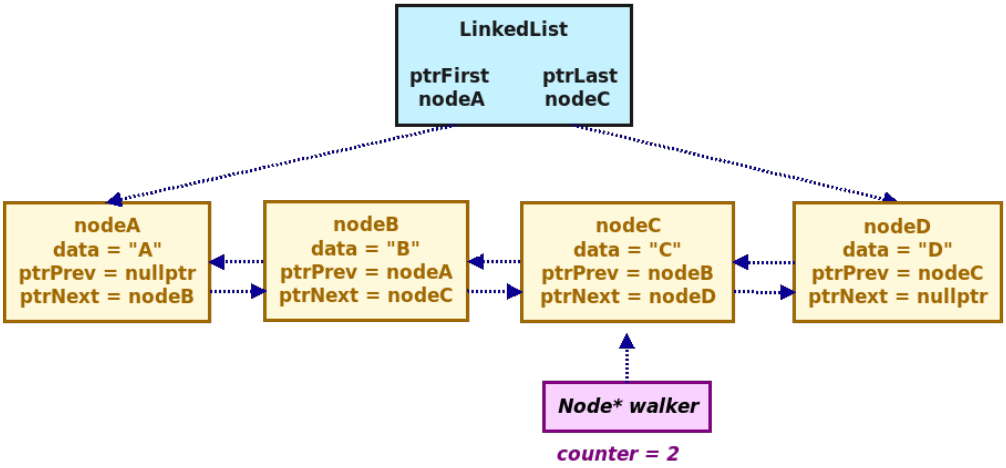
We begin by pointing our *walker* Node pointer to wherever the list's `ptrFirst` is pointing to, and we set our **counter** to 0.

while counter is not the index we want... walk forward. To walk forward, we set our *walker* node to point to the current location's `ptrNext`.

```
while ( counter != index )
{
  walker = walker->ptrNext;
}
```



We keep moving forward...



. . . And once our loop ends, we have arrived at the index desired.

Keep this functionality in mind as we continue, it will be needed for the "PushAt", "PopAt", and "GetAt" functions.

(d) Accessing data

i. GetFront - Retrieving the data at the beginning of the list
   Return the data stored in the `ptrFirst` node.

ii. GetBack - Retrieving the data at the end of the list
   Return the data stored in the `ptrLast` node.

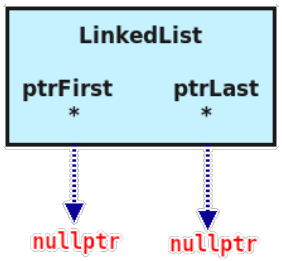
iii. GetAt - Retrieving the data in the middle of the list
   Use a **walker** to get to a certain index, and return the data at the `walker` node.

(e) Adding data

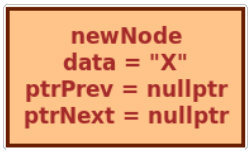
The **Push** functions are to add data to our list. We will be passing in some data and it will be added at the *front*, *back*, or maybe in the *middle*.

We're going to have slightly different instructions to follow based on whether our list is starting out **empty**, or if we already have some items stored in it, so make sure to take notice of the **multiple scenarios**.

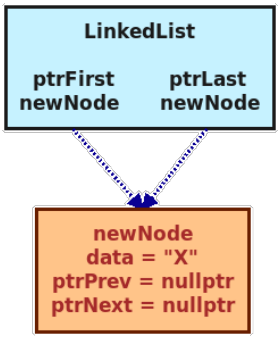
i. Adding data to an EMPTY LIST
   An empty list is one that has its `ptrFirst` and `ptrLast` both pointing to `nullptr`.



First, we allocate space for a **new Node** and set its data:



Then, we update the LinkedList's pointers - the new node is now our new **first** and **last** item:

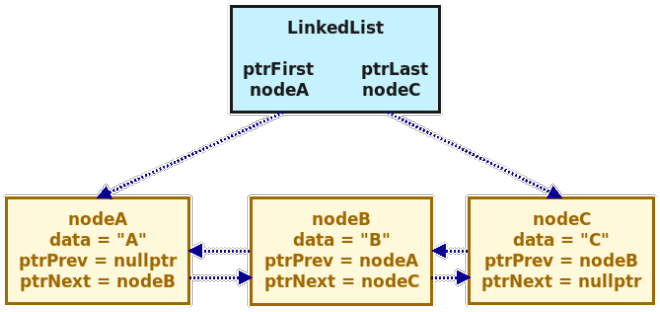


ii. PushFront - Add a new item to the beginning of the list

Scenario 1: If the list is currently empty, then follow the "Adding data to an EMPTY LIST" steps above.

Scenario 2: If the list contains at least one item, then follow this.

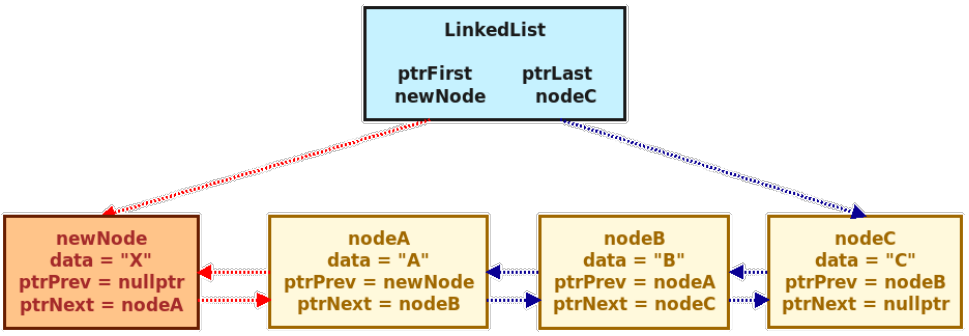
First, the list begins in a state with nodes already stored:



Next, we create a new node and set its data:

```
Node<TYPE>* newNode = new Node<TYPE>;
newNode->data = data_from_parameter;
```

Then we update the pointers:



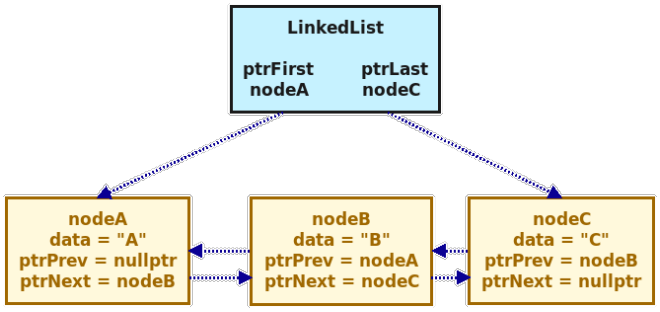
- The `newNode`'s `ptrNext` is the list's current `ptrFirst`.
- The list's current `ptrFirst`'s new `ptrPrev` is the `newNode`.
- The list's new `ptrFirst` is the `newNode`.

iii. PushBack - Add a new item to the end of the list

Scenario 1: If the list is currently empty, then follow the "Adding data to an EMPTY LIST" steps above.

Scenario 2: If the list contains at least one item, then follow this.

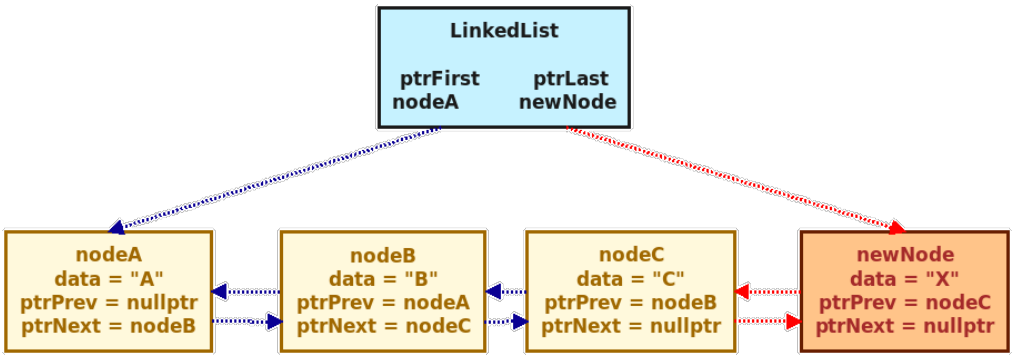
First, the list begins in a state with nodes already stored:



Next, we create a new node and set its data:

```
Node<TYPE>* newNode = new Node<TYPE>;
newNode->data = data_from_parameter;
```

Then we update the pointers:



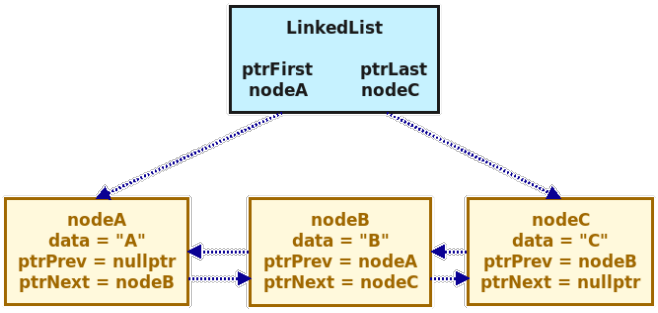
- The `newNode`'s `ptrPrev` is the list's current `ptrLast`.
- The list's current `ptrLast`'s new `ptrNext` is the `newNode`.
- The list's new `ptrLast` is the `newNode`.

iv. PushAt - Add a new item to the *middle* of the list

Scenario 1: If the list is currently empty, then follow the "Adding data to an EMPTY LIST" steps above.

Scenario 2: If the list contains at least one item, then follow this.

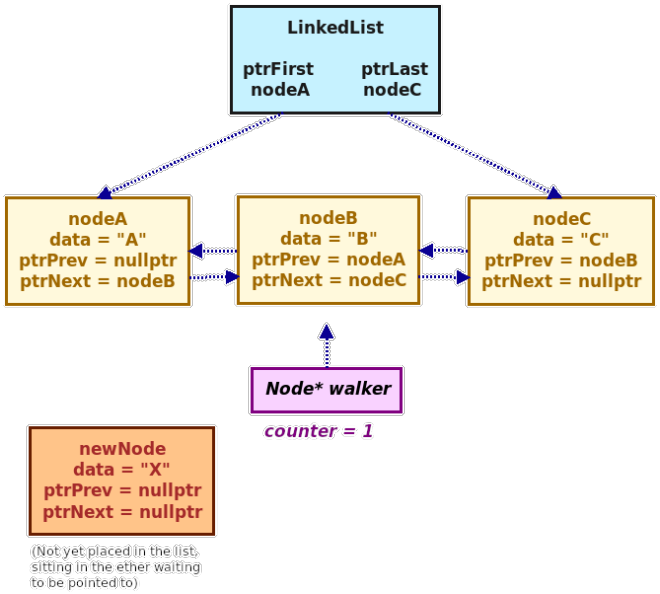
First, the list begins in a state with nodes already stored:



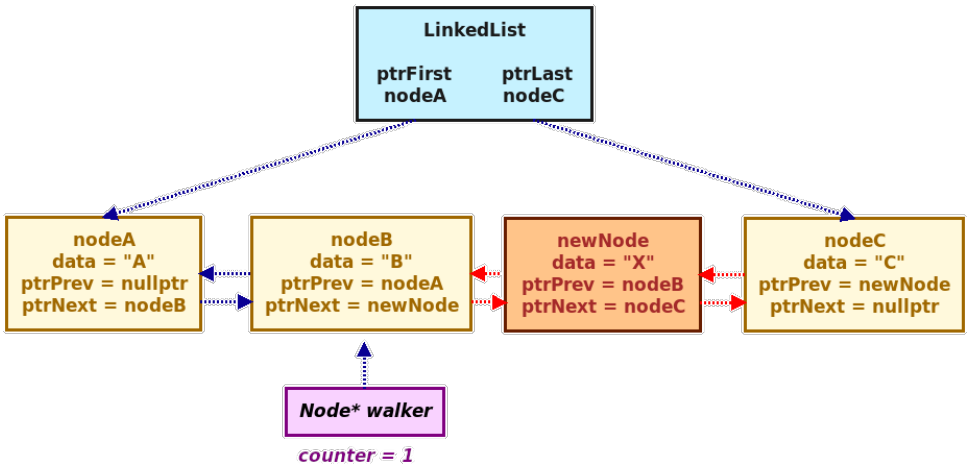
Next, we create a new node and set its data:

```
Node<TYPE>* newNode = new Node<TYPE>;
newNode->data = data_from_parameter;
```

Now we need to **walk the list** to find the proper location:



Then we update the pointers:

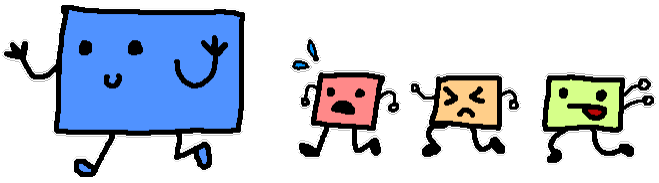


The walker node will be pointing at the item that will go *before* the `newNode`, so we can use it to adjust pointers. I also like to make some temporary pointers to "previous" and "next" to keep the code clean.

```
Node<TYPE>* previous = walker;
Node<TYPE>* next = walker->ptrNext;

newNode->ptrPrev = previous;
newNode->ptrNext = next;

next->ptrPrev = newNode;
previous->ptrNext = newNode;
```

If you didn't create these "aliases" for the previous/next nodes, it would look like this instead:

```
newNode->ptrPrev = walker;
newNode->ptrNext = walker->ptrNext;

walker->ptrNext->ptrPrev = newNode;
walker->ptrNext = newNode;
```



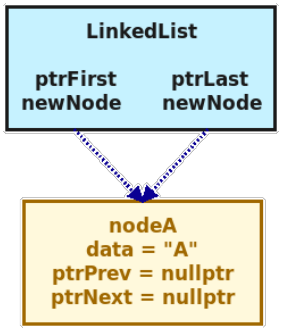
(f) Removing data

We will also be able to remove data from the *front*, *back*, or in the *middle* of our list. We will again have two scenarios: If we're removing the last item in the list, or if there is more than one item in the list.

We will need an error check as well: If the list is empty, we can't remove any data!

i. Removing the LAST ITEM in the list

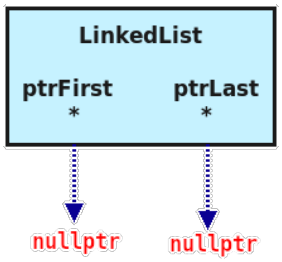
A list with 1 item remaining in it means that its `ptrFirst` and `ptrLast` are both pointing at the same node.



In this case, we delete the node via one of these pointers:

```
\begin{lstlisting}[style=cpp]
delete this->ptrFirst;
\end{lstlisting}
```

And then we reset the list's `ptrFirst` and `ptrLast` to point to `nullptr`.

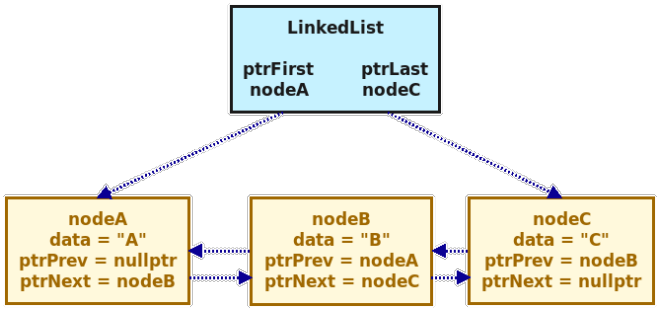


ii. PopFront - Remove the item at the front of the list

Scenario 1: If there's only 1 item in the list, then follow the "Removing the LAST ITEM in the list" steps above.

Scenario 2: If the list has more than 1 item in it, then follow this.

First, the list begins in a state with nodes already stored:

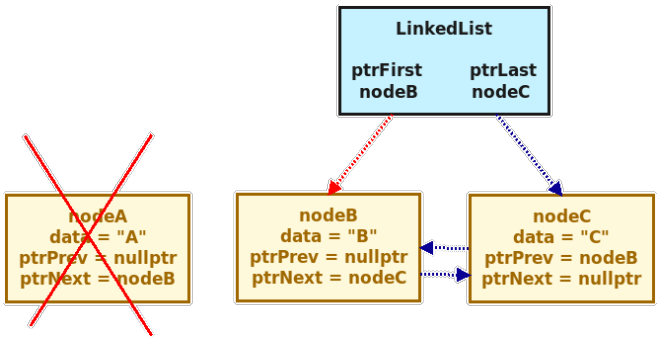


We want to assign a new `ptrFirst` - whatever node is after the first one becomes the new first:

```
this->ptrFirst = this->ptrFirst->ptrNext;
```

Then we can delete the "old first" item, and set the "new first"'s `ptrPrev` to nullptr:

```
delete this->ptrFirst->ptrPrev;
this->ptrFirst->ptrPrev = nullptr;
```

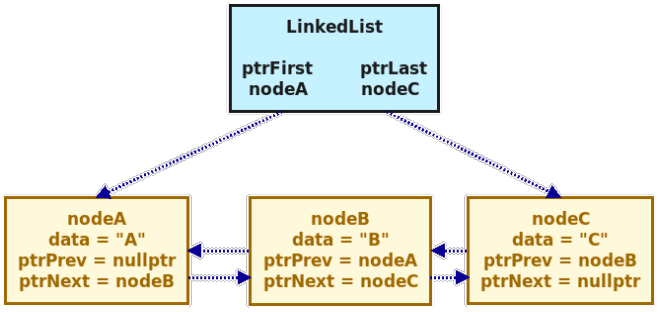


iii. PopBack - Remove the item at the end of the list

Scenario 1: If there's only 1 item in the list, then follow the "Removing the LAST ITEM in the list" steps above.

Scenario 2: If the list has more than 1 item in it, then follow this.

First, the list begins in a state with nodes already stored:

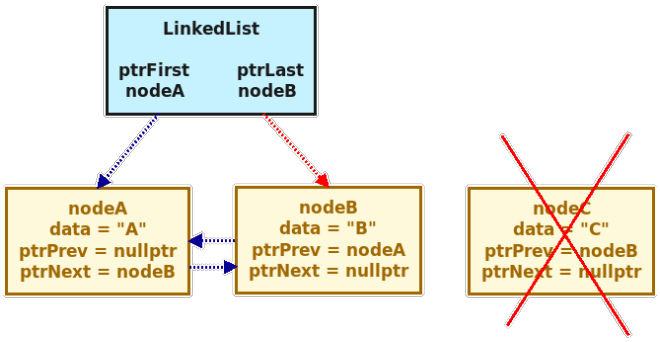


We want to assign a new `ptrLast` - whatever node is before the last one becomes the new last:

```
this->ptrLast = this->ptrLast->ptrPrev;
```

Then we can delete the "old last" item, and set the "new last"'s `ptrNext` to nullptr:

```
delete this->ptrLast->ptrNext;
this->ptrLast->ptrNext = nullptr;
```

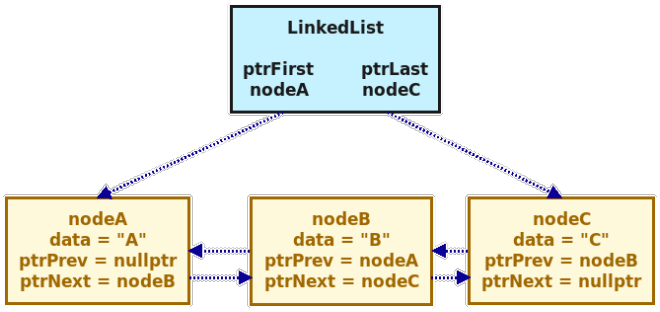


iv. PopAt - Remove an item from the middle of the list

Scenario 1: If there's only 1 item in the list, then follow the "Removing the LAST ITEM in the list" steps above.

Scenario 2: If the list has more than 1 item in it, then follow this.

First, the list begins in a state with nodes already stored:



We need to use a **walker** to get to the index that we want to remove. Then, we can update the item *before* and *after* the `walker` to point to each other.

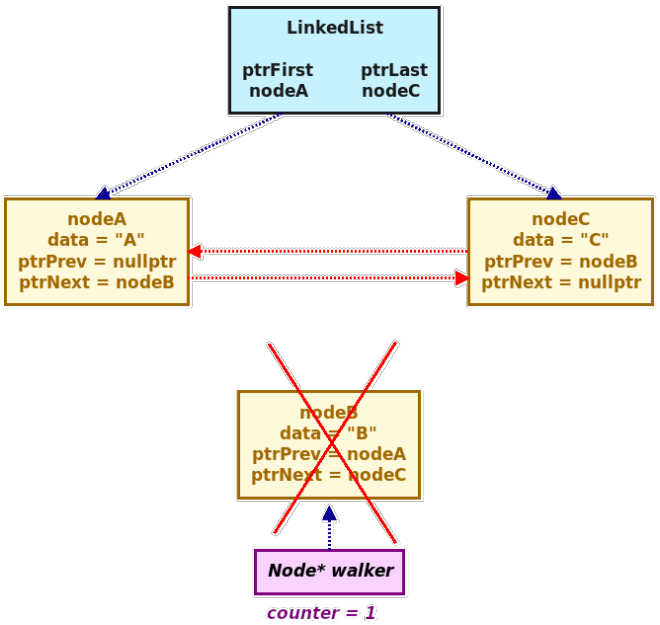
With aliases:

```
Node<T>* before = walker->ptrPrev;
Node<T>* after = walker->ptrNext;
before->ptrNext = after;
after->ptrPrev = before;
```

Without aliases:

```
walker->ptrPrev->ptrNext = walker->ptrNext;
walker->ptrNext->ptrPrev = walker->ptrPrev;
```

We then delete the Node at the `walker` location and we're done.



7.2 Lab: Linked List structure

7.2.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - Assignment direct link
  - R.W.'s courses homepage

7.2.2 Assignment information

Getting started:

1. This lab is located in your **repository folder** under `ArrayStructures`.
2. In VS Code use "Open Folder" to open the specific subprogram you wish to work with.
3. BUILD: Since this is a big project, use the `make` command (in the same directory as the Makefile) to build the program.
4. RUN: `./release.exe` for the release version, or `gdb ./debug.exe` for the debug version.

~

Structures: We will work on the **LinkedList** structure. We will cover the Stack and Queue structures at a later date.

~

Turning in your work:

To turn in the assignment, make sure you've **committed** and **synced (pushed)** your changes to your GitLab repository. Double check the file from the GitLab webpage to ensure it's up-to-date. Then, submit the URL to the folder for the week.

~

Working on this assignment:

After building the program you can run it with the following arguments:

- `./release.exe linkedlist` - Run tests for linked list
- Array Structures reference:
  - Textbook, archived class lecture
  - Lecture sildes

7.2.3 Included files:

```
LinkedStructures
 DataStructures
    LinkedList
       LinkedList.h
       LinkedListNode.h
       LinkedListTester.cpp
       LinkedListTester.h
    LinkedQueue
       LinkedQueue.h
       LinkedQueueTester.cpp
       LinkedQueueTester.h
    LinkedStack
        LinkedStack.h
        LinkedStackTester.cpp
        LinkedStackTester.h
 Exceptions
    InvalidIndexException.h
    ItemNotFoundException.h
    NotImplementedException.h
    NullptrException.h
    StructureEmptyException.h
    StructureFullException.h
 instructions.org
 main.cpp
 Makefile
 Utilities
```

```
StringHelper.cpp
StringHelper.h
Style.cpp
Style.h
```

7.2.4 Instructions

1. bool LinkedList<T>::IsEmpty() const

   The list is considered empty if the `m_itemCount` is 0.

2. int LinkedList<T>::Size() const

   Return the `m_itemCount`.

3. T& LinkedList<T>::GetFront()

   - Error check: If the LinkedList is empty, then throw an `Exception::StructureEmptyExcept` `"FUNCTION NAME", "MESSAGE" )`.
   - Return the data stored at the front of the list; i.e., use the

   `m_ptrFirst` to access the Node, then access the Node's `m_data`.

4. T& LinkedList<T>::GetBack()

   - Error check: If the LinkedList is empty, then throw an `Exception::StructureEmptyExcept` `"FUNCTION NAME", "MESSAGE" )`.
   - Return the data stored at the back of the list; i.e., use the

   `m_ptrLast` to access the Node, then access the Node's `m_data`.

5. void LinkedList<T>::PushFront(T newData)

   - Create a new `DoublyLinkedListNode<T>` by using a pointer. Set its `m_data` to the `newData` passed in as a parameter.
   - Now we have to figure out where it goes:

   (a) Scenario 1: Add item to EMPTY LIST

   cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_emptylist.png?ref_type=heads

   - If the list is empty, then set the `m_ptrFirst` and `m_ptrLast` pointers to your new node.

   (b) Scenario 2: Add item to NOT-EMPTY LIST

   cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_pushfront.png?ref_type=heads

- This node will be our new FIRST node, so we'll have to update the current-first's pointers and the new-first's pointers:
  - Old `m_ptrFirst`'s `m_ptrPrev` should point to the new node.
  - New node's `m_ptrNext` should point to the old `m_ptrFirst`.
  - Set the list's `m_ptrFirst` to the new node.

(c) And then...

- Don't forget to increment `m_itemCount` before the end of the function!

6. void LinkedList<T>::PushBack(T newData)

- Create a new `DoublyLinkedListNode<T>` by using a pointer. Set its `m_data` to the `newData` passed in as a parameter.
- Now we have to figure out where it goes:

(a) Scenario 1: Add item to EMPTY LIST

cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_emptylist.png?ref_type=heads

- If the list is empty, then set the `m_ptrFirst` and `m_ptrLast` pointers to your new node.

(b) Scenario 2: Add item to NOT-EMPTY LIST

cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_pushback.png?ref_type=heads

- This node will be our new LAST node, so we'll have to update the current-last's pointers and the new-last's pointers:
  - Old `m_ptrLast`'s `m_ptrNext` should point to the new node.
  - New node's `m_ptrPrev` should point to the old `m_ptrLast`.
  - Set the list's `m_ptrLast` to the new node.

(c) And then...

- Don't forget to increment `m_itemCount` before the end of the function!

7. void LinkedList<T>::PopFront()

- Error check: If the list is empty, then throw an `Exception::StructureEmptyException( "FUNCTION NAME", "MESSAGE" )`.

(a) Scenario 1: Remove the LAST ITEM

cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_lastitem.png?ref_type=heads

- Delete the node at `m_ptrFirst`.
- Set `m_ptrFirst` and `m_ptrLast` to nullptr.

(b) Scenario 2: Remove the NOT-LAST ITEM

cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_popfront.png?ref_type=heads

- The 2nd item in the list will be our new first node.
- Set `m_ptrFirst` equal to its next node.
- Delete `m_ptrFirst->m_ptrPrev` afterwards.
- Set `m_ptrFirst->m_ptrPrev` to nullptr.

(c) And then...

- Don't forget to decrement `m_itemCount` before the end of the function!

8. void LinkedList<T>::PopBack()

- Error check: If the list is empty, then throw an `Exception::StructureEmptyException( "FUNCTION NAME", "MESSAGE" )`.

(a) Scenario 1: Remove the LAST ITEM

cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_lastitem.png?ref_type=heads

- Delete the node at `m_ptrFirst`.
- Set `m_ptrFirst` and `m_ptrLast` to nullptr.

(b) Scenario 2: Remove the NOT-LAST ITEM

cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_popback.png?ref_type=heads

- The 2nd-to-the-last item in the list will be our new last node.
- Set `m_ptrLast` equal to its prev node.
- Delete `m_ptrLast->m_ptrNext` afterwards.
- Set `m_ptrLast->m_ptrNext` to nullptr.

(c) And then...

- Don't forget to decrement `m_itemCount` before the end of the function!

9. "At" - Walking functionality

   cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_walkB.png?ref_type=heads

   You will need to be able to "walk" through your list for the "At" functions. Here's the general idea:

   (a) Create a walker pointer (sometimes I name this `ptrWalker` or `ptrCurrent`) and point it to the first Node in the list.

   (b) Create a loop that will iterate `index` amount of times. Within the loop, move your walker pointer to its next item.

   (c) When the loop is done, your walker pointer will be pointing at Node #index.

```
DoublyLinkedListNode<T>* ptrWalker = m_ptrFirst;
for ( int i = 0; i < index; i++ )
{
    ptrWalker = ptrWalker->m_ptrNext;
}
```

10. void LinkedList<T>::PushAt(T newItem, int index)

    - Error check: If the index is invalid, then throw an `Exception::InvalidIndexException( "FUNCTION NAME", "MESSAGE" )`.
    - Don't repeat yourself check: If the `index` is 0, then call `PushFront( newItem )` then return.
    - Don't repeat yourself check: If the `index` is `m_itemCount`, then call `PushBack( newItem )` then return.

    cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_pushat.png?ref_type=heads

    If none of the above were triggered, then do the following:

    - Create a **walker pointer**, move to the `index` location.
    - To keep the logic straight, you might want to add some helper pointers:
      (a) `ptrBefore` points to your walker's `m_ptrPrev`.
      (b) `ptrAfter` points to your walker's `m_ptrWalker`.
    - Create a new node, set its `m_data`.

- Update the pointers:
  - `ptrBefore`'s next will be the new node.
  - `ptrAfter`'s prev will be the new node.
  - New node's next will be the `ptrAfter`.
  - New node's prev will be the `ptrBefore`.
- Increment `m_itemCount`.

11. void LinkedList<T>::PopAt(int index)

    - Error check: If the list is empty, then throw an `Exception::StructureEmptyException( "FUNCTION NAME", "MESSAGE" )`.
    - Error check: If the index is invalid, then throw an `Exception::InvalidIndexException( "FUNCTION NAME", "MESSAGE" )`.
    - Don't repeat yourself check: If the `index` is 0, then call `PopFront()` then return.
    - Don't repeat yourself check: If the `index` is `m_itemCount-1`, then call `PopBack()` then return.

    cs250/student-repo/LinkedStructures/https:/gitlab.com/moosadee/courses/-/raw/main/current/cs250/read/images/reading_u14_LinkedList_popat.png?ref_type=heads

    If none of the above were triggered, then do the following:

    - Create a **walker pointer**, move to the `index` location.
    - To keep the logic straight, you might want to add some helper pointers:
      (a) `ptrBefore` points to your walker's `m_ptrPrev`.
      (b) `ptrAfter` points to your walker's `m_ptrNext`.
    - Delete the desired Node:
      - Set `ptrBefore`'s next to `ptrAfter`.
      - Set `ptrAfter`'s prev to `ptrBefore`.
    - Update the neighbor pointers:
      - Delete the node that your walker is pointing to.
      - Decrement `m_itemCount`.

12. T& LinkedList<T>::GetAt(int index)

    - Error check: If the list is empty, then throw an `Exception::StructureEmptyException( "FUNCTION NAME", "MESSAGE" )`.

- Error check: If the index is invalid, then throw an `Exception::InvalidIndexException( "FUNCTION NAME", "MESSAGE" )`.
- Don't repeat yourself check: If the `index` is 0, return `GetFront()` instead.
- Don't repeat yourself check: If the `index` is `m_itemCount-1`, then return `GetBack()` instead.

If none of the above were triggered, then do the following:

- Walk to the Node at the `index` given.
- Return the `m_data` at that Node.

8 Week 8: Algorithm efficiency and search/sort algorithms

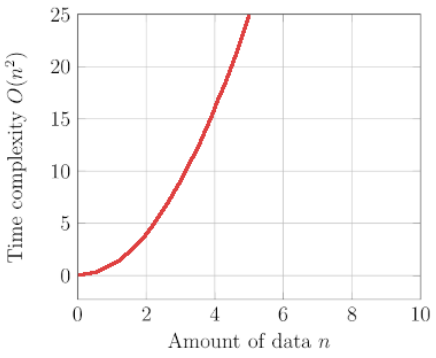
8.1 Intro: Algorithm efficiency

8.1.1 Introduction: Algorithm Efficiency (and why we care)

Processing power continues increasing as time moves forward. Many things process so quickly that we barely notice it, if at all. That doesn't mean we can just write code as inefficiently as possible and just let it run because "computers are fast enough, right?" - as technology evolves, we crunch more and more data, and as "big data" becomes a bigger field, we need to work with this data efficiently - because it doesn't matter how powerful a machine is, processing large quantities of data with an inefficient algorithm can still take a long time.

And some computers today are slow. Not usually the computers that the average person is using (Even though it might feel that way if they're running Windows). Some computers that handle systems are pretty basic, or they're small, and don't have the luxury of having great specs. Fitness trackers, dedicated GPS devices, a thermostat, etc. For systems like these, processing efficiency is important.

Let's say our algorithm's time to execute increases quadratically. That means, as we increase the amount of items to process (n), the time to process that data goes up quadratically.



For processing 1 piece of data, it takes 1 unit of time (we aren't focused so much on the actual unit of time, since each machine runs at different speeds) - that's fine. Processing 2 pieces of data? 4 units of time. 8 pieces of data? 64 units of time. We don't just double the amount of time it takes when we double the data - we square it.

How much data do you think is generated every day on a social media website that has millions of users? Now imagine the processing time for an algorithm with a quadratic increase...

Data to process (n)	**Time to process** ("time units")
1	1
2	4
3	9
4	16
5	25
...	...
100	10,000
1,000	1,000,000
10,000	100,000,000
1,000,000	1,000,000,000,000

From a design standpoint we also need to know what the efficiency of different algorithms are (such as the algorithm to find data in a Linked List or the algorithm to resize a dynamic array) in order to make design decisions on what is the best option for our software.

8.1.2 Example: Fibonacci sequence, iterative and recursive

As an example, you shouldn't use a recursive algorithm to generate a Fibonacci sequence (Each number F_n in the sequence is equal to the sum of the two previous items: $F_n = F_{n-1} + F_{n-2}$). It's just not as efficient! Let's look at generating the string of numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

The most efficient way to do this would be with an iterative approach using a loop. However, we could also figure it out with a recursive approach, though the recursive approach is much less efficient.

Fibonacci sequence, iterative:

```
int GetFib_Iterative(int n)
{
  if (n == 0 || n == 1) { return 1; }

  int n_prev = 1;
  int n_prevprev = 1;
  int result = 0;

  for (int i = 2; i <= n; ++i)
    {
      result = n_prev + n_prevprev;
      n_prevprev = n_prev;
      n_prev = result;
    }

  return result;
```

```
}
```

This algorithm has a simple loop that iterates n times to find a given number of the Fibonacci sequence. The amount of time the loop **iterates** based on n is:

n	3	4	5	6	7	8	9	10	...	20
iterations	2	3	4	5	6	7	8	9	...	19

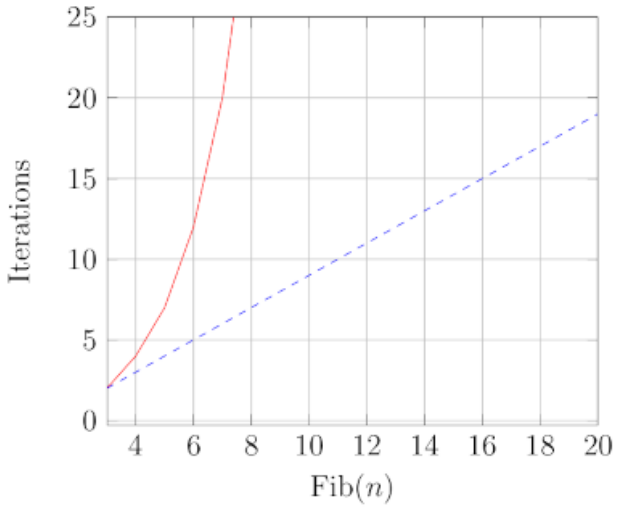
Fibonacci sequence, recursive:

```
int GetFib_Recursive(int n)
{
  if (n == 0 || n == 1) { return 1; }

  return GetFib_Recursive(n - 1) + GetFib_Recursive(n - 2);
}
```

The way this version is written, any time we call this function with any value n, it has to compute the Fibonacci number for $Fib(n-1)$, $Fib(n-2)$, $Fib(n-3)$, ... $Fib(1)$, $Fib(0)$ ... twice. This produces duplicate work b ecause it effectively *doesn't "remember"* what $Fib(3)$ was after it computes it, so for $Fib(n)$ it has to recompute $Fib(n-1)$ and $Fib(n-2)$ each time...

n	3	4	5	6	7	8	9	10	...	20
iterations	2	4	7	12	20	33	54	88	...	10,945

The growth rate of the iterative version ends up being **linear**: As n rises linearly, the amount of iterations also goes up **linearly**. The growth rate of the recursive function is **exponential**: As n rises linearly, the amount of iterations goes up **exponentially**.

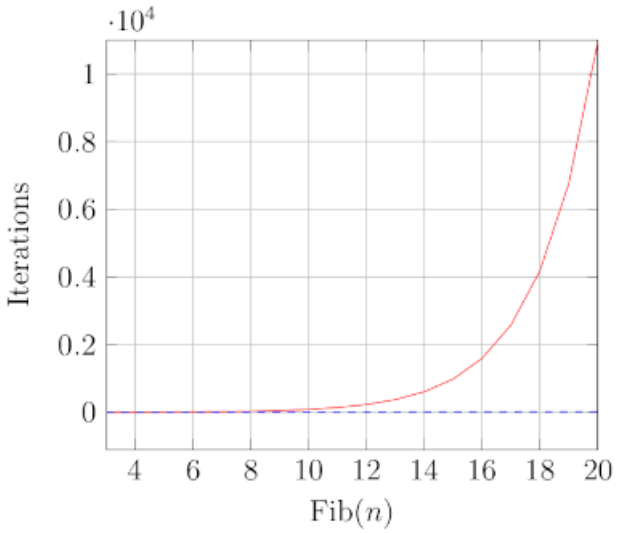


- Red/solid: Recursive growth rate

- Blue/dashed: Iterative growth rate

- Y scale from 0 to 25 (the 0th, to 25th Fibonacci number to generate).

For small amounts this might not be too bad. However, if we were generating a Fibonacci number further in the list, it would continue getting even slower...



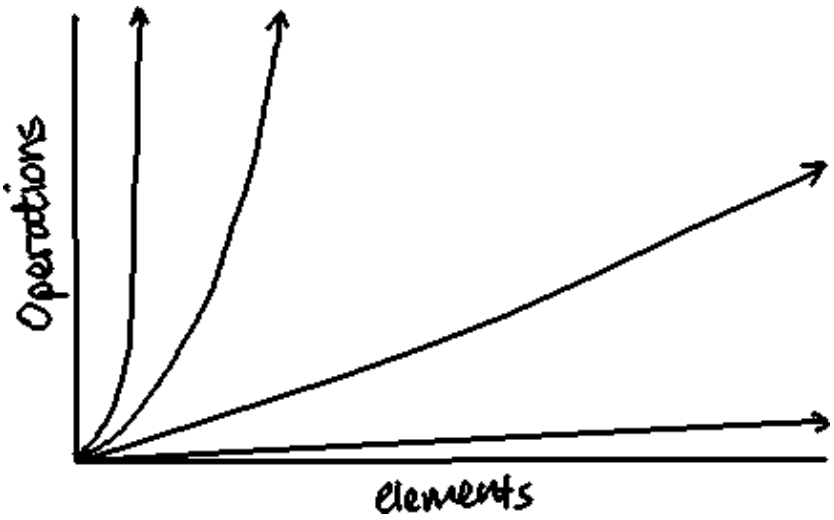
- Red/solid: Recursive growth rate

- Blue/dashed: Iterative growth rate

- Y scale from 0 to 11,000 (the 11,000th Fibonacci number to generate).

If we are trying to generate the 11,000th item in the sequence, the iterative approach requires 11,000 iterations in a loop. That linear increase is still *much faster* than each $Fib(n)$ calling $Fib(n-1)$ and $Fib(n-2)$ recursively.

8.1.3 Big-O Notation and Growth Rates



For the most part, we don't care much about the exact amount of times a loop runs. After all, while the program is running, n could be changing depending on user input, or how much data is stored, or other scenarios. Instead, we are more interested in looking at the big picture: the generalized **growth rate** of the algorithm. (This can include time-to-process growth or space growth.)

We use something called "Big-O notation" to indicate the growth rate of an algorithm. Some of the rates we care about are:

Constant time	$O(1)$
Logarithmic time	$O(log(n))$
Linear time	$O(n)$
Quadratic time	$O(n^2)$
Cubic time	$O(n^3)$
Exponential time	$O(2^n)$

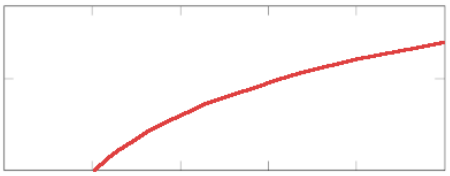
1. Constant time, $O(1)$

We think of any single command as being constant time. The operation $a = b + c$ will take the same amount of computing time no matter what the values of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are.

```
int F(int x)
```

```
{
  return 3 * x + 2;
}
```

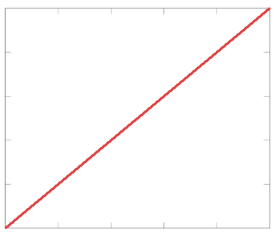
2. Logarithmic time, $O(log(n))$



Having an algorithm that halves its range each iteration of the loop will result in a logarithmic growth rate.

```
int Search(int l, int r,
           int search, int arr[])
{
  while ( l <= r )
    {
      int m = l + (r-l) / 2;
      if ( arr[m] == search )
        { return m; }
      else if ( arr[m] < search )
        { l = m+1; }
      else if ( arr[m] > search )
        { r = m-1; }
    }
  return -1;
}
```

3. Linear time, $O(n)$

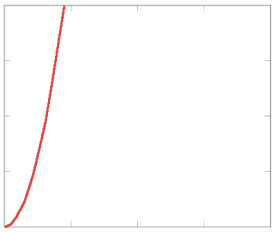


Having a single loop that iterates over a range will be a linear time increase.

```
int Sum( int n ) {
  int sum = 0;
  for (int i=1; i<=n; i++) {
    sum += n;
  }
  return sum;
}
```

If there is some scenario that causes the loop to halve its range each time, then its growth rate would be *less than linear*: as *(log(n))*.

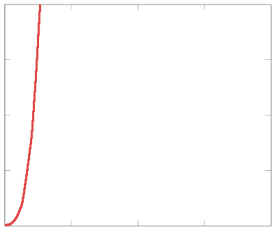
4. Quadratic time, $O(n^2)$



Quadratic time comes into play when you have one loop iterating n times nested within an other loop that also iterates n times. For example, if we were writing out times tables from 1 to 10 ($n = 10$), then we would need 10 rows and 10 columns, giving us $10^2 = 100$ cells.

```
void TimesTables(int n) {
  for (int y=1; y<=n; y++) {
    for (int x=1; x<=n; x++) {
      cout << x << "*" << y << "="
           << x*y << "\t";
    }
    cout << endl;
  }
}
```

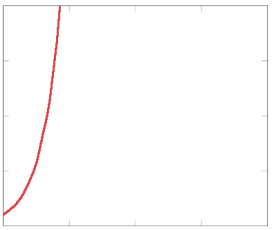
5. Cubic time, $O(n^3)$



Just like nesting two loops iterating n times each gives us a n^2 result, having three nested loops iterating n times each gives us a $O(n^3)$ growth rate.

```
void CubicThing(int n) {
  for (int z=1; z<=n; z++) {
    for (int y=1; y<=n; y++) {
      for (int x=1; x<=n; x++) {
        cout << x << " "
             << y << " "
             << z << endl;
      }
    }
  }
}
```

6. Exponential time, $O(2^n)$



With an exponential function, each step *increases* the complexity of the operation. The Fibonacci example is a good illustration: Figuring out `Fib(0)` and `Fib(1)` are constant, but `Fib(2)` requires calling `Fib(0)` and `Fib(1)`, and `Fib(3)` calls `Fib(1)` and `Fib(2)`, with `Fib(2)` calling `Fib(0)` and `Fib(1)`, and each iteration adds that many more operations.

```
int Fib(int n)
{
  if (n == 0 || n == 1)
    return 1;

  return
    Exponential_Fib(n-2) +  Exponential_Fib(n-1);
}
```

8.1.4 Counting commands

We are more interested in generalizations about efficiency, but for now let's count concrete commands executed that occur with an algorithm.

We can treat a single command (like a variable assignment) as a constant 1, but once we hit a loop, the internal command is executed *1 * n* times.

Let's look at how many commands are executed based on (1) the input value n, and (2) the amount of commands in the algorithm.

And note, we don't count the function header and opening/closing curly braces as a command. Only the contents of the function.

1. Example 1

   ```
   int Func( int n )
   {
     return 3 * n; // +1 command
   }
   ```

   How many commands are executed when...

   - n = 1 ?
   - n = 1 0?
   - n = 100 ?

   This is a **constant** time, so no matter what n is, we will always just be executing *one arithmetic command.*

2. Example 2

   ```
   int Func( int n )
   {
     n += 1; // +1 command

     if ( n > 100 )
       return n+100; // +1 command, or
     else if ( n > 10 )
       return n+10; // +1 command, or
     else
       return n; // +1 command
   }
   ```

   Given the if/else if statement, note that only one of these will be executed. Don't count the if/else if statement itself, just the return commands.

   How many commands are executed when...

   - n = 1 ?
   - n = 10 ?

- n = 100 ?

This is another example of a **constant** time algorithm. Evaluating the "if" statements are essentially negligible, time-complexity-wise.

3. Example 3

```
void Func( int n )
{
  for ( int i = 0; i < n; i++ )
  {
    cout << i << " "; // +1 command each iteration
  }
}
```

How many commands are executed when...

- n = 1 ?
- n = 10 ?
- n = 100 ?

Since we have a loop in here, it runs n commands, so the amount of commands executed ***are*** affected by the n value. In this case, with just the one loop, this is a **linear** time increase.

4. Example 4

```
void Func( int n )
{
  int iterations = 0; // Don't count me
  for ( int i = 0; i < n; i++ )
  {
    for ( int j = 0; j < n; j++ )
    {
      cout << i << "," << j << endl; // +1 command each iteration
      iterations++; // Don't count me
    }
  }
  cout << "Iterations: " << iterations << endl; // Don't count me
}
```

(You can put this code in an IDE to see the result for different n values. Just ignore the items marked `// Don't count me`.)

How many commands are executed when...

- n = 1 ?
- n = 10 ?

- n = 100 ?

Given some n value the **nested** loops will cause some command to be processed n^2 amount of times. This increase is **quadratic**.

8.1.5 Identifying growth rates

We were just counting amount of commands or iterations for a function, but we aren't concerned with these concrete numbers when we are analyzing the efficiency of an algorithm. Instead, we care about generalized growth rate.

Here's a "cheat sheet" for quickly identifying algorithm increases. This is for very general cases, so you may need to study an algorithm more to figure out its actual growth rate.

Growth rate	Big-O notation	Identification
Constant	*O(1)*	Single statements
Logarithmetic	*O(log(n))*	Algorithm halves its work each iteration
Linear	*O(n)*	A for loop
Quadratic	$O(n^2)$	Two for loops nested
Exponential	$O(2^n)$	Algorithm doubles its work each iteration

8.2 Intro: Searching and sorting algorithms

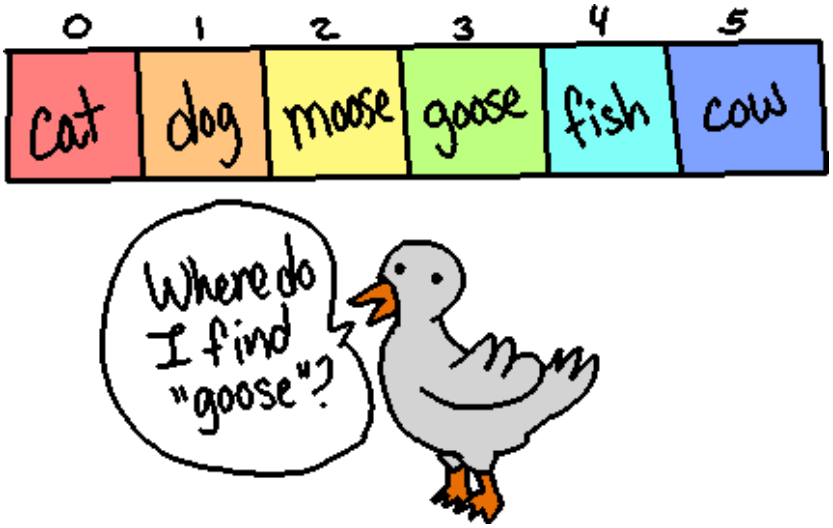
8.2.1 Introduction to searching and sorting

Searching and sorting algorithms are things that we'll study in college, and perhaps see during job interviews, but for the most part in the daily life of an average developer we may not be implementing these algorithms from scratch.

Like data structures, these algorithms have already been implemented and are available in various forms, usually already tested and optimized by someone else, and you can use them in your programs by adding in third party libraries.

I'm just saying this because a some of this functionality of sorting algorithms may not make "intuitive" sense, and I find the code pretty ugly for a lot of these, but you won't have to worry about these in the real world. Unless you decide to go for a masters degree and sorting algorithms are your special area of study, but why would you do such a thing? :/

8.2.2 Searching



We're going to keep this section short for now because we're mostly going to be focusing on sorting algorithms.

1. Linear search

   When we're searching for items in an ***unsorted*** **linear structure** there's not much we can do to speed up the process. We can basically either start at the beginning and move forward, or start and the end and move backward, checking each item in the structure for what you're looking for.

```
template <typename T>
int LinearSearch( const vector<T>& arr, T findme )
{
  int size = arr.size();
  for ( int i = 0; i < size; i++ )
  {
    if ( arr[i] == findme )
    {
      return i; // Item was found at this index
    }
  }

  return -1; // Item was not found; return an invalid index
}
```

   - We begin at the first index 0 and iterate until we hit the last index. Within the loop, if the element at index `i` matches what we're looking for, we return this index.
   - If the loop completes and we haven't returned an index yet that means we've searched the entire structure and have not found the item. In this case, it is not in the structure and we can throw an exception to be dealt with elseware or return something like -1 to symbolize "no valid index".

   This search algorithm's growth rate is *O(n)* – the more items in the structure, the time linearly increases to search through it. Not much we can do about that, which is why we have different types of data structures that sort data as it is inserted - more on those later on.

   (Note, instead of -1, if you want to return a `size_t` to avoid compiler warnings, you could instead return the max value of `size_t` by using `numeric_limits<size_t>::max` from the `<limits>` library.)

2. Linear search - Returning multiple matches

   We could also adjust this linear search so that it returns an array of matches - either matching indices or matching values. This might be useful if we're doing a partial search on a term and want to find all strings that contain our search term. For this, we create an array/vector of matches within the function and add to it when the search term is found. We return this "matches" vector at the end of the function, so it may contain 0 items (nothing found), one item, or multiple items.

```
vector<string> LinearSearch(vector<string> original, string findme)
{
    vector<string> matches;

    for (size_t i = 0; i < original.size(); i++)
    {
        if (StringContains(original[i], findme))
        {
            matches.push_back(original[i]);
        }
    }

    return matches;
}
```

(Note: You could also return `vector<size_t`, to return all indices where a match was found.)

3. Binary search

   OK, but what if the structure *is* sorted?

   We're going to be learning about sorting algorithms, so what if we happen to have a structure that *is* sorted? How can we more intelligently look for some *value* in the structure?

   Let's say we have a simple array like this:

Value:	"aardvark"	"bat"	"cat"	"dog"	"elephant"	"fox"
Index:	0	1	2	3	4	5

   And we want to see if `"dog"` is in the array. We could investigate what the first item is (Hm, starts with an `"a"`) and the last item (`"f"`), and realize that `"d"` is about halfish way between both values. Maybe we should start in the middle and move left or right?

   - Index 0 is `"aardvark"`. Index 5 is `"fox"`. Middle value $\frac{0+5}{2}$ is 2.5 (or 2, for integer division). What is at position 2? – `"cat"`. If arr[2] < findme, move left (investigate arr[1] next) Or if arr[2] > findme, move right (investigate arr[3] next).
   - `"d"` is greater than `"c"` so we'll move right... Index 3 gives us `"dog"` - we've found the item! Return 3.

   In this case, we basically have two iterations of a loop to find "dog" and return its index. If we were searching linearly, we would have to go from 0 to 1 to 2 to 3, so four iterations.

   This still isn't the most efficient way to search this array - just starting at the midpoint and moving left or moving right each time. However, we can

build a better search that imitates that first step: Checking the mid-way point each time.

Here is the code:

```
template <typename T>
int BinarySearch( vector<T> arr, T findme )
{
  int size = arr.size();
  int left = 0;
  int right = size - 1;

  while ( left <= right ) {
    int mid = ( left + right ) / 2;

    if ( arr[mid] < findme ) {
      left = mid + 1;
    }
    else if ( arr[mid] > findme ) {
      right = mid - 1;
    }
    else if ( arr[mid] == findme ) {
      return mid;
    }
  }

  return -1; // not found
}
```

With the binary search we look at the left-most index, right-most index, and mid-point. Each iteration of the loop, we look at our search value `findme` – is its value greater than the middle or less than the middle?

(a) **Example: Binary search on a sorted array**

Let's say we have this array, and we are searching for `'p'`.

Value:	`'a'`	`'c'`	`'e'`	`'h'`	`'i'`	`'k'`	`'m'`	`'o'`	`'p'`	`'r'`
Index:	0	1	2	3	4	5	6	7	8	9

Step 1: `left` is at 0, `right` is at 9, `mid` is $\frac{0+9}{2} = 4$ (integer division).

Value:	`'a'`	`'c'`	`'e'`	`'h'`	`'i'`	`'k'`	`'m'`	`'o'`	`'p'`	`'r'`
Index:	0	1	2	3	4	5	6	7	8	9
	left				mid					right

Next we compare `i` to `'p'`. `'p'` comes later in the alphabet (so `p > i`), so next we're going to change the `left` value to look at `mid+1` and keep `right` as it is.

Step 2: `left` is at 5, `right` is at 9, `mid` is $\frac{5+9}{2} = \frac{14}{2} = 7$.

Value:	`'a'`	`'c'`	`'e'`	`'h'`	`'i'`	`'k'`	`'m'`	`'o'`	`'p'`	`'r'`
Index:	0	1	2	3	4	5	6	7	8	9
						left		mid		right

Now we compare the item at `arr[mid]` `'o'` to what we're searching for (`'p'`). `p > o` so we adjust our left point again to our current midpoint.

Step 3: `left` is at 7, `right` is at 9, `mid` is $\frac{7+9}{2} = \frac{16}{2} = 8$.

Value:	`'a'`	`'c'`	`'e'`	`'h'`	`'i'`	`'k'`	`'m'`	`'o'`	`'p'`	`'r'`
Index:	0	1	2	3	4	5	6	7	8	9
								left	mid	right

Now we compare the item at `arr[mid]` (`'p'`) to what we're searching for (`'p'`). The values match! So the result is `mid` as the index where we found our item.

Each step through the process we **cut out half the search area** by investigating mid and deciding to ignore everything either *before it* (like our example) or *after it*. We do this every iteration, cutting out half the search region each time, effectively giving us an efficiency of $O(log(n))$ - the inverse of an exponential increase.

8.2.3 Sorting

I'll update the text here later for next semester but I never liked sorting algorithms. I always found the approach to studying them really tedious in uni. I'm not going to make you have to figure out these algorithms yourself - the algorithms are online.

I'm just going to give you the code and we can visually step through how they work. It's possible you'll be asked to implement some sorting algorithms in a job interview if the company is really annoying, but for the most part you're going to be using sorting algorithms already implemented in your day-to-day life rather than implementing these yourself from scratch each time.

You'll find animations and stuff on the class webpage that hopefully illustrate it better than we could in a typewritten format.

Sorting algorithm efficiency (From https://www.bigocheatsheet.com/)

Algorithm	Best time	Average time	Worst time
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(nlog(n))$
Quick Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(n^2)$

1. Bubble Sort

   Note that the code you are implementing in class may use different variable names.

   - Visualization of Bubble Sort by Timo Bingmann: https://www.youtube.com/watch?v=Cq7SMsQBEUw

```
template <typename T>
void BubbleSort( vector<T>& arr )
{
  for ( int i = 0; i < arr.size() - 1; i++ )
  {
    for ( int j = 0; j < arr.size() - i - 1; j++ )
    {
      if ( arr[j] > arr[j+1] )
      {
        swap( arr[j], arr[j+1] );
      }
    }
  }
}
```

2. Insertion Sort

    - Visualization of Insertion Sort by Timo Bingmann: https://www.youtube.com/watch?v=8oJS1BMKE64

```
template <typename T>
void InsertionSort( vector<T>& arr )
{
  size_t arraySize = arr.size();
  size_t i = 1;

  while ( i < arraySize )
  {
    int j = i;
    while ( j > 0 && arr[j-1] > arr[j] )
    {
        swap( arr[j], arr[j-1] );
        j = j - 1;
    }

    i = i + 1;
  }
}
```

3. Selection Sort

    - Visualization of Selection Sort by Timo Bingmann: https://www.youtube.com/watch?v=92BfuxHn2XE

```
template <typename T>
void SelectionSort( vector<T>& arr )
{
  int arraySize = arr.size();

  for ( size_t i = 0; i < arraySize - 1; i++ )
  {
    int minIndex = i;

    for ( size_t j = i + 1; j < arraySize; j++ )
    {
      if ( arr[j] < arr[minIndex] )
      {
          minIndex = j;
      }
```

```
    }

    if ( minIndex != i )
    {
      swap( arr[i], arr[minIndex] );
    }
  }
}
```

4. Merge Sort

   - Visualization of Merge Sort by Timo Bingmann: https://www.youtube.com/watch?v=ZRPoEKHXTJg

```
// Declarations
template <typename T>
void MergeSort( vector<T>& arr );

template <typename T>
void MergeSort( vector<T>& arr, int left, int right );

template <typename T>
void Merge( vector<T>& arr, int left, int mid, int right );

// Definitions
template <typename T>
void MergeSort( vector<T>& arr )
{
  MergeSort( arr, 0, arr.size() - 1 );
}

template <typename T>
void MergeSort( vector<T>& arr, int left, int right )
{
  if ( left < right )
  {
    int mid = ( left + right ) / 2;

    MergeSort( arr, left, mid );
    MergeSort( arr, mid+1, right );
    Merge( arr, left, mid, right );
  }
}
```

```
template <typename T>
void Merge( vector<T>& arr, int left, int mid, int right )
{
  const int n1 = mid - left + 1;
  const int n2 = right - mid;

  vector<T> leftVec;
  vector<T> rightVec;

  for ( int i = 0; i < n1; i++ )
  {
    leftVec.push_back( arr[left + i] );
  }

  for ( int j = 0; j < n2; j++ )
  {
    rightVec.push_back( arr[mid + 1 + j] );
  }

  int i = 0;
  int j = 0;
  int k = left;

  while ( i < n1 && j < n2 )
  {
    if ( leftVec[i] <= rightVec[j] )
    {
      arr[k] = leftVec[i];
      i++;
    }
    else
    {
      arr[k] = rightVec[j];
      j++;
    }
    k++;
  }

  while ( i < n1 )
  {
    arr[k] = leftVec[i];
    i++;
    k++;
  }

  while ( j < n2 )
  {
    arr[k] = rightVec[j];
    j++;
    k++;
```

```
  }
}
```

5. Quick Sort

    - Visualization of Quick Sort by Timo Bingmann: https://www.youtube.com/watch?v=8hEyhs3OV1w

```
// Declarations
template <typename T>
void QuickSort( vector<T>& arr );

template <typename T>
void QuickSort( vector<T>& arr, int low, int high );

template <typename T>
int Partition( vector<T>& arr, int low, int high );

// Definitions
template <typename T>
void QuickSort( vector<T>& arr )
{
  QuickSort( arr, 0, arr.size() - 1 );
}

template <typename T>
void QuickSort( vector<T>& arr, int low, int high )
{
  if ( low < high )
  {
    int partIndex = Partition( arr, low, high );
    QuickSort( arr, low, partIndex - 1 );
    QuickSort( arr, partIndex + 1, high );
  }
}

template <typename T>
int Partition( vector<T>& arr, int low, int high )
{
  T pivotValue = arr[high];
  int i = low - 1;

  for ( int j = low; j <= high - 1; j++ )
  {
```

```
    if ( arr[j] <= pivotValue )
    {
      i++;
      swap( arr[i], arr[j] );
    }
  }

  swap( arr[i+1], arr[high] );
  return i + 1;
}
```

6. Radix Sort

   - Visualization of Radix Sort by Timo Bingmann: https://www.youtube.com/watch?v=LyRWppObda4

```
// Adapted from https://www.geeksforgeeks.org/radix-sort/

void CountSort( vector<int>& arr, int n, int exp )
{
  int * output = new int[n];
  int count[10] = { 0 };

  for ( int i = 0; i < n; i++ )
  {
    count[ (arr[i] / exp ) % 10 ]++;
  }

  for ( int i = 1; i < 10; i++ )
  {
    count[i] += count[i-1];
  }

  for ( int i = n-1; i >= 0; i-- )
  {
    output[ count[ (arr[i] / exp ) % 10 ] - 1] = arr[i];
    count[ (arr[i] / exp ) % 10 ]--;
  }

  for ( int i = 0; i < n; i++ )
  {
    arr[i] = output[i];
  }
```

```
  delete [] output;
}

int GetMax( vector<int>& arr )
{
  int max_value = arr[0];
  for ( int i = 1; i < arr.size(); i++ )
  {
    if ( arr[i] > max_value )
    {
      max_value = arr[i];
    }
  }
  return max_value;
}

void RadixSort( vector<int>& arr )
{
  int max_value = GetMax( arr );

  for ( int exponent = 1; max_value / exponent > 0; exponent *= 10 )
  {
    CountSort( arr, arr.size(), exponent );
  }
}
```

7. Heap Sort

    - Visualization of Heap Sort by Timo Bingmann: https://www.youtube.com/watch?v=_bkow6IykGM

```
// https://en.wikipedia.org/wiki/Heapsort

int LeftChildIndex( int index )
{
  return 2 * index + 1;
}

int RightChildIndex( int index )
{
  return 2 * index + 2;
}

int ParentIndex( int index )
```

```
{
  return int( ( index - 1 ) / 2 );
}

void SiftDown( vector<int>& arr, int root, int end )
{
  while ( LeftChildIndex( root ) < end )
  {
    int child = LeftChildIndex( root );

    if ( child + 1 < end && arr[child] < arr[child+1] )
    {
      child++;
    }

    if ( arr[root] < arr[child] )
    {
      swap( arr[root], arr[child] );
      root = child;
    }
    else
    {
      return;
    }
  }
}

void Heapify( vector<int>& arr, size_t size )
{
  int start = ParentIndex( size-1 ) + 1;

  while ( start > 0 )
  {
    start--;
    SiftDown( arr, start, size );
  }
}

void HeapSort( vector<int>& arr )
{
  Heapify( arr, arr.size() );
  int end = arr.size();

  while ( end > 1 )
  {
    end--;
    swap( arr[end], arr[0] );
    SiftDown( arr, 0, end );
  }
}
```

8.3 Lab: Searching and sorting

8.3.1 Assignment and policy info

- Assignment info:
  - Practice and Graded programs
  - Resubmission and regrading policy and procedure
  - Due dates and available until dates
- How To:
  - Build and Run your program in VS Code
  - Turn in your lab
  - Use Git and VS Code
- Quick reference:
  - CS 200 code reference
  - C++ program arguments
  - Debugging with gdb / lldb
  - Common issues on Mac
- Links:
  - R.W.'s courses homepage
  - Assignment direct link

Make sure to reference the Searching and sorting algorithms chapter of the textbook!

8.3.2 Included files:

```
wk08_SearchSort
 graded_program
    HeapSort.h
    main.cpp
    MergeSort.h
    QuickSort.h
    RadixSort.h
    Tester.cpp
    Tester.h
 instructions.html
 instructions.org
 practice1_linearsearch
    LinearSearch.cpp
```

```
   LinearSearch.h
   main.cpp
   native-plants.txt
   Tester.cpp
   Tester.h
practice2_binarysearch
   BinarySearch.cpp
   BinarySearch.h
   Course.cpp
   Course.h
   courses.txt
   main.cpp
   Tester.cpp
   Tester.h
practice3_bubblesort
   BubbleSort.h
   main.cpp
   Tester.cpp
   Tester.h
practice4_insertionsort
   InsertionSort.h
   main.cpp
   Tester.cpp
   Tester.h
practice5_selectionsort
   main.cpp
   SelectionSort.h
   Tester.cpp
   Tester.h
tester.sh
```

8.3.3 Practice programs

1. Practice 1 - Linear search

   (a) Starter code

```
int LinearSearchSingle( const vector<string>& arr, string find_me )
{
    // TODO: Implement me!
    return -1;
}

vector<int> LinearSearchAll( const vector<string>& arr, string find_me )
{
    vector<int> found_indices;
    // TODO: Implement me!
    return found_indices;
}
```

(b) Reference

- Textbook: Searching and sorting
- C++ string find function

```
if ( STRING.find( SECONDSTRING ) != string::npos )
{
    // SUBSTRING FOUND
}
```

(c) Example output

Unit tests:

```
$ ./a.out test

[PASS] LinearSearchSingle( { 10, 20, 30, 40 }, 40 ) should find item
[PASS] LinearSearchSingle( { 10, 20, 30, 40 }, 100 ) shouldn't find item
[PASS] LinearSearchAll( { "cat", "cathode", "tv", "cat5e" }, "cat" )
[PASS] LinearSearchAll( { "cat", "cathode", "tv", "cat5e" }, "dog" )
```

Program:

```
$ ./a.out native-plants.txt Poppy

Searching for "Poppy" in file native-plants.txt...

Single result:
- INDEX: 55: Bush's Poppy Mallow,https://grownative.org/native_plants/bushs-

All results:
- INDEX 55: Bush's Poppy Mallow,https://grownative.org/native_plants/bushs-p

- INDEX 64: Celandine Poppy,https://grownative.org/native_plants/celandine-p

- INDEX 115: Fringed Poppy Mallow,https://grownative.org/native_plants/fring

- INDEX 233: Purple Poppy Mallow,https://grownative.org/native_plants/purple
```

(d) Instructions

i. **LinearSearchSingle** For `LinearSearchSingle` implement linear search. As soon as you find a match within the for loop, return the index `i` immediately.

If we finish the for loop without a return, that means no item was found. Return `-1` after the for loop.

Make sure you're using the string library's `find()` function to check for partial matches!

ii. **LinearSearchAll**

This function will be similar, except we have a `vector<int> found_indices`. Whenever a match is found (again using the string `find()` function), then we push the index `i` into the found indices vector (NOT return).

At the end of the function make sure the `found_indices` vector is returned.

2. Practice 2 - Binary search

(a) Starter code

```
int BinarySearch( const vector<Course>& arr, string find_me )
{
    // TODO: Implement me!
    return -1;
}
```

(b) Reference

- Textbook: Searching and sorting

(c) Example output

Unit tests:

```
$ ./a.out test
[PASS] BinarySearch test 1 - find item
[PASS] BinarySearch test 2 - shouldn't find item
```

Program:

```
$ ./a.out courses.txt CS 200
321 course(s) loaded.

Searching for "CS 200" in file courses.txt...
RESULT: INDEX 303
CS 200 - Concepts of Programming Algorithms Using C++*
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.
4 Hours
```

(d) Instructions

For the Binary Search you will be using exact comparisons using ==, rather than string find. Implement the **BinarySearch** function.

If a match is found, the index (which will be stored at `mid`) will eventually be returned. If no match is found, `return -1;` at the end of the function.

3. Practice 3 - Bubble sort

   (a) Starter code

   ```
   template <typename T>
   void BubbleSort( vector<T>& arr )
   {
       // TODO: Implement me!
   }
   ```

   (b) Reference

   - Textbook: Searching and sorting

   (c) Example output

   ```
   $ ./a.out
   TEST BUBBLE SORT

   [PASS] Check if sort for { 'z', 'o', 'r', 'd', 's' } was correct
   [PASS] Check if sort for { 3, 2, 1 } was correct
   ```

   (d) Instructions

   Working with `vector<T>& arr`, implement the BubbleSort algorithm here.

4. Practice 4 - Insertion sort

   (a) Starter code

   ```
   template <typename T>
   void InsertionSort( vector<T>& arr )
   {
       // TODO: Implement me!
   }
   ```

   (b) Reference

   - Textbook: Searching and sorting

   (c) Example output

   ```
   $ ./a.out
   TEST INSERTION SORT

   [PASS] Check if sort for { 'z', 'o', 'r', 'd', 's' } was correct
   [PASS] Check if sort for { 3, 2, 1 } was correct
   ```

(d) Instructions

Working with `vector<T>& arr`, implement the InsertionSort algorithm here.

5. Practice 5 - Selection sort

(a) Starter code

```
template <typename T>
void SelectionSort( vector<T>& arr )
{
    // TODO: Implement me!
}
```

(b) Reference

- Textbook: Searching and sorting

(c) Example output

```
$ ./a.out
TEST SELECTION SORT

[PASS] Check if sort for { 'z', 'o', 'r', 'd', 's' } was correct
[PASS] Check if sort for { 3, 2, 1 } was correct
```

(d) Instructions

Working with `vector<T>& arr`, implement the SelectionSort algorithm here.

8.3.4 Graded programs

1. Graded program - Merge, Quick, Heap, and Radix sorts

(a) Reference

- Textbook: Searching and sorting

(b) Example output

```
$ ./a.out
MERGE SORT TESTS
[PASS] Check if sort for { 'z', 'o', 'r', 'd', 's' } was correct
[PASS] Check if sort for { 3, 2, 1 } was correct

QUICK SORT TESTS
[PASS] Check if sort for { 'z', 'o', 'r', 'd', 's' } was correct
[PASS] Check if sort for { 3, 2, 1 } was correct
```

```
HEAP SORT TESTS
[PASS] Check if sort for { 'z', 'o', 'r', 'd', 's' } was correct
[PASS] Check if sort for { 3, 2, 1 } was correct

RADIX SORT TESTS
[PASS] Check if sort for { 3, 2, 1 } was correct
[PASS] Check if sort for { 456, 234, 17, 3443, 12323, 112, 1123, 634 } was c
```

(c) Instructions

- Within the **MergeSort.h**, **QuickSort.h**, **HeapSort.h** and **RadixSort.h** implement the corresponding sort algorithm.
- Run the program to make sure the unit tests pass.
- Make sure to reference the textbook.

9 Common general issues

9.1 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.

9.2 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".

10 Common mac-related issues

10.1 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.
~

10.2 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!

---

11 Syllabus

11.1 Course information

College	Johnson County Community College
Division	CSIT (Computer Science/Information Technology)
Instructor	R.W. Singh (they/them)
Semester	Spring 2025 (1/21/2025 - 5/19/2025)
Course	CS 235: Object Oriented Programming using C++ (4 credit hours)
Online Section	Section 300 / CRN 11922 / Online
HyFlex Section	Section 400 / CRN 11923 / HyFlex
Schedule (HyFlex Section)	Tuesdays/Thursdays, 11:00 am - 12:15 pm
Room (HyFlex Section)	Regnier center, room 353
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 continues developing problem solving techniques by focusing on object-oriented styles using C++ abstract data types. Basic data structures such as queues, stacks, trees, dictionaries, their associated operations, and their array and pointer implementations will be studied. Topics also include recursion, templates, fundamental algorithm analysis, searching, sorting, hashing, object-oriented concepts and large program organization. Students will write programs using the concepts covered in the lecture. 3 hrs. lecture, 2 hrs. lab by arrangement/wk. Catalog link: https://catalog.jccc.edu/coursedescriptions/cs/#CS_250

Prerequisites CS 235 or (CS 200 and CS 210 or CS 236 or CS 255 or CIS 240 or MATH 242).

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.

11.1.1 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.

11.1.2 Course delivery

1. Online section

   Section 300 is setup as an **Online** course. This means the following:

   - There is no scheduled class time each week; everything is online asynchronous.
   - Attendance is counted by assignments completed for the assigned week.
   - Video lectures and archives of the *other* section's class will be provided via Yuja.
   - Exams are also given online, through Canvas.

   To see more about JCCC course delivery options, please visit: https://www.jccc.edu/student-resources/course-delivery-methods.html

2. HyFlex section

   Section 400 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:
     (a) In person in the classroom during class time.
     (b) Remotely via Zoom during class time.
     (c) 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

11.1.3 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.

11.1.4 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/)

11.1.5 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?

11.2 Course policies

11.2.1 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_250)

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.

11.2.2 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.

11.2.3 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.

11.2.4 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 20	Welcome, setup, CS 200 review
2	Jan 27	Testing, debugging, templates, friends
3	Feb 3	Software development, Project v1
4	Feb 10	Standard Template Library, Exceptions
5	Feb 17	Intermediate class features, pointers
6	Feb 24	Polymorphism, Static members
7	Mar 3	Smart pointers, Project v2
8	Mar 10	Algorithm efficiency
		Mar 17 - Mar 23 - Spring Break
9	Mar 24	Test-driven Development, Project v3
10	Mar 31	Recursion
11	Apr 7	Design patterns, Project v4
12	Apr 14	Overloading functions, constructors, operators
13	Apr 21	Anonymous functions
14	Apr 28	Linking libraries
15	May 5	Catch-up / project day
16	May 12	Finals week (See JCCC finals schedule)

11.2.5 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.

11.2.6 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.

11.2.7 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.

11.2.8 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/

11.3 Additional information

11.3.1 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)

11.3.2 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.

11.3.3 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.

11.3.4 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

11.3.5 College emergency response plan

https://www.jccc.edu/student-resources/police-safety/police-department/college-emergency-response-plan/

11.3.6 Student code of conduct policy

http://www.jccc.edu/about/leadership-governance/policies/students/student-code-of-conduct/student-code-conduct.html

11.3.7 Student handbook

http://www.jccc.edu/student-resources/student-handbook.html

11.3.8 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.

11.4 Course catalog info

https://catalog.jccc.edu/coursedescriptions/cs/#CS_235

Objectives:

Develop C++ programs using a disciplined object-oriented approach to software development. Create C++ classes using the concepts of encapsulation and data abstraction. Create programs that integrate advanced programming topics. Develop new classes by inheriting existing classes. Implement polymorphism to produce dynamic, run-time applications. Employ commonly accepted programming standards for code and documentation.

Content Outline and Competencies:

```
I.  Software Development
  A. Identify C++ features needed to solve problems in C++.
  B. Define the problem and identify the classes.
  C. Develop a solution.
  D. Code the solution.
  E. Test the solution.
II.  Encapsulation and Data Abstraction
  A. Apply the C++ syntax to define classes.
  B. Use the string class and the vector template classes.
  C. Create copy constructors.
  D. Create friend functions.
  E. Create overloaded operators including the >>, << and = operators.
  F. Implement class composition.
  G. Describe and create constructors and destructors.
  H. Describe use of the "this" pointer.
  I. Implement classes that contain arrays of objects.
  J. Compare static and instance data members in a class.
  K. Compare private and public access specifiers.
III. Advanced Programming Topics
  A. Establish pointers to manage dynamic memory.
    1. Use new and delete operators.
    2. Declare dynamic memory arrays.
    3. Define and use pointers to objects.
    4. Define objects containing pointers.
  B. Create recursive functions.
  C. Organize projects into multiple files.
  D. Handle exceptions caused by unusual or error conditions during program execution.
  E.  Describe and use the C++ I/O system.
  F.  Create templates to develop data independent classes.
IV. Inheritance
```

A. Explain how inheritance can be used to develop new classes.
B. Explain the “is-a” and the “has-a” relationships between classes.
C. Describe and use constructors and destructors for inherited classes.
D. Define the base class.
E. Use inherited classes.
F. Create overloaded, inherited functions.

V. Polymorphism
A. Explain how polymorphism is used to solve programming problems.
B. Create virtual functions.
C. Create abstract classes using pure virtual functions.
D. Explain static binding and dynamic binding.

VI. Code Standards
A. Create descriptive identifiers according to language naming conventions.
B. Write structured and readable code.
C. Create documentation.