3/29/21 - Lecture 1 - Into information

Pretty much basic intro lecture for today…

Boilerplate statement: short, high-level background on something. Not CS-specific.

First three weeks are more intense--”regreasing the programming wheels,” etc.

Get started early!!!

I’m gonna have to finish assignment 1 ASAP…

Quizzes are open notes, open book, open whatever -- IE no excuse to not get 100% :P Two attempts, take the first one to see what you’re at, then look up stuff from the second attempt.

3/31/21 - Lecture 2 - Pointers review

Pointers and such

Pointers are basically variables that can only store addresses. A mailman has an address -- he doesn't know what the house address looks like, etc.

int a = 5; << There will be some location in memory where a is stored.

int* b or int * b or int *b
"Pointer of type int with name of b"
Technically, we're saving enough memory somewhere for an int.

int* b = &a; "the pointer b is pointing to the address of a"

It is **not good practice** to use namespace std. It copies everything you don't need (and the things you need) into your program instead of just what you need.

Dereferencing pointers: say I have int* b
To get the actual value at that address I would use the * to dereference
Like so: cout << *b << endl;

Stack:
Int stack_array[10]; // Allocates space for 10 ints.

Int stack_Array21[5][7];

```
int** heap_Array_2d = new int*[5];
for(int 1 = 0; i < 5; i++){
     heap_Array_2d[i] = new int[7];
}
```

For 1d arrays on heap

Delete[] heap_Array;

For 2d dynamic arrays, you have to loop through and delete each of the inside arrays first

```
for(int i = 0; i < 5; i++){
     Delete[] heap_array_2d[i];
```

```
}
Delete[] heap_Array_2d;
```

Objects:
Objects have sub-objects, little things that are parts of them
Examples:
Car

- Engine
- Wheels
- Etc

WHen programming we often want to group variables together:

- Arrays are a primitive techniche but items must be the same type
- We want more flexibility

Structs:

- We can define custom objects called structs
- Containers which hold many variable types
- Can be used like any other data type, with extra features
- Traditionally structs contain only variables, but classes, which are fancier, also contain functions

How to define:

```
Struct book {
     Int pages;
     String title;
     string* authors;
};

book text_book;
```

Can be declared above main, or below -- but like a

```
Book bookshelf[10]

For (int i = 0; i < 10; i++) {
     Bookshelf[i].num_pages = 100;
     Bookshelf[i].title = "Place holder";
     Bookshelf[i].authors = new string[2];
}

Book text_book;
Text_book.pages = 1000;
Text_book.pub_date = 1874;

Cout << text_book.pages << endl;

book* bk_ptr = &text_book
(*bk_ptr).pages = 1337; //This will dereference the pointer then
access the pages. Parentheses are important!
```

Way most commonly used tho:

```
bk_ptr->pages = 15; //This arrow does the same thing as the above --
it's just clearer and stuff to make it more readable. Its
functionally the same as (*bk_ptr).

book* bk2 = new book;
Same deal here, we have a pointer again.
bk2->pages = 333;

Delete bk2; // This will take care of all the stuffs

book bookshelf[10];
```

Not super important atm but structs are contiguous in memory -- everything is right next to each other in memory.

NOTES ON ASSIGNMENT 1:
Spellbooks are linked. We might have to extract data from it? hmm

4/2/21 - Lecture 3 - Makefile and file separation

File separation:

Interface file: .h

- Header file, sometimes .hpp or no extension
- Tells the world what your code components do

Implementation file:

- .cpp
- Same name as the paired interface file (example.h, example.cpp)
- Actually does the work promised in the .h file -- so .h has the declarations, etc--.cpp has the actual functions and code

Driver file:

- Also .cpp
- Unique program
- Demonstrates functionality
- Sometimes called "client code" (so I suspect this means it is the high-level functionality code--your implementation file holds all the nitty gritty functions you need to use, then the driver file sticks them all together in the main idea of the program.)

File separation:

- Different ways to seperate and group files:
    - By objects
        - book.cpp/.h
            - Struct book[]
            - etc
    - By common functionality
        - Math
        - UI
        - Etc

You never include implementation files!

```
#ifdnef SOMETHING_UNIQUE_H
#define SOMETHIN_UNIQUE_H

//something unique
```

```
#endif
```

So this will read
“If something_unique has not already been defined, then define it.”
Otherwise, it will skip past all of the stuff inside the #define and #endif

Book.h

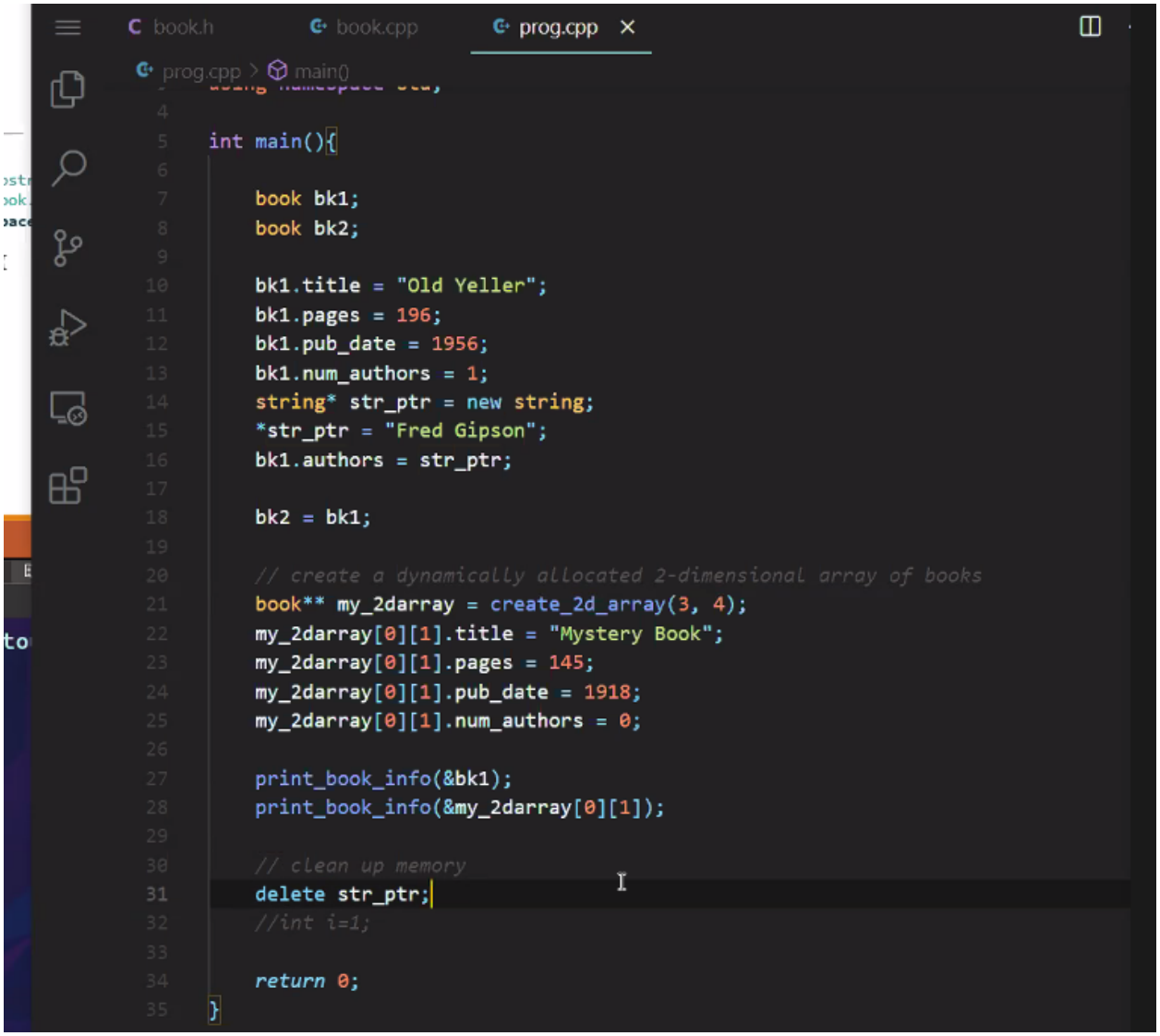
- This has the function declarations etc

Book.cpp

- This holds functions and stuff

Prog.cpp

- Prog holds int main


book.h
book.cpp
prog.cpp
prog.cpp > main()
int main(){
book bk1;
book bk2;
bk1.title = "Old Yeller";
bk1.pages = 196;
bk1.pub_date = 1956;
bk1.num_authors = 1;
string* str_ptr = new string;
*str_ptr = "Fred Gipson";
bk1.authors = str_ptr;
bk2 = bk1;
// create a dynamically allocated 2-dimensional array of books
book** my_2darray = create_2d_array(3, 4);
my_2darray[0][1].title = "Mystery Book";
my_2darray[0][1].pages = 145;
my_2darray[0][1].pub_date = 1918;
my_2darray[0][1].num_authors = 0;
print_book_info(&bk1);
print_book_info(&my_2darray[0][1]);
// clean up memory
delete str_ptr;
//int i=1;
return 0;
}

Things called makefiles
In makefiles we can have a bunch of commands to run

Here's an example makefile:

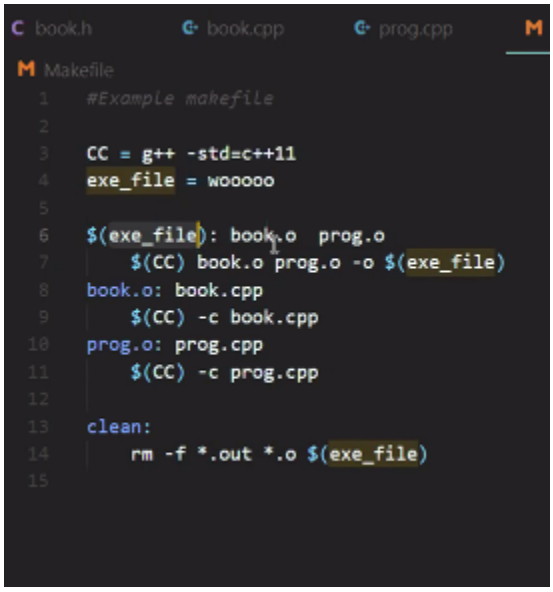
#Hashtags are comments

CC = clang++ -std=c++11
#CC = is the compiler to use, the standard -std is not needed

exe_file = wooo
#thats the executable name

#to use a variable, you do the below
$(exe_file):
echo "Hello world" <<This HAS to be a tab character for the makefile command

//END



So a makeful

.o is adding a middle step, we will learn about later.

Makefiles just run all the commands inside -- so you have it link stuff, compile, etc.

4/5/21 - Lecture 4 - File input/output (yes!)

File I/O

Types of IO we've used before:

- CIN, COUT, CERR (CERR is error stuff)

We have to create file stream objects before we can use file IO.

General algorithm:

1. Create file object
2. Open the file
3. Perform file action (read or write)
4. Close the file

Example:

```
#include <fstream>

Int main(){
     fstream f;
     ifstream fin;
     ofstream fout;

     Return 0;
}
```

Example of using the class/filestream:

```
Int main(){
     fstream f;
     ifstream fin;
     ofstream fout;

     f.open("filename.ext"); //IE cout.txt
     //Now we have our filename attached to the variable f.

     //Or, we could do this:
     fstream f("cout.txt"); //This does it in a single line.
```

f("file.ext") can take BOTH c-string and std::string types. There was some confusion in class.

```
    f.open("file.txt", ios::in); //Opens a file for input
    f.open("file.txt", ios::out); //Opens file for output

    f.open("file.txt", ios::ate); //


    f.is_open()
    //This is a boolean that checks if the file is successfully
    //opened.

    //We can use this to give error messages.

    //We also have

    f.fail(); //Returns true if the file open has failed.
    Pessimistic sibling to f.is_open().

    //do stuff to fi
    return 0;
}
```

File I/O – Open the file

- Use filename as parameter, mode optional
  - Syntax: open(filename, mode)
  - 2 ways

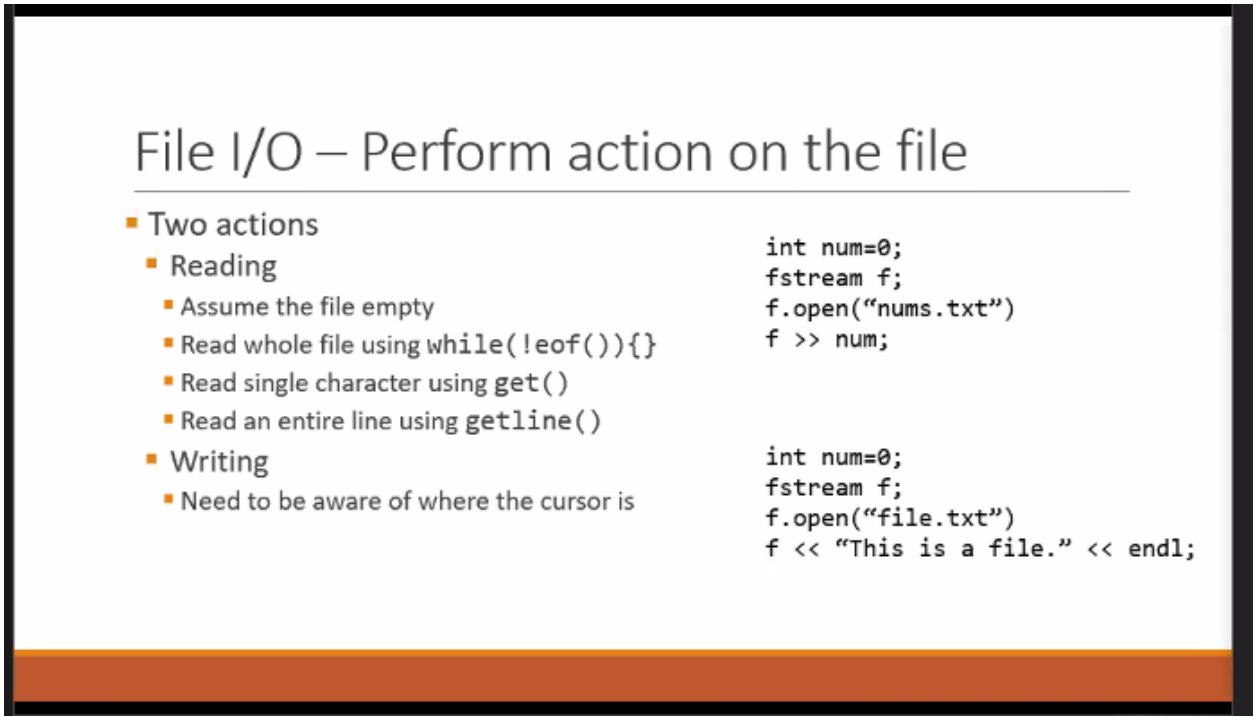
```
fstream f("file.txt");
```

```
fstream f;
f.open("file.txt")
```

- Modes
  - ios::in – open file for input
  - ios::out – open for output
  - ios::binary – open file in binary mode
  - ios::ate – opens a file and puts the output position "at the end"
  - ios::app – opens a file and appends to the end
  - ios::trunc – deletes the existing file contents

```
f.open("file.txt", ios::in);
```

Actual file I/O -- reading / writing:



It looks like f.getline might be easiest to use for Assignment 1…

We could use a while loop and eof() to know when we've reached the end. Or, each spellbook has the info that tells us how many lines to read (IE, this many spellbooks, this many spells per book… So we don't need to use eof.)

f.get() returns a **single** character. How useful :P

We can also do things like

```
Int num = 0;
F >> num;
Cout << num;
```

F << "This is a file." << endl; //But we need to be aware of the cursor position…

Closing the file:

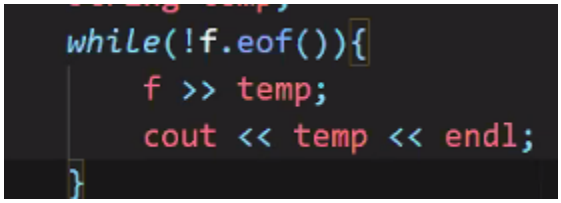
```
f.close();
```

Parsing files:

Typical common parsers:

- Newline \n
- Comma
- Tab char
- Colon
- Space

Aha!! getline() and >> use spaces to parse! They will break at spaces in other words. Very useful for assignment 1



Example code ^

So basically this sentence would be read like so.

> "So"
> "basically"
> "this"
> "sentence"
> "would"
> "be"
> "read"
> "like"
> "so."

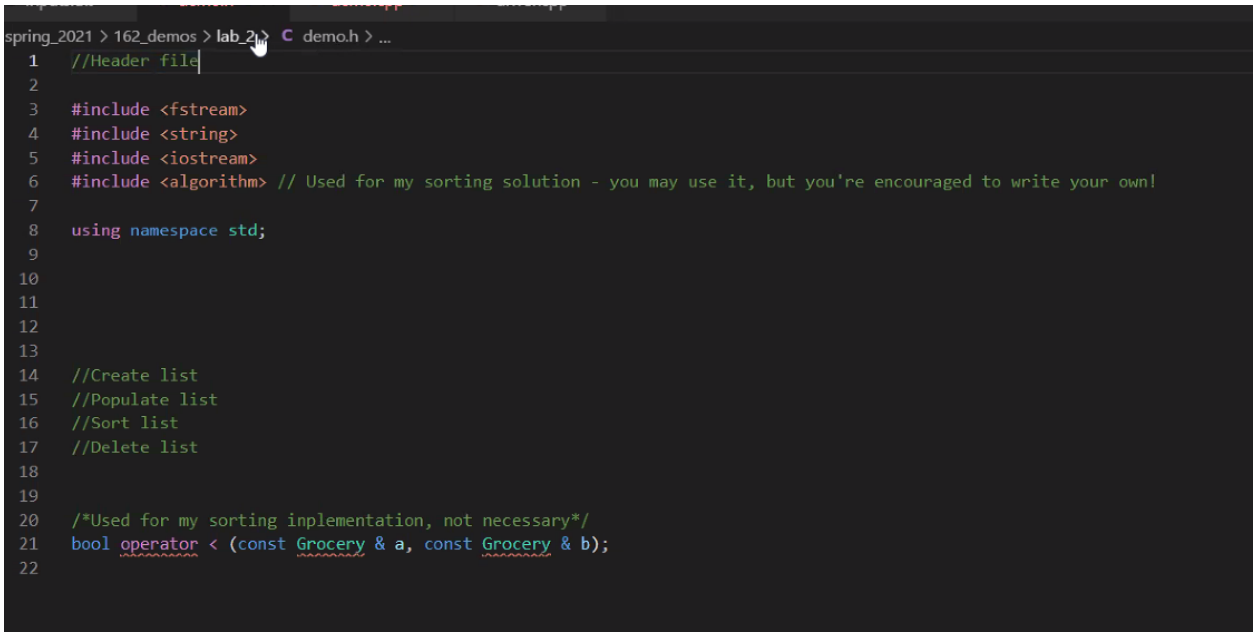
Different parses:

We can use getline like so:

```
getline(filestreamobject, stringtoreadinto, parsechar);
//So this is used like
getline(f, title, ',');

//Obviously we can change the parsing character (delimiter) however we want.

//This is useful if we know the structure of a file already. Getline will also ignore the delimiter.

fout << "Hello world!" << endl;
```

Assignment 1 notes: We can use the c++ standard sorting library. I don't know anything about that so we might have to look up how to use it. Probably would make assignment easier if we can learn how to use it.

That's inside #include<algorithm> //but don't overuse it :P



4/7/21 - Lecture 5 - Objects

Objects!!

- Piece of memory for holding values

Create objects that combine both properties and behaviors

- Properties: name, age, major
- Behavior: sleeping, talking, eating, programming

Why use OOP?

- Easier to read/write because it makes the subject of the behavior clearer
- Enables us to use inheritance, polymorphism, encapsulation, and abstraction.

```
object you;
object shark;
shark.eat(you);
```

Fundamental building block:

- Classes:
  - User defined datatype
  - Similar to structs
  - Has both member variables (like structs) and **member functions** (unlike structs)

Why use classes?

- Adds functionality
- Same as OOP reasons

Example struct and usage:

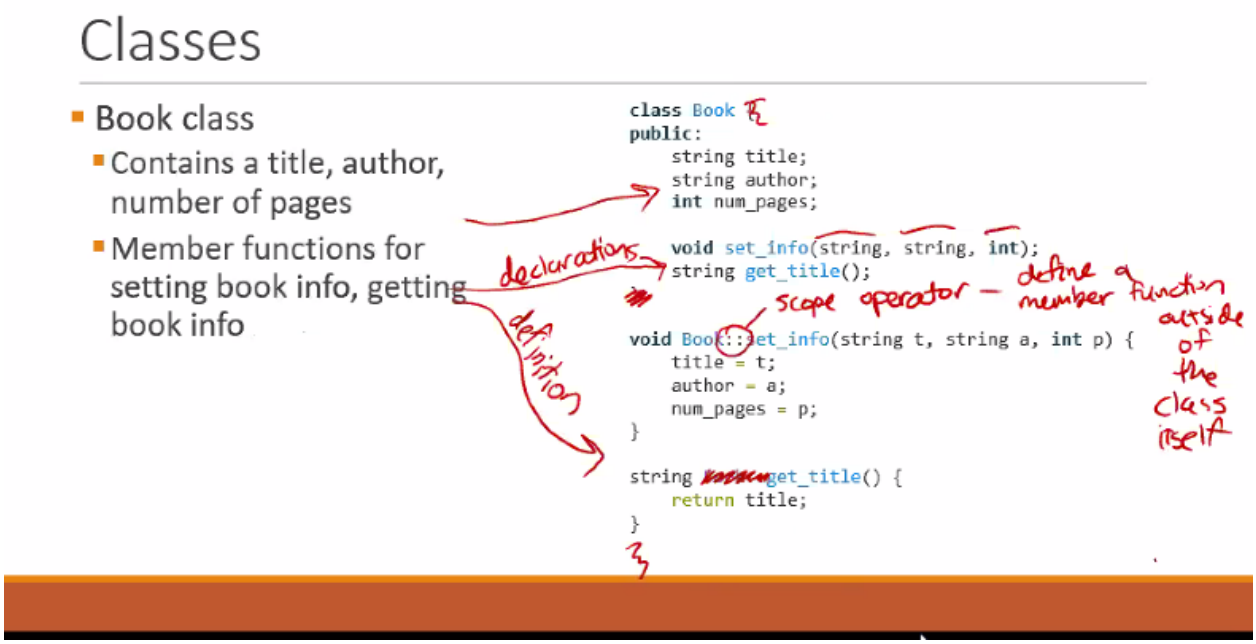
```
Class Book {
      String title;
      String author;
      Int pages;
      Void print_info(){
            //print stuff
            Cout << title << endl;
            Cout << author << endl;
            Cout << pages << endl;
      }
```

```
};

Book b;
b.print_info();
```

Ok! So when we have a function tied to a class, it can access all those class variables as if they were global variables! IE

```
Class Book {
Public: 
```
When we put public here, it acts the rest of the way like a struct. Otherwise, things like the inner variables cannot be accessed from outside. So if you want to be able to access inner variables besides in the inner functions, we need to declare those public with the public: keyword. Anything after that will be accessible.
```
      String title;
      String author;
      Int pages;
      Void print_info(){
            //print stuff
            Cout << title << endl;
            Cout << author << endl;
            Cout << pages << endl;
      }
};


Class Book {
      String title;
      String author;
      Int pages;
      Void print_info(){
            //print stuff
            Cout << title << endl;
            Cout << author << endl;
            Cout << pages << endl;
      }

      Void set_info(string t, string a, int p){
            Title = t;
            Author = a;
            Pages = p;
      }
};
```

```
Book b2;

b2.set_info("Goodnight moon", "Margaret Wise Brown", 12);
```



Apparently we'll sometimes see functions defined outside the scopes of a class.

```
Class Book {
public:
      String title;
      String author;
      Int pages;

      Void print_info();
      Void set_info(string, string, int);
};

Void Book::print_info(){
      //print stuff
      Cout << title << endl;
      Cout << author << endl;
      Cout << pages << endl;
}

Void Book::set_info(string t, string a, int p){
      Title = t;
```

```
     Author = a;
     Pages = p;
}
```

So when we do this, the Book:: (or more generally, the Class::) keyword tells the compiler that it belongs to the class Book, but then we also need to put function prototypes inside the class definition.

This works because it's all still public.

Different demo

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

Class Point{
Public:
     Int x
     Int y;
     Int z;

     Void move_left(int); // Example of the kind of intrinsic
behavior of a point
};

Void Point::move_left(int delta){
     x = x - delta;
}

Int maint(){

     Point p1;
     P1.x = 8;
     P1.y = 4;
     P1.z = 12;

     Cout << p1.x << p1.y << p1.z << endl;

     Return 0;
}

//-==-==-==-==-==-==-==-==-==-==-==-==-= end of code
```

Best practices for working with classes is **not** to make the variables public. We'd prefer to hide the details from other parts of the program -- IE, so that variables can't be directly accessed, or so that there aren't conflicts in the program.

Benefits:

- Code maintainability (avoid broken code) (imagine you change variable names that people reference -- then it breaks)
- Um

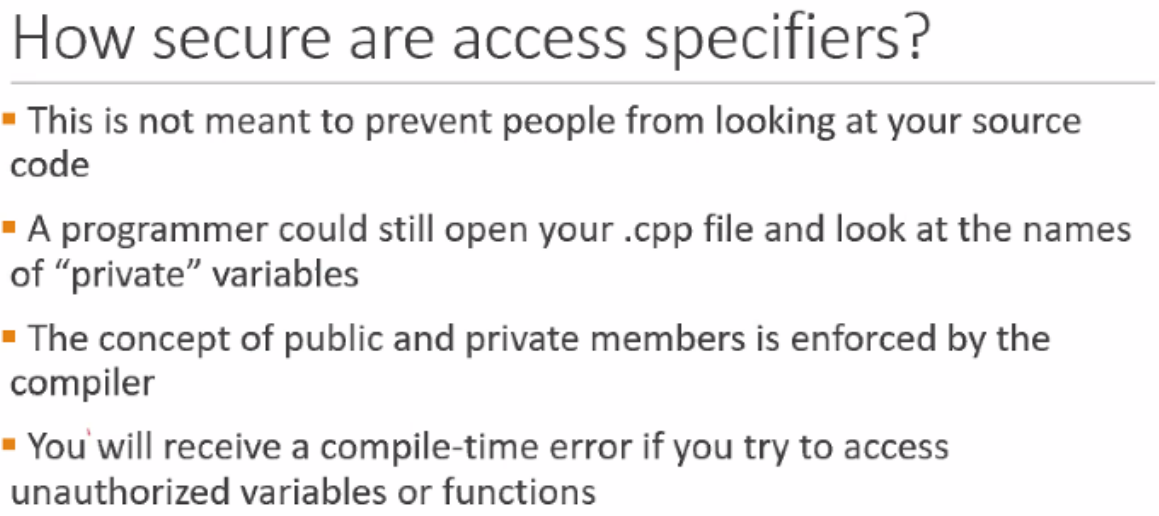
There are functions we use to retrieve or change variable values inside classes. Often called "getters" and "Setters"

```
Class Point{
Private: // This is redundant, but it's nice to make it clear that it's private to anyone reading the code.
      Int x
      Int y;
      Int z;

public:
      Int get_x();
      Int get_y();
      Int get_z();

      Void set_x(int;
      Void set_y(int);
      Void set_z(int);
};

Void Point::set_x(int newx){
      X = newx;
}

Int Point::get_x(){
      Return x;
}

//the rest are similar.
```

Now insteasd of p1.x, we want users to type p1.get_x() and p1.set_x(5) instead of p1.x = 5;

Typically, we will make our member variables private for a class, and our member functions public.

Getters and setters are properly named (formally) accessors and mutators

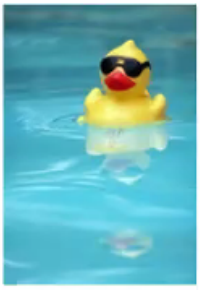


Ke
Keeeeee

4/9/21 - Lecture 6 - Classes more?

Tips & Tricks

- If you've been staring at your code for a long time, take a short break (check out the Pomodoro method)
- If you're trying to think through a problem
  - rubber duck it
  - Draw it out
- Save and test frequently, with comments
- Don't copy...
- Check that all files needed are actually in your tarball before AND after you submit, and that it compiles on the ENGR servers



Assignment 1 is due this weekend!

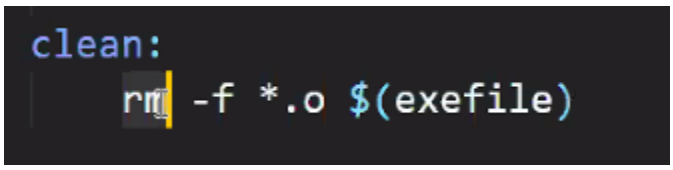
Classes:
Typically written with their own header .h file and implementation .cpp files

- Point.h - contains class definition and member function prototypes and variables
- Point.cpp - contains the member function definitions
- Prog.cpp - driver file still the same, but it includes the .h for the class.

Basically, the class is similar to the implementation files, and also has to be included. I suppose we could include the class file in the main .h in our implementation file?

Wait we can add a clean thing to our makefile!

```
clean:
	rm -f *.0 $(exefile)
```



This says “delete anything with the extension .o”

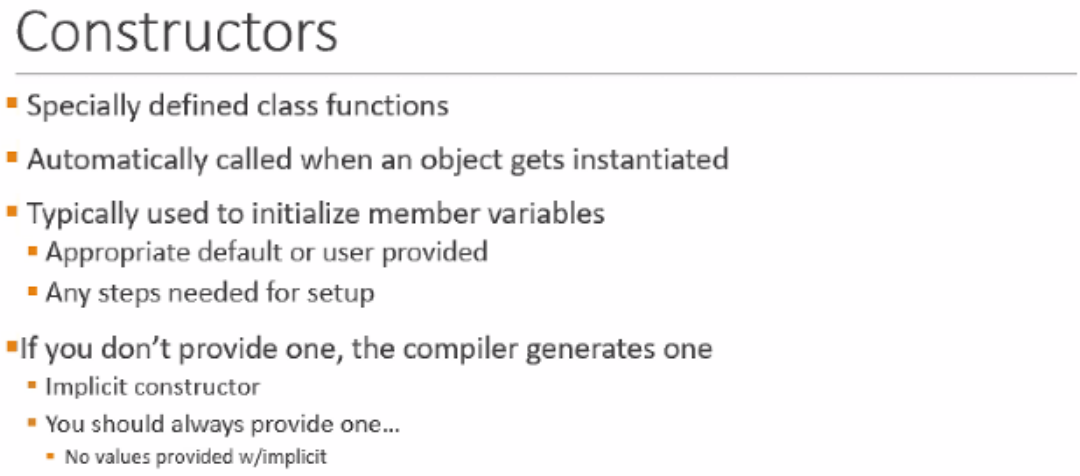
We can add a tar command!

```
tar:
     tar -cvf assign1.tar *.cpp *.h Makefile
```

4/12/21 - Lecture 7 - Constructors (classes stuff still)

Get stuff checked off ASAP because then if something goes wrong we have late days to fix stuff with :P

One annoying thing about classes with private variables is having to use their setter functions to initialize them all / have to pass in 10 things to one function that does them all, it's annoying! We want to reduce lines of code.



Constructors:

- These are special functions called at the very beginning initialization of an object. It will set things up -- IE, default values, or other things to set up.

Code: -=-

```
Class point {
Private:
     Int x;
     Int y;
     Int z;

Public:

     Point();
```

// This is the constructor. There is no return type--nothing is returned! Not even void. That's part of how it gets recognized.
//you can also have multiple constructors!
//you can also have parameters in the constructor:

```
Point(int, int, int);
```

You can't call these constructors with the classic . operator, like Point p2.Point(); < That is not legal

There are different types of constructors (i guess and we might want to change which one we use depending on stuff)

```
Point(); //Default constructor
(No parameters, put most default values here)

Point::Point(){
      //Just put default stuff here.
      X = 0;
      Y = 0;
      Z = 0;
}

Point(int, int, int); //Parameterized constructor

Point::Point(int newx, int newy, int newz){
      //Just put default stuff here.
      X = newx;
      Y = newy;
      Z = newz;
}

To call that:
Point p3(3, 5, 7);
```

```
Point::Point(int a, int b){
    x = a;
    y = b;
}

Point::Point(int a, int b):x{a},y{b}{}
```

You can also do the above as shown.

You can also have multiple constructors I think

Reducing Constructors

- Want to minimize the number contructors
  - Some can be redundant
- Can set default values in a constructor that accepts parameters
- Still considered a default constructor
  - But can accept a couple user provided values as well
  - Follows same rules as function defaults

```
Point::Point(int a=0, int b=0){
    x = a;
    y = b;
}

int main() {
    Point p1;
    Point p2(8,4);
    Point p3(7);

    return 0;
}
```

So we can just make a default that can be set if needed.

Let's say you have a class that has a member variable that is a member of another class:

(Kinda like the spell struct inside the spellbooks struct in As. 1)

When your constructor gets to the variables that are in the other class, they call that class' default constructor.

So you want to write a default constructor, and one that accepts parameters so we can pass things down and such.

Constructors

- Don't create objects
  - Compiler sets up memory for the object before constructor is called
- Determine who is allowed to create objects
  - Object can only be made if there is a matching constructor
- Best practice to initialize all member variables on creation of object
  - Constructors
- Don't use a constructor to re-initialize an object
  - Compiler will create a temporary object and then discard it

Also, from a constructor, we can call other functions! IE, we can call set(int x, int y, int z) from inside the constructor and so we can reduce duplicate code. Also good so we don’t have to change multiple pieces of code when we change how we do one thing.

4/14/21 - Lecture 8 - Const, static, destructors

```
#include “point.h”
#include <iostream>

Using namespace std;

Int main(){


     Return 0;
}
```

Last time… Constructors:

- 3 types:
    - Implicit
    - Default
    - Accepts parameters
- Special class member function that is called automatically and only once when class is creates
- … etc

Passing objects:

- Can be passed by value just like any other variable
    - Value `void print_info(Point p1){}`
    - Pointer `void print_info(Point* p1){}`
    - Reference `void print_info(Point& p1){}`

- Generally we pass by reference or address
    - More efficient usually
    - Pass by value makes two copies so its less efficient
    - Pass by reference uses the OG variable and can’t be NULL
    - Can be problematic since changes to references / pointer persist (IE changes made in a function are permanent, not done in a copy).

Special Pointers - NULL

- NULL Pointer is a constant with a value of zero
- Special macro inherited from C
- Good practice to assign the pointer NULL to a pointer variable in case you don't have the exact address to be assigned
- Avoid accidental misuse of an unitialized pointer
  - Garbage values can make debug difficult
- In C++11, use nullptr instead of NULL

Special pointers: this

- Implicit *this
  - What the compiler uses when calling class functions associated with objects
  -

```
//what you see
Point p1;
p1.set_location(4, 8);

//what the compiler sees
Point p1;
set_location(&p1, 4, 8);
```

```
//what you see
void set_location(int new_x, int new_y){
    x = new_x;
    y = new_y;
}
//what the compiler sees
void set_location(Point* const this, int new_x, int new_y){
    this->x = new_x;
    this->y = new_y;
}
```

Useful behind the scenes, but we can also use the this keyword to do fun stuff…

```
Point& point::move_left(int delta){
     x = x - delta;
     Return this*;
}
```

So then we can do stuff like so:

```
p1.move_left(3).move_up(7);
```

Const Objects:

- Create an object that **cannot** be changed
- Instantiated class objects can be make constant:
  - const Point p1(50, 50);
  - const Point p2; //calls default constructor
- Can't modify any member variables after constructor
  - Neither directly or via a setter function

Const references and Functions:

- To prevent changes to an object being passed, put const in the parameter listing
  - bool isGreater(const Point&a, const Point&b);
- If a function isn't supposed to change anything, put a const at the end
  - void print() const;
  - void Point::print() const {/* definition here */}
- Will cause an error if anything in print() code changes stuff
- If you use const for one parameter of a particular type, then you should use it for every other parameter of that type that is not changed by the function call
- Const can't be a member variable of a class

Static variables:

- A variable shelled by **all objects of the same class**
  - EX: static int count;
- Allowed to be private (but does not have to be)
- Permits objects of the same class to communicate with each other
- Must be initialized outside of the class function
  - EX: int Point::count = 0;
- The person writing the class does this
- Static variables cannot be initialized twice

You can have static functions:

- They cannot use non-static things
- They are not attached to an object
  - (No this* pointer)

Example:

Example

```
static int getCount();
int Point::getCount() { //note: no static keyword
      count++; return count;
}
```

Function call: `Point::getCount();`

4/14/21 - Lecture 9 - Const, static, destructors, more

Static variables:

- They don't get a unique instance for each class object
- So basically ALL classes of that type will share the same variable -- IE sort of a shared mailbox if you will.
- It must be initialized outside of the class function
- It can be private or public, being private doesn't make it not accessible to all members of the function, just to us.

```
static int count; //You define inside the private/public like so
```

Then we initialize it"

```
int Point::count = 0; //The static keyword doesn't show up here!
```

To access it, we can't call it with a point (p1.count) like we would other variables or functions in the class.

Instead:

```
point Point::get_count(){
     return count;
}
```

So we could do that if we like, making a getter

Static Functions

- Can have static functions
  - They cannot use non-static things
  - They are not attached to an object
    - AKA, no this* pointer
- Example

```
static int getCount();
int Point::getCount() { //note: no static keyword
     count++; return count;
}
```

Function call: `Point::getCount();`

Destructors:

- Destructor gets automatically called whenever an object goes out of scope.
- Opposite of the constructors.
- Cannot return
- CANNOT take arguments
- Always in the form below
- Must have the ~
- Must have same name as class

Define like so:
~Point(); (only difference between constructor and destructor types is the tilde)

Constructors are only called when we allocate a non-pointer type. If we were to

Another example, the above would only destructor line 7, not the others. We never called delete[] for the one on line 9, so…

Class Line{
Private:
Int num_points;
point* points;

Public:

Line();

```
Line(int);

Int get_num_points();
point* get_points();

Void set_num_points():
Void set_points(point*);

Void set_point_specific(int index, point* point&);

~Line();

Tada the end for now
```

4/19/21 - Lecture 10 - The Big Three

Started with a plug for the linux club...

The Big Three (whatever that is…)

Destructors:

~Point()

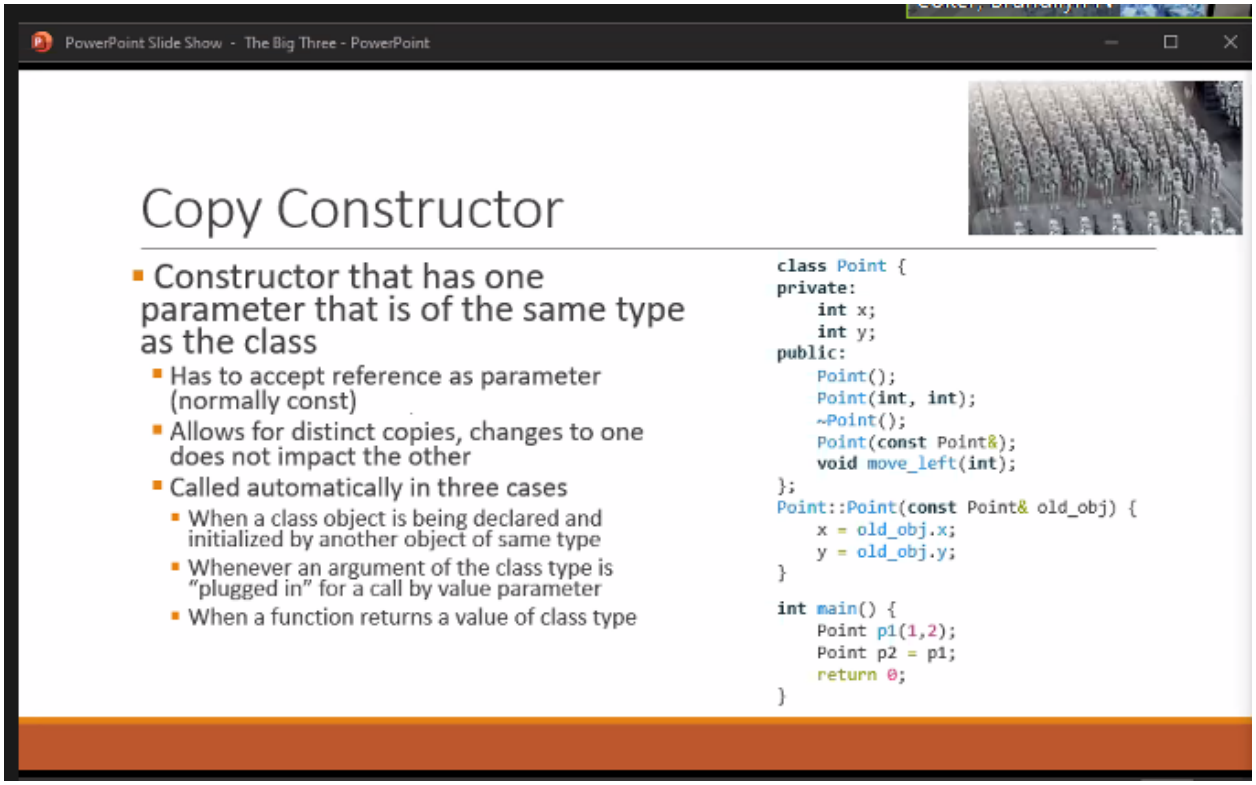
- Only one destructor per class
- Opposite of constructor
- Etc, mostly review from last time

When are they useful?

```
Class Line{
      Point* points;
      // … other stuff too
}

//Default constructor
Line::Line(){
      num_points = 2; //There must be at least two points
      points = new Point[num_points];
      points[num+points-1].set(1,1,1);
      cout << "Default line constructor" << endl;
}

//Parameterized constructor
Line::Line(int n){
      num+points = n;
      points = new Point[num_points];

      for(int i = 0; i < num_points; i++){
            points[i].set(i, i, i);
      }
}

//Destructor - Cannot handle dynamic stuff.
Line::~Line(){
      delete[] points;
      cout << "LINE DESTRUCTO" << endl;
}
```

```
int main(){

      Line l1;

      Line l2(4);
//THIS IS HOW YOU PASS IN PARAMATERS FOR A PARAMATERIZED CONSTRUCTOR!

      return 0;
}

//-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

COPY CONSTRUCTOR:

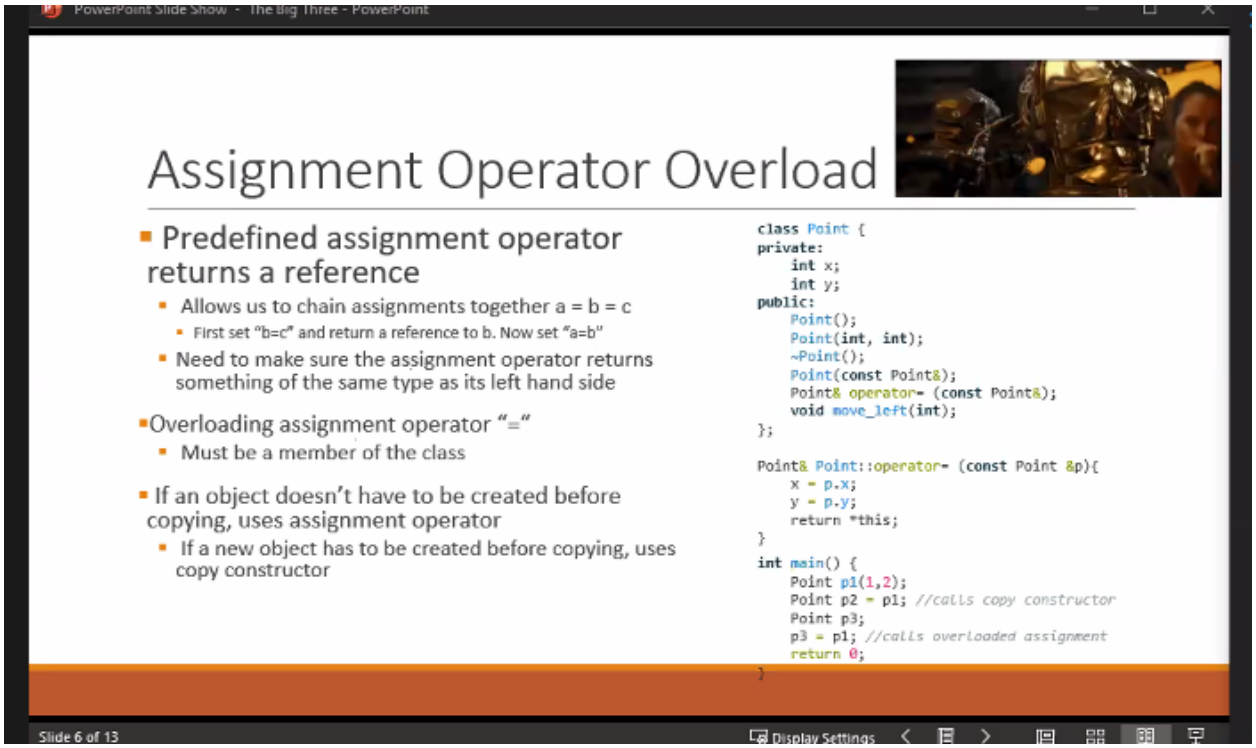


- Same name as class
- **REQUIRED** to have a parameter that is same type as the class.
- Still doesn't return anything.
- Normally the parameter is a const
- There is one that is explicitly there -- a default copy constructor. Those are fine, AS LONG AS we aren't dealing with dynamic memory.

Let's say we want to create a copy of an old line:

```
Line l3 = l1; //Copy constructor is used.

//Copy constructor is also used when you pass something in by value.


Line::Line(const Line& old_line){
      num_points = old_line.num_points;
      points = new Point[num_points];
      for(int i = 0; i < num_points; i++){
            points[i] = old_lin.points[i];
      }
      cout << "Line copy constructor" << endl;
}
```

THE LAST OF THE BIG THREE: THE ASSIGNMENT OPERATOR OVERLOAD:



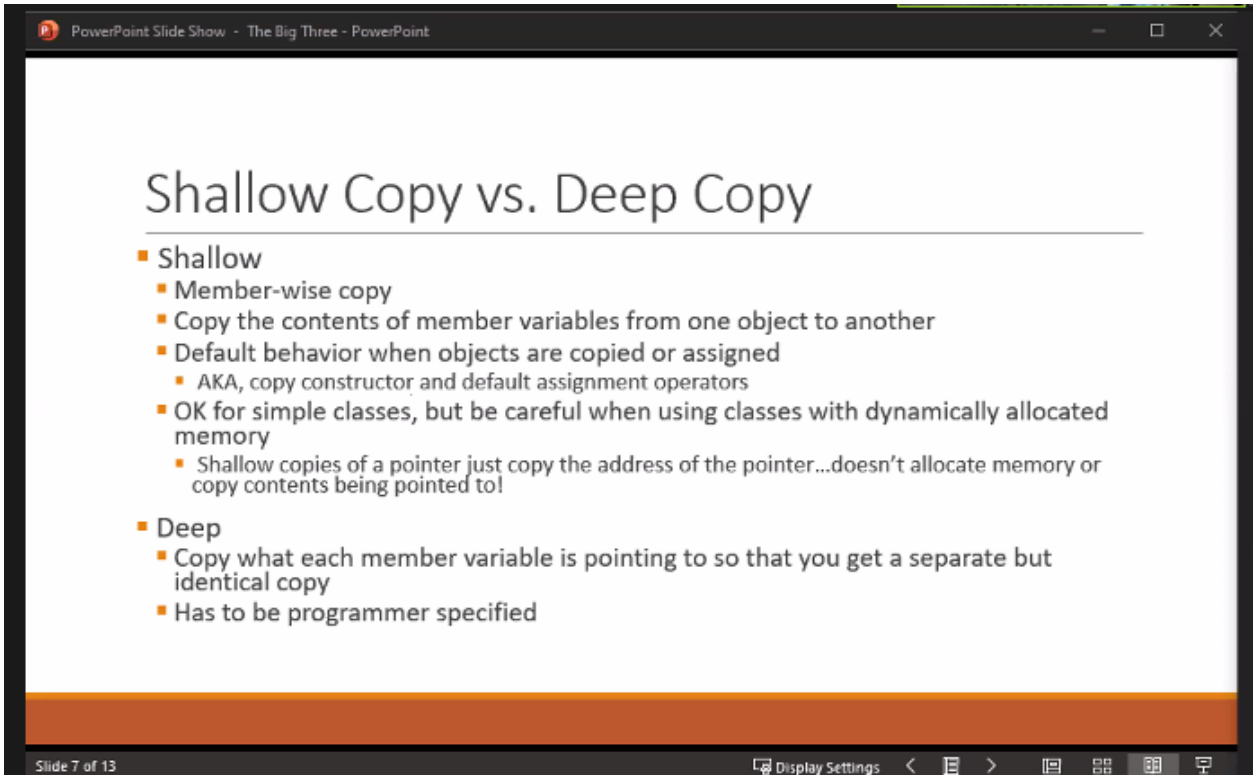
- Very specific syntax to be recognized as an overload.
- Needs to return a reference to the object
-

```
Line& operator=(const Line&); //This is called when both have already
been created, unlike the copy constructor.
```

```
Line& Line::operator=(const Line& old_line){
      Num_points = old_line.num_points;
      //We can't just do exactly what we did in the last time
      //Cause the lines may not be the same size.

      delete[] points; //The one for the object being copied into

      points = new Point[num_points];

      for(int i = 0; i < num_points; i++){
            points[i] = old_lin.points[i];
      }

      return *this;
}
```

Then in main():

```
l2 = l1;
```



The default assignment operator is shallow, and it would try to do this:

```
points = old_line.points;
```

Works great for static stuff, not for dynamic stuff. Dynamic, we use pointers. The issue is that we'd have two point arrays pointing to the same exact address for the same dynamic thing, no longer independent.

Deep copying is what we've done with the dynamic memory copying.

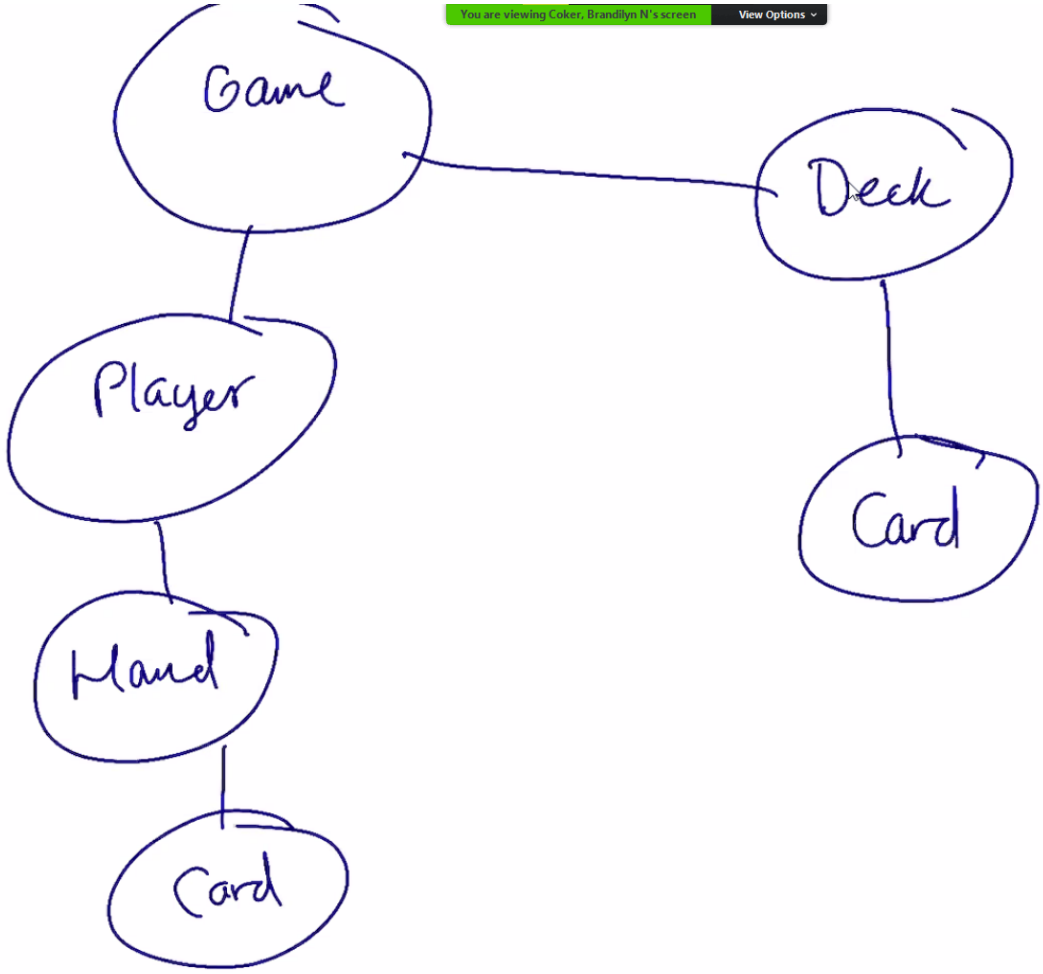
4/21/21 - Lecture 11 - In-class activities

But first some stuff **about** program 2:

Try to keep hand class independent of deck class. IE, we don't want to have to include the Deck class.

So we need an "add card" -- but it has no clue what it's coming from. IE, it just wants to be called to get a card.

Object relationship map:



Game knows Player and Deck

Deck knows Card

Player knows Hand
Hand knows Card

4/23/21 - Lecture 12 - Relationships between objects

Composition and inheritance:

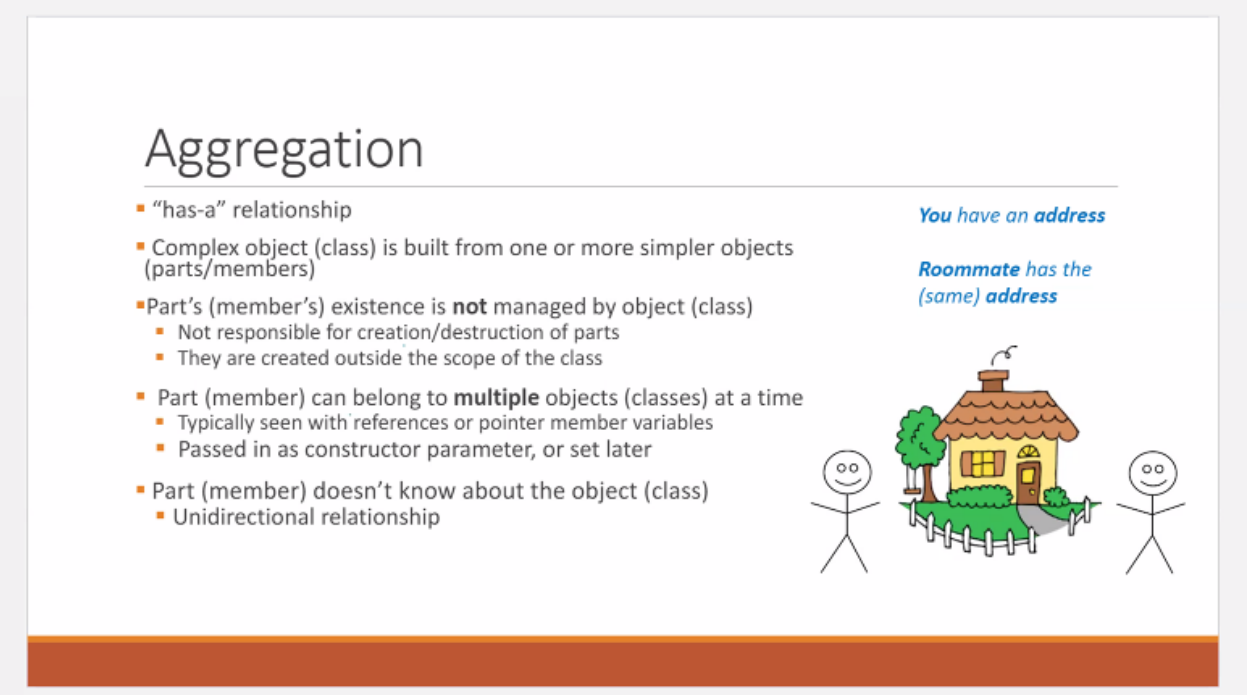
Composition:

- "Part-of" relationship
- Complex object (class) is built from one or more simpler objects
- Part's existence managed by object (class)
    - Responsible for creation/destruction of parts
    - Parts' lifetime is bound to the lifetime of its object
- Part can only belong to one object at a time
    - Excluding static variables of course
- Part doesn't know about the object (class)
    - Unidirectional relationship

So classes are made up of smaller bits that have no clue they are part of a class. Those little bits' life is tied to the life of the object, and the object controls how those bits are created.

Aggregation:

- "has -a" relationship
- Complex object is built from one or more simpler objects
- Parts' existence is **not** managed by object
    - Not responsible for creation/destruction of parts
    - They are created outside the scope of the class
- Part can belong to **multiple** objects (classes) at a time
    - Typically seen with references or pointer member variables
    - Passed in as a constructor parameter or set later.
- Part member doesn't know about the object
    - Unidirectional relationship



Association:

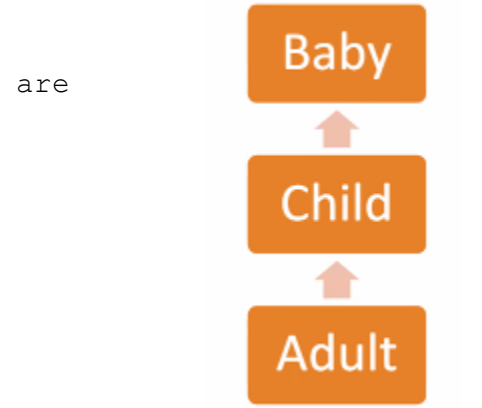
- "Uses-a" relationship (not a part or whole)
- Objects are otherwise unrelated
- Members existence is not managed by class
    - Not creating/destroying parts
- Members can belong to multiple classes
    - Typically use pointer members that point to object outside of scope of aggregate class
- Members may or may not know about the object
    - Unidirectional or bidirectional

Inheritance:

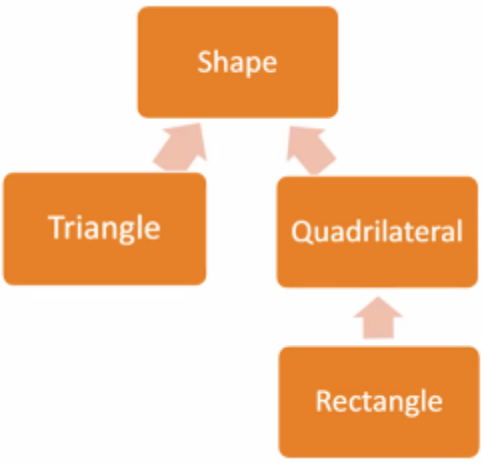
- "Is a" relationship
- Create complex objects by directly inheriting the attributes and behaviours of other objects and then specializing them
- Class being inherited from is known as parent, base, or superclass
- Class inheriting is known as child, derived, or subclass
- Helps avoid reinventing the wheel (duplicate code)

Inheritance:

- Uses hierarchies to show how objects are related
- Two common types:
    - Progression
        - Over time
    - Categorization:
        - From general to specific



One of the most common examples is when we're dealing with geometries and shapes:

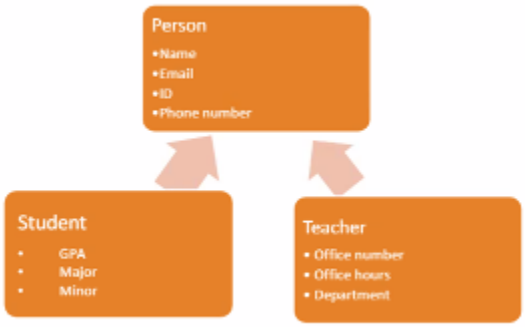


Inheritance

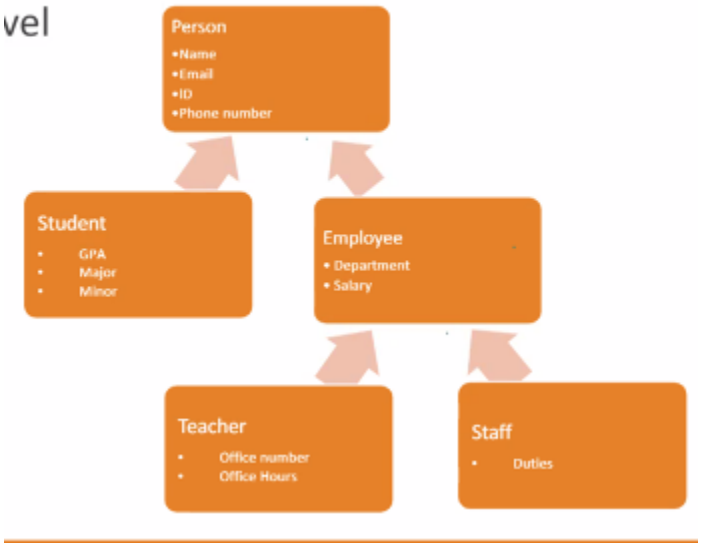
- Suppose that we want to implement two C++ classes with the following member variables

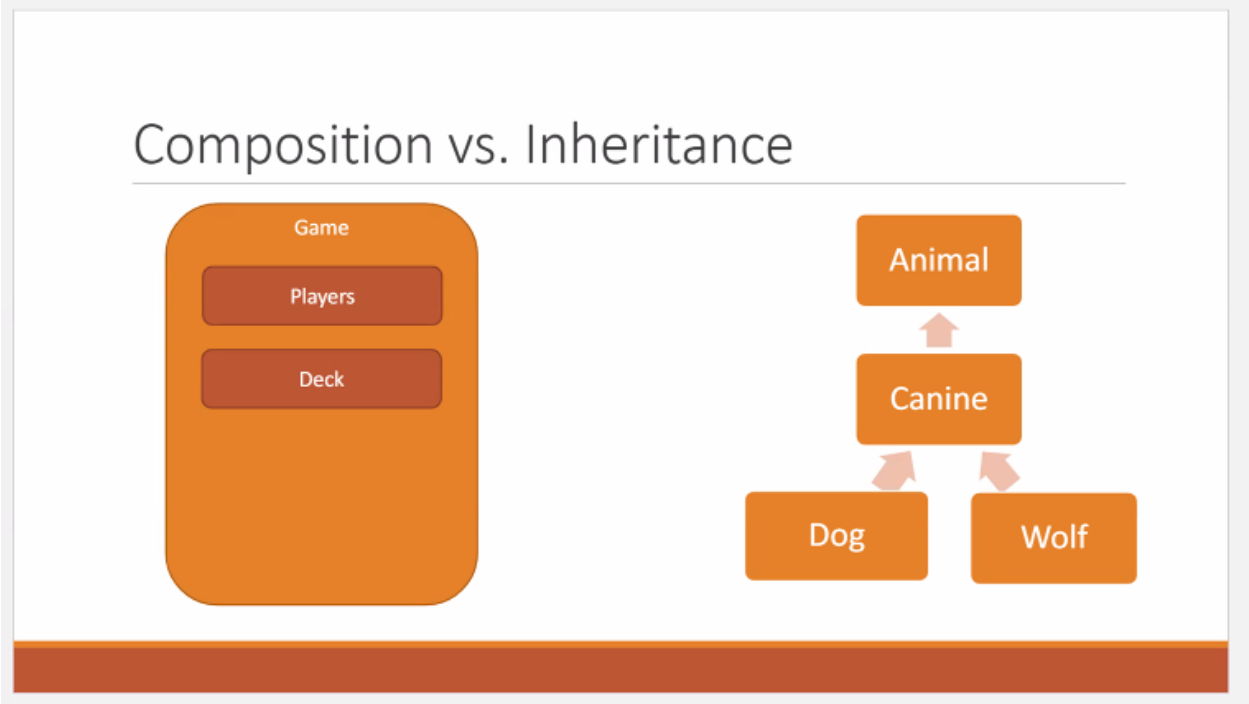
- Student
  - Name
  - ID number
  - Email address
  - Phone number
  - GPA
  - Major
  - Minor
- Teacher
  - Name
  - ID number
  - Email address
  - Phone number
  - Office number
  - Office Hours
  - Department

Students and teachers have a lot of things in common. If we used inheritance for this: instead of having to write all the shared things twice, we could make a person class to hold the command things:



Inheritance is not limited to a single level:



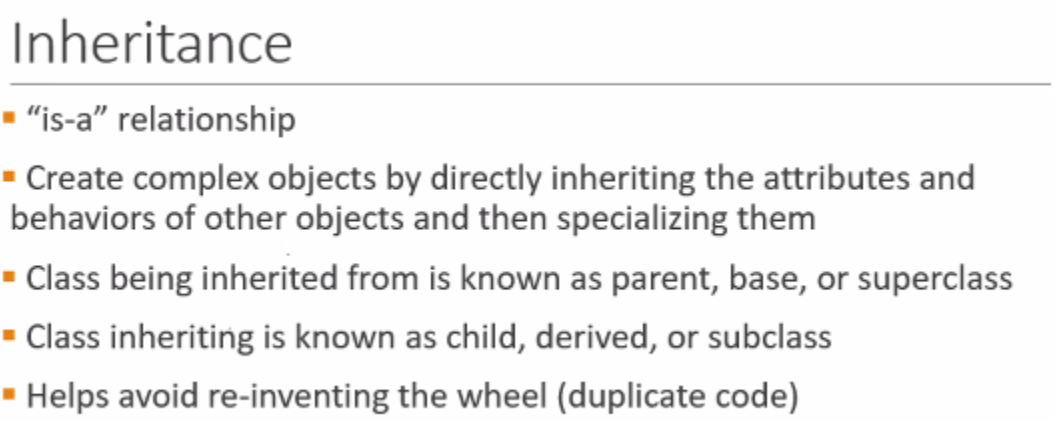


Inheritance is a **big** thing for program 3.

MIDTERM IS DURING CLASS ON WEDNESDAY. WILL BE 50 MIN AND COVER UP TO THE BIG 3. Finishing program 2 will be helpful.

Will be similar to quizzes. 1.2 or 1.3 minutes per question.

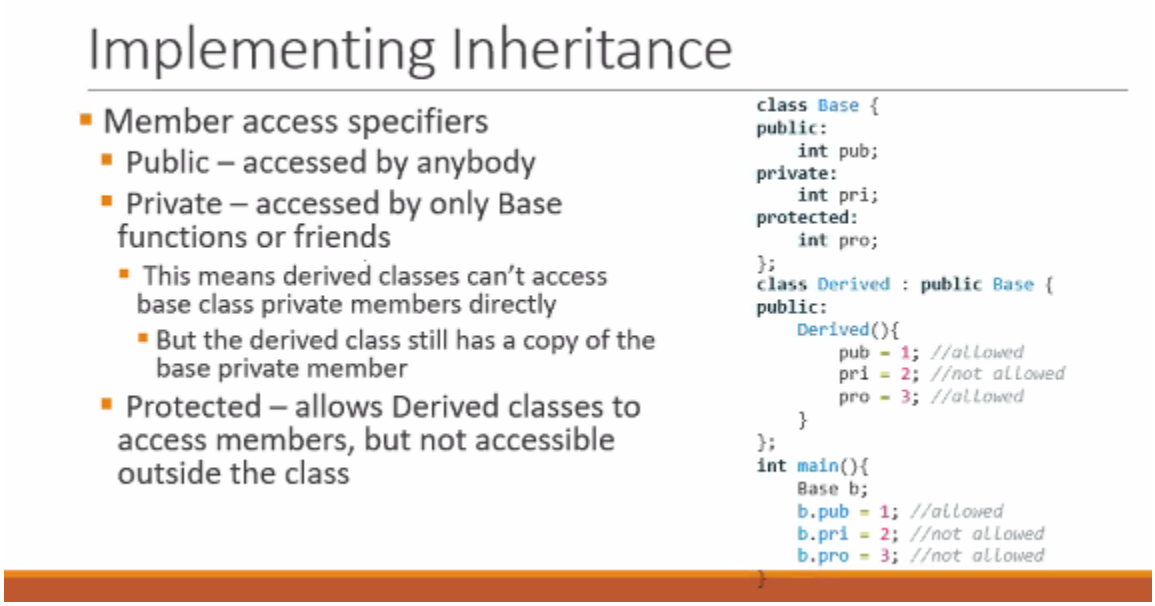
4/23/21 - Lecture 13 - Implementing inheritance




```
class Base {
Public:
      Int pub;
      Int pri;
      Int pro;
};

//I have this class derived, inheriting from Base (both public and
private)

class Derived : public Base{

};

class Derived2 : public Base{

};

Int main(){
      Base b;
      Derived d;

      //derived will have all the public stuff

      b.pub = 0;
      d.pub = 0;
```

```
      Return 0;
}
```

//STATIC VARIABLES NOT SHARED BETWEEN INHERITED CLASSES -- THEY CREATE THEIR OWN STATIC THINGS




```
class Base {
Public:
      Int pub;
Private:
      Int pri
Protected:
      Int pro;
};
```

So derived classes can't access private variables. The protected ones can be accessed inside the class, but cannot be accessed outside of the class stuff.

Basically so:

Private gets passed down but kinda like a const -- the "kids can see it but not change it"
Protected gets passed down kinda like privates but shared among the family
Public is public

If we were to say

```
class Derived : private Base{ //base access
public:
    int better;
```

This means everything now becomes private at the child level, IE you can't access *anything* in the child variable outside of its own scope.

```
class Derived : protected Base{ //base access
public:
    int better;
```

Similar but with protected -- everything becomes protected at child level.

HOWEVER we won't ever use private or protected classes. Just be aware of them, that they exist.

When you call an inherited class, it will call the inherited class's constructor, then the inheriting class's constructor.

You can call a parameterized constructor that calls a base class parameterized constructor:

```
Derived(int x) : Base(x){
    better = 2;
    cout << "Derived constructor!" << endl;
}
```

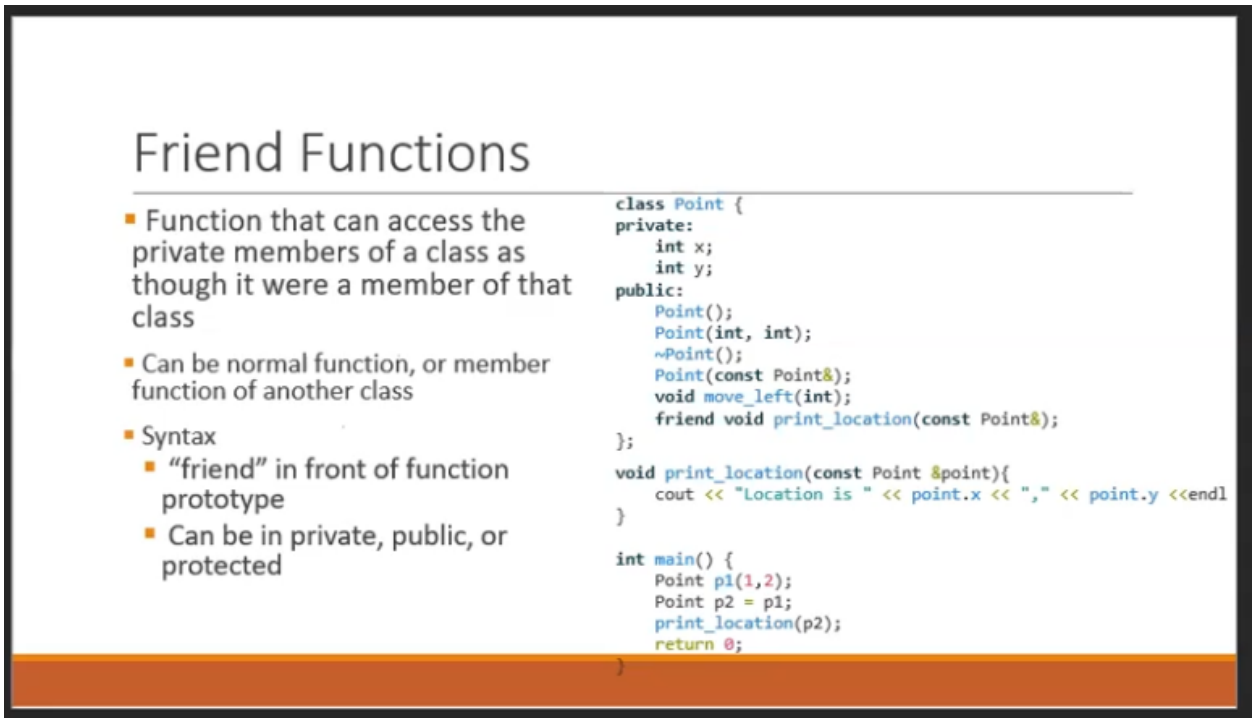
DESTRUCTORS:

So it calls the derived destructor first, then the class destructor. (as part of destroying just the derived destructor).

So children destroyed first, then parents.

4/23/21 - Lecture 14 - Overloading and Friend functions

Friend functions are **not** a member of a class, **but** have access to all the private variables of a class as if they were a member function of the class.



A lot of the time, a friend function will be tied to the specific class -- otherwise it doesn't make sense to not have it as a member of the class.

```
friend void print_location(const Point&); //This is a friend function

//But when writing it:

void print_location(const Point&){
     Cout << location y'know
}

//Notice no scope operator! It's not a member function of the class.
We can still put them into the .cpp files for the class if we have no
better place for them though.
```

You can also make classes friends!!

For example, inside a Point class' public section, you could write friend class Line;

However you need to also include that line in the Line class.

And same thing, gives access to all private members of the other class. No access to implicit this* though.

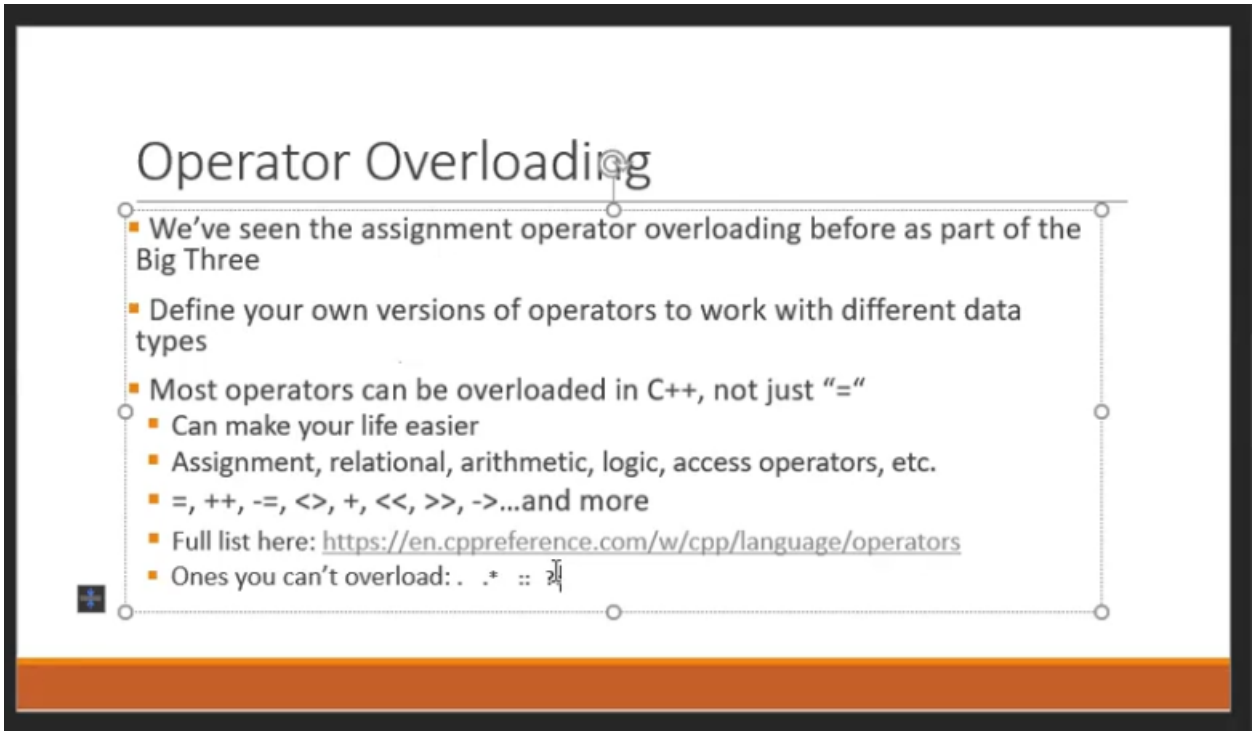
However, friendship is a one-way street. If the Line class is a friend of the point class, this is declared in both classes -- but the Point class is not a friend of the line class.

SO: Avoid friends when possible (in coding, not IRL). This messes with encapsulation.

Uncommon for classes working together.

BUT it is commonly used for operator overloading!

So you can overload most operators, IE, write your own functions for any type of operator for a class.



This way we can do arithmetic operations, comparisons, etc.

When writing a new operator, you cannot make a completely new operator.

Also, your custom operator has to take at least one user-defined datatype.

You can't change the number of datatypes it takes, IE, "+" takes two, can't change that.

You also can't change operator precedence.

TRY TO KEEP OPERATORS CLOSE TO ORIGINAL INTENT. If it's not 100% clear what your defined operator does without any explanation, then just write a new function. IE, keep in the spirit of the new operator.

Writing overloaded functions:

1. Make it a member function of a class.

```
Point& operator+=(int x); //Returns type point, takes type int.
```

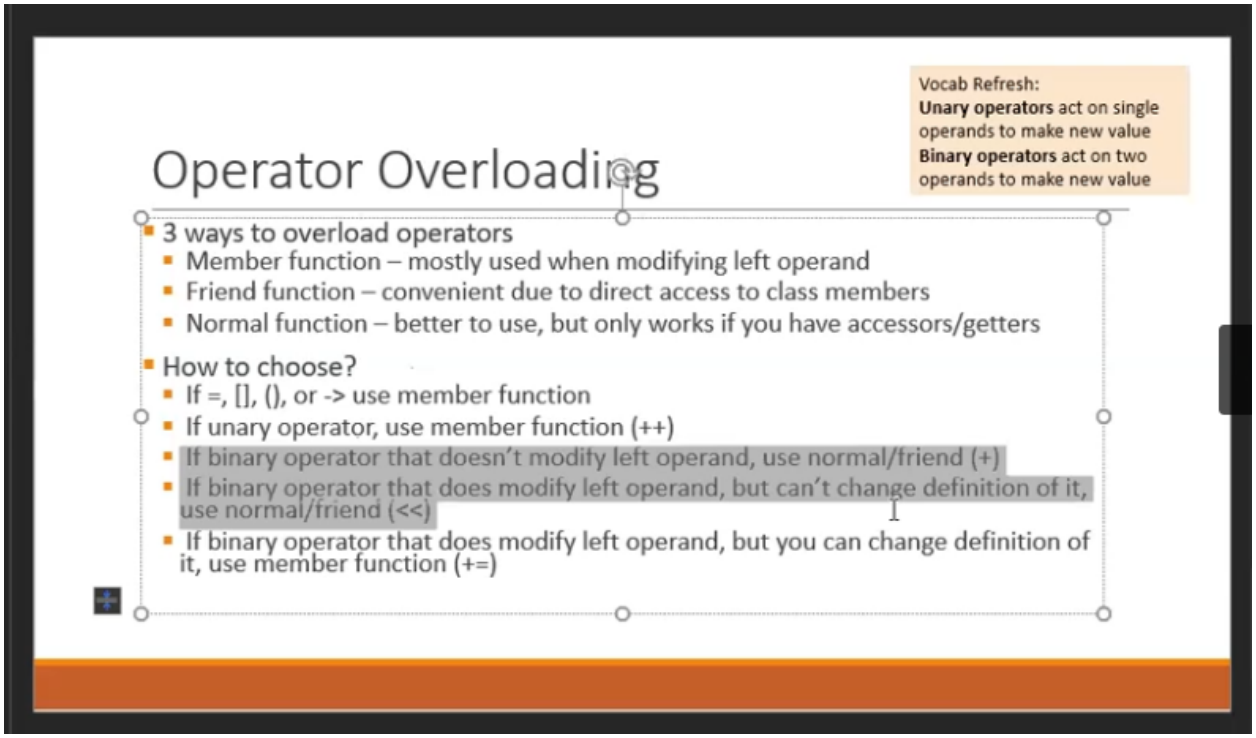
Here's what's in the += operator originally:

```
operator+=(const lh, const rh);
```

So let's write out what this function would do.

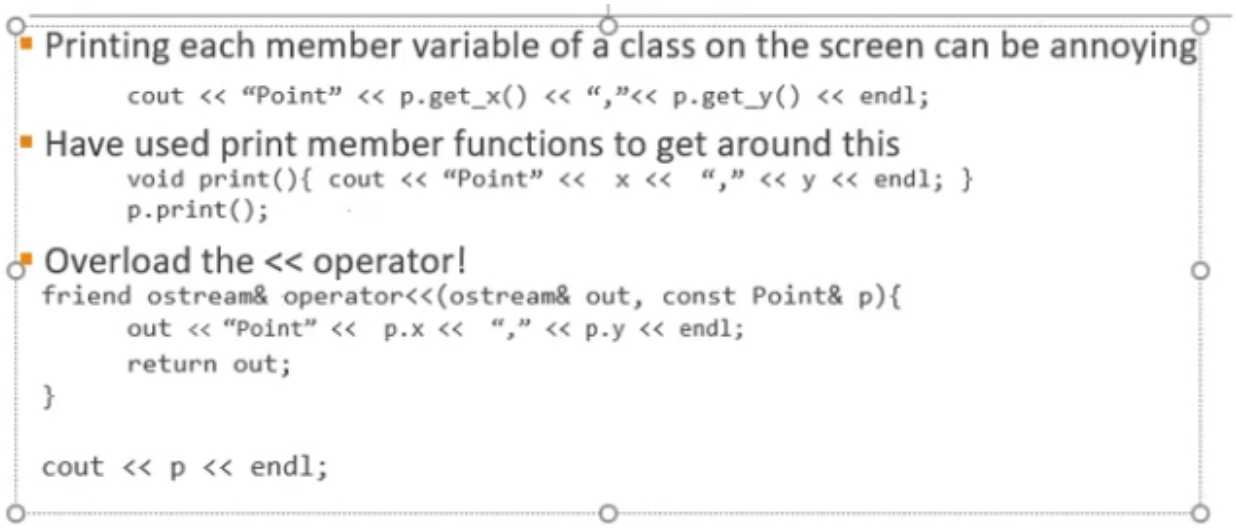
```
Point& Point::operator+=(int x){
     this->x += x;
     this->y += y;
}
```

So yu can essentially create a bunch of shortcuts for yourself.



OPERATOR OVERLOADING WITH I/O:

You can overload the << and >> operators!



The output stream object returns the outputstream object:

```
ostream& operator<<(ostream& out, const Point& p){
     out << "Point " << p.get_x() << p.get_y() << endl;
     return out;
}
```

You'd mostly need to make operator overloads friends when you need access to private variables and want your overload function to not be a part of the class.

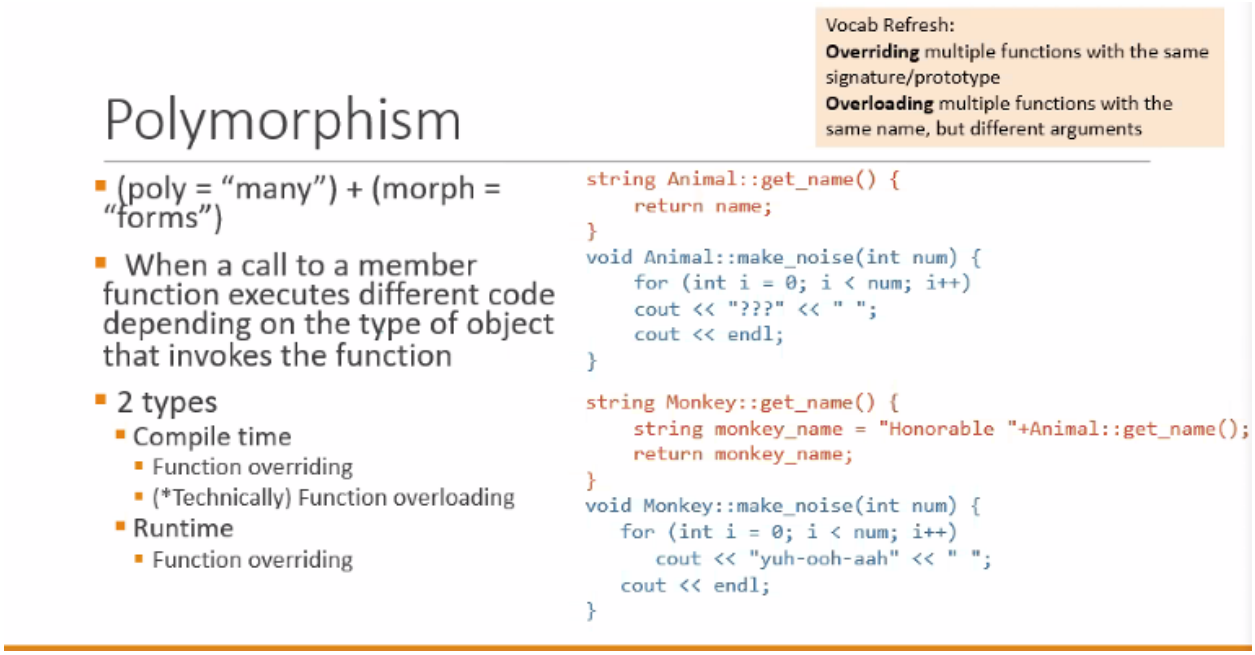
5/3/21 - Lecture 15 -

Magic makefiles now available!
Automagically makes everything into an exe called project.

Doesn’t tar it… So could add a tar command to it.

YOU AN OVERWRITE FUNCTIONS IN DERIVED / INHERITED CLASSES: IE, if you have a get_age() function defined in the parent class, you can have a function with the same name in the subclass that modifies the behaviour.

5/3/21 - Lecture 16 - Polymorphism



Vocab Refresh:
Overriding multiple functions with the same signature/prototype
Overloading multiple functions with the same name, but different arguments

Polymorphism

- (poly = "many") + (morph = "forms")
- When a call to a member function executes different code depending on the type of object that invokes the function
- 2 types
  - Compile time
    - Function overriding
    - (*Technically) Function overloading
  - Runtime
    - Function overriding

```
string Animal::get_name() {
    return name;
}
void Animal::make_noise(int num) {
    for (int i = 0; i < num; i++)
    cout << "???" << " ";
    cout << endl;
}

string Monkey::get_name() {
    string monkey_name = "Honorable "+Animal::get_name();
    return monkey_name;
}
void Monkey::make_noise(int num) {
   for (int i = 0; i < num; i++)
      cout << "yuh-ooh-aah" << " ";
   cout << endl;
}
```

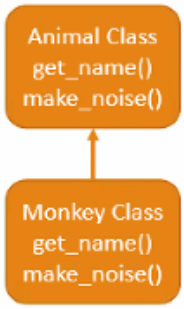
So this is basically how you can override functions for classes that have inheriting classes.

There's compiler polymorphism, and runtime polymorphism. Not sure when runtime is called -- but it means the compiler doesn't decide which to use… that's determined in the program.

So each animal in Assignment 3 might have an overriding function for things like being sick and such.

Pointers, references, and derived classes

- When you create a derived class, it is composed of multiple parts: one part for the inherited class, and a part for itself
- Key feature of inheritance: you can assign a pointer or reference of the base class to a derived class
- Which version of get_name() will run when we call
  - aptr2->get_name()?




```
//using objects
Animal a1;
a1.get_name();
Monkey m1;
m1.get_name();

//using pointers
Animal *aptr = &a1;
aptr->get_name();
Animal *aptr2 = &m1;
aptr2->get_name();
```

It calls the animal-general get_name -- it cannot see the monkey specific stuff. So it will print out whatever name is stored in the parent class.

The use of this is that we can have a general parent pointer that we can pass into functions that look for general animal classes, and then we're actually passing monkey or bear or etc classes in.

Basically, useful cause we can pass in monkeys or bears or tigers or … to a function expecting a generic animal.

```
Animal* array[2]
array[0] = aptr;
array[1] = aprt;

for(int i=0; i<2; i++){
     array[i]->make_noise();
}
```

Pointers

- Pointers allow us to treat an object as a different type without permanently losing data
- Create an array of Base pointers instead of Base objects

```
Animal* array[2];
Monkey m1;
Lion l1;

array[0] = &m1;
array[1] = &l1;

for (int i=0; i<2; i++;){
   array[i]->make_noise();
}
```

NOW WE'RE GETTING INTO RUNTIME POLYMORPHISM (also called Virtual Functions)

Polymorphism

- Virtual functions
  - Special type of function that resolves to the most derived version of the function that exists between the base and derived class
  - Keyword "virtual" indicates the compiler should wait until run-time to determine which version of the function should run
    - Isn't explicitly needed for derived functions, but doesn't hurt
  - Can be overridden if it is re-defined in a child class

```
//in animal.h
virtual string get_name();
virtual void make_noise(int);
```

```
//in monkey.h
string get_name();
void make_noise(int);
```

Useful in relation to the previous example somehow...

Abstract Polymorphism

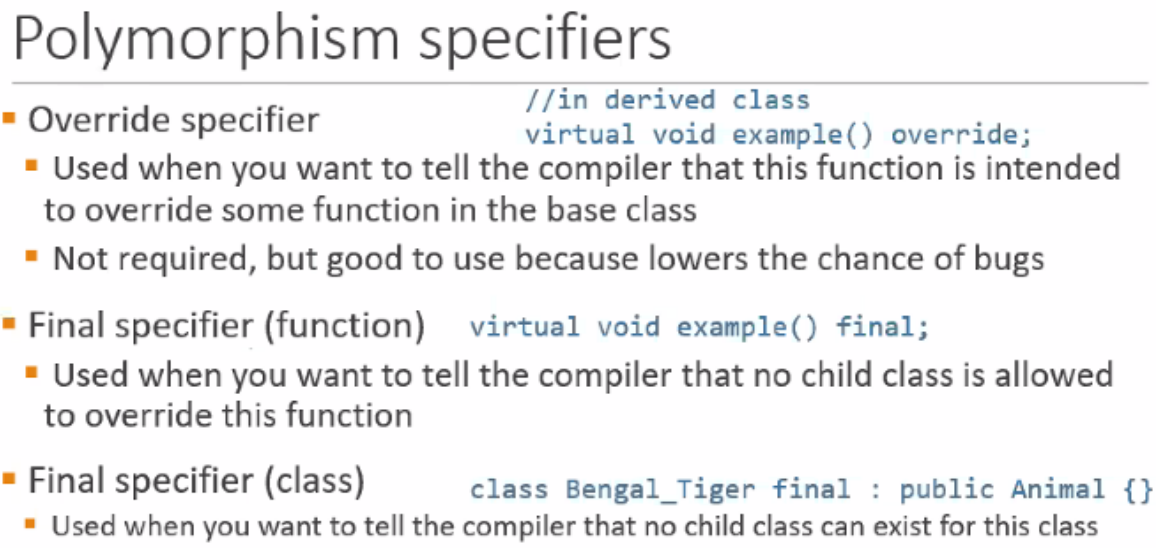
- Abstract function `virtual void example() = 0;`
  - Also known as pure virtual function
  - A virtual function that has no definition in the base class
  - Used when you are intending for child classes to implement the function
- Abstract class
  - Any class that has one or more pure virtual functions
  - An abstract class cannot be instantiated (i.e. you cannot create an object out of an abstract class)
    - But you can use pointers & references to abstract class types

So this is useful when every single child will need to have a unique version of a function, and we don't want to write that function for the base class.

Like the make_noise function, which isn't useful for the base animal class…

Adding this makes it so EVERY child in the class HAS to override/implement this function. Otherwise the compiler will be annoyed and tell you that you're missing overriding functions.

"It goes down as far as it can to find the best thing." < On grandchildren and etc.



Just like we had const, there are specifiers for polymorphism.

(You add them to function declarations.)

They help control classes and how things get "polymorphed."

The first:

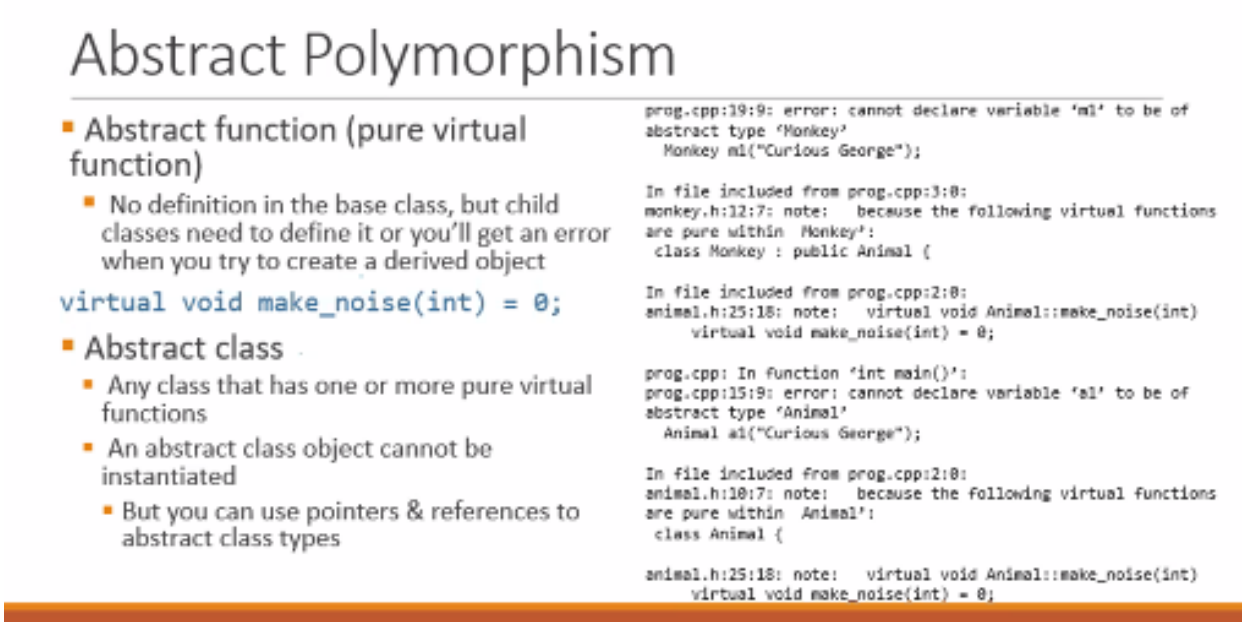
- **Override specifier:**
  - Tells the compiler that this function is intended to override the function in the base class
  - Not required but good to specify to other programmers, and reduced bugs. So this should be added to my Prog 3 stuff
  - Void make_noise(int) override;
- **Final specifier:**
  - Used when you want to tell the compiler that a function in a parent (or derived!) class is **not to be overridden.** IE, it will stop looking for overriding functions there / give errors if you do try to override.
  - Can be applied to **both functions and classes!**
  - **Void make_noise(int) final; //Same syntax as override**

5/7/21 - Lecture 17 - Polymorphism continued

Specifiers from last time:

- Override
- Final

Also, remember virtual: this means the function doesn't get defined until later down when a class inherits it. IE, every child class MUST define it.



What happens when we don't override the pure virtual function.

Interface Classes

- A class with NO member variables, and where ALL of the functions are pure virtual/abstract
- Useful when you want to define functionality derived classes must implement, but leave the details up to the derived class
- No keyword in C++, just how you set it up

```
class Shape {
public:
  virtual ~Shape();
  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual float area() = 0
  //...
};

class Rectangle : public Shape {
public:
  virtual ~Rectangle();
  void move_x(distance x);
  void move_y(distance y);
  float area();
private:
  float width;
  float height;
  //...
};
```

Interface classes have no member variables -- just public virtual functions only. There's no keyword to say that's what it is.

One useful example -- if you wanted to make a user interface, you could use an interface class to have basic stuff like errors, clicking on a button, sidebar, etc.

Inheritance & Polymorphism

- Things that are not *truly* inherited
  - Constructors
  - Destructors
  - friends
- Whenever you are dealing with inheritance, make any explicit destructors virtual
  - Memory leaks if you delete a base class pointer pointing to a derived object
  - virtual ~Base(); virtual ~Derived();

Some arguments about this -- but for the purpose of this class we will say they are not inherited. (Hint hint for exams).

Make destructors virtual when you have explicit destructors -- cause otherwise it won't know to go down the line to destruct the child classes.

Vocabulary:

- Binding

The process of converting variables and stuff into actual memory values and places. IE, "this animal.get_name() function is stored [here] in memory."

- Early/Static binding

Early/static means everything gets bound during compilation.

- Late/Dynamic binding

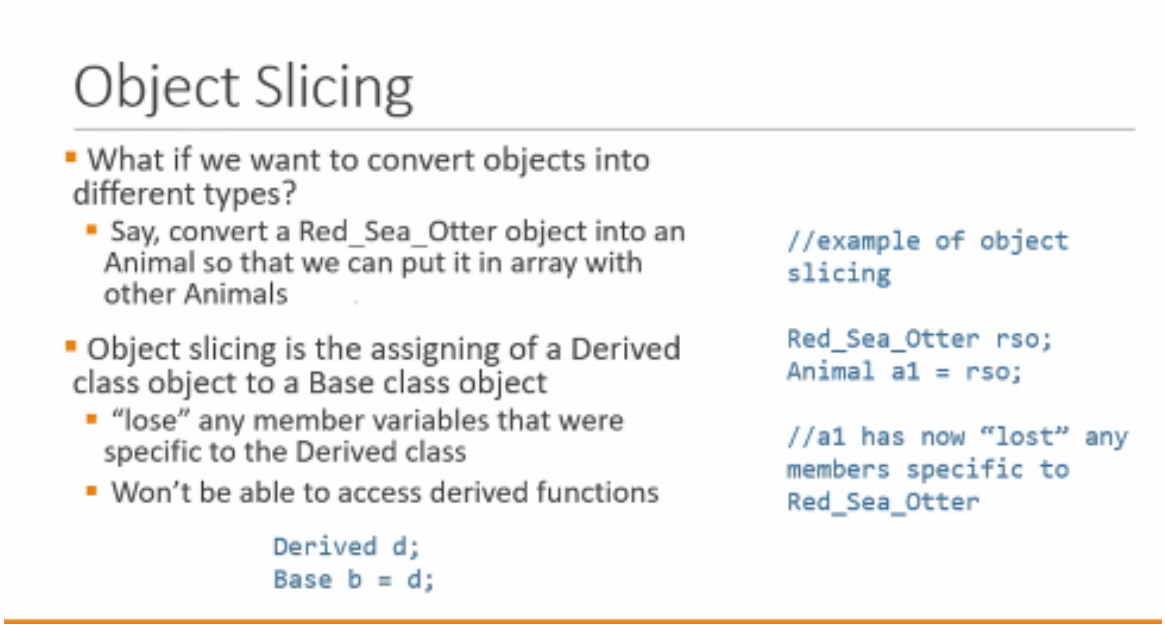
Polymorphism (runtime) is deciding which function to use during runtime -- so it cannot bind when compiling.

That's called late/dynamic binding.

DOWNSIDES OF POLYMORPHISM:

Late binding comes at a cost. Because it has to wait till runtime to make those decisions, it's less efficient, could take more time. For most uses, this really doesn't matter. But IE for microcontroller programs on tight time and memory, you wouldn't use polymorphism much probably.

I NEED TO LOOK MORE INTO PARENT POINTERS. HOW USEFUL? CAN WE USE THEM WITH ASSIGNMENT 3?



Slicing objects makes them lose child class information! So we'd basically never do this!

So, don't directly assign children to parent objects.

```
Red_Sea_Otter rso("Pop"); //Creates red sea otter
Animal a1 = rso; //DON'T DO!
```

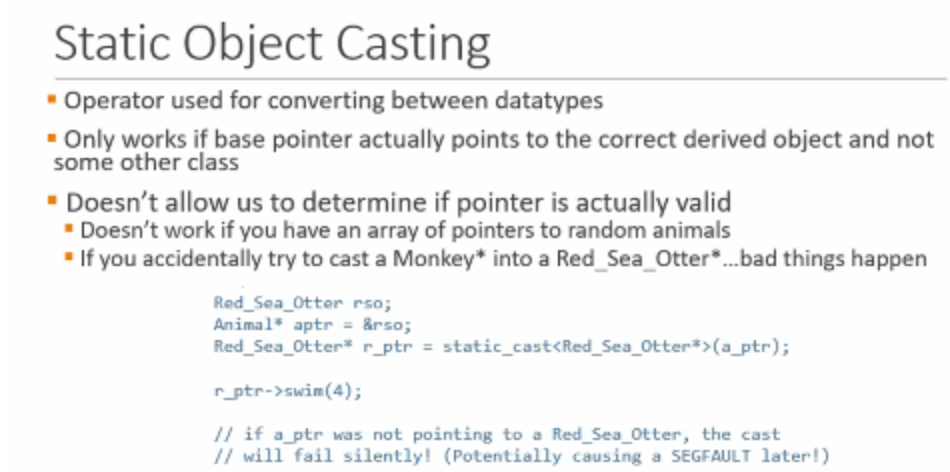
But, we can use pointers, like so:

```
Animal *aptr2 = &rso;
```

Then we can use the Animal class functions with aptr. I think this makes the RSO functions inaccessible, BUT they still exist and the data remains.

```
Red_Sea_Otter *r_ptr = aptr2; //But we can't do that. We can't convert an animal pointer to a red sea otter pointer. How do we get back to red sea otter?
```

CASTING:




```
Red_Sea_Otter *r_ptr = static_cast<Red_Sea_Otter*>(aptr2);

//This WON'T work if you're not casting it to a pointer of the same type. It will just silently fail. Then cause a seg fault or something

//Only use static casting if you're 100% certain you can do it. The parent MUST be pointing back to the correct child type.

Why do we need to know about it then? Well, it exists, and sometimes it's useful.
```

DYNAMIC CASTING:

```
Red_Sea_Otter *r_ptr = dynamic_cast<Red_Sea_Otter*>(aptr2);

But this will not fail silently. So, instead of doing a segfault, it will give us a nullptr instead. So we can do checks:
```

```
if(r_ptr!=nullptr)
    r_ptr->swim(4);
else
    cout << "Wrong type of dynamic casting! "
```

Dynamic Casting

- Operator used for converting base pointers to derived pointers
- Useful when you don't know what the type of the object is
  - You have an animal pointer, but don't know what type of animal
- Cast is verified at runtime
  - If the cast was successful, you get a valid pointer
  - If unsuccessful, you get a nullptr
- Also works with references

```
Red_Sea_Otter rso;
Red_Sea_Otter* tmp;
Animal* aptr = &rso;

tmp = dynamic_cast<Red_Sea_Otter*>(aptr);
if (tmp != nullptr){
    tmp->swim(12);
} else {
    cout << "Red_Sea_Otter cast error" << endl;
}
```

5/7/21 - Lecture 18 - Exceptions

Program 4: Called Hunt the Wumpus
Considered a classic text-based horror-survival game.

Intro on Wednesday.

You are required to use vectors on 4. You'll learn about those next week I guess. Maybe a good idea to read ahead on this / read the slides ahead of time?

Error handling:
It's a lot of our lives as programmers. At a certain point we should *always* implement error handling. IE, we want our code to be rock-solid, no user errors.

Syntax errors -- easy to catch, like missing semicolons.

Semantic errors: When the program compiles, but it doesn't work like you want it to.

- Logic errors -- code logic is incorrect
- Violated assumptions -- programmer assumed something wrong

Defensive programming:

- Design program to identify where assumptions might be violated and write code to detect and handle that appropriately
- Detecting errors
    - Check that parameters are correct before each function
    - Check the return value / error reporting mechanisms after a function finishes
    - Check user input to make sure it matches what we want them to input

How to best defensive programming?

- No best way!
- Some methods
    - Skip code that depends on assumption being valid
    - If in function, return error code back to caller and let caller deal with it (IE, when I returned 314 to let my program decide to do something else)
    - Use **exit()** function to terminate program (CAN LEAD TO MEM LEAKS)

- If user enters invalid inputs, ask for input again
- Use **cerr** output stream to write error messages on the screen or to a file
- Use an **assert** statement

Assert statement:

Assert

- Evaluates a conditional (true or false) at runtime
  - If true, does nothing
  - If false, displays an error message and program is terminated
- #include <cassert>
- Often used for checking parameters are valid, and if return values are valid
- Error message contains the conditional that failed, along with the name of the file and the line number

```
#include <cassert>

int getValue(const array<int, 10> &a, int i)
{
    assert(i >= 0 && i <= 9);
    return a[i];
}

//call function with invalid index
int value = getValue(array, -3);
```

```
Assertion failed: i >= 0 && i <= 9, file C:\\prog.cpp, line 10
```

If something is false, it will exit the program and print an error message and show what line it occurred on.

```
#include <cassert>

assert(i >= 0 && i <= 9 && "Invalid range");
```

If I doesn't meet those, then it will print:

"Assertion 'i >= 0 && i <= 9 && "Invalid range"' failed."

EXCEPTIONS:

- Not your typical error handling…
- Why do need them?
  - Return codes can be cryptic
  - Functions only return one value -- can't return results AND errors
  - Some functions can't return things
  - Functions callers may not be equipped to handle error codes

- Exception handling lets use decouple error handling from control flow of the code
    - Meant for dealing with edge cases and stuff

Exception terms:

- Try
- Throw
- Catch

- Looking for exceptions: **try**
    - Try blocks look for any exceptions that are thrown by statements
    - Doesn't do anything, but actively looks for exceptions
- Throwing exceptions: **throw**
    - Throw statements signal an exception has happened and needs to be handled
    - Happens when a try finds an exception
- Handling an exception: **catch**
    - Catches the thrown exceptions
    - Handles the actual exceptions, catches from throw which was thrown by try.

Inside main:

```
array<int, 10> test;
try{
    int value = getValue(test, 3);
    throw -1;
}catch(int){
    cerr << "Caught int exception" << endl;
}
```

Catch must be on the end of throw as shown, like an if/else block.

The type it's catching (the catch(int) line) depends on what we're trying to catch.

```
array<int, 10> test;
try{
    int value = getValue(test, 3);
    throw -1;
    throw 5;
}catch(int x){
    cerr << "Caught int exception " << x << endl;
}
```

So the throw statement (the FIRST throw statement) will immediately call the catch block.

It goes from top to bottom.

So prioritize which you want handled.

Exceptions in Functions

- Throw statements do not have to placed directly inside a try block
- Usually throw in one function, and catch in another
  - Makes code more modular, as the caller function can handle the exception in different ways
  - Useful in member functions
    - Operators, constructors, etc.
    - Doesn't require a change to function signature

```
double Sea_Otter::predict_swim_time(int distance)
{
    if (swim_speed == 0) {
        throw swim_speed;
    }
    return distance/swim_speed;
}
```

```
try {
    int d = 30;
    cout << tso1.predict_swim_time(d) << endl;
    cout << "Everything was executed!" << endl;
} catch (double e_val) {
    cout << "Double exception " << e_val << endl;
}
```

```
if(!(i>=0 && i<=9)){
    throw i;
}
```

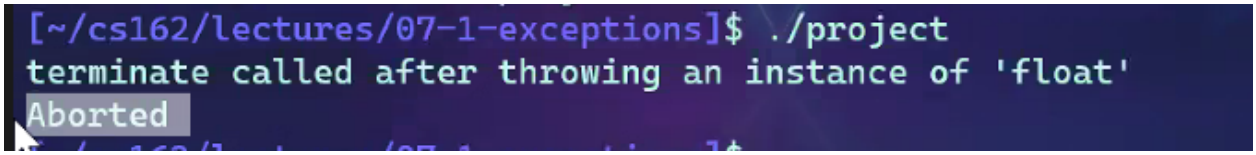
So this will throw an exception, but not affect the return value.

Oh we can have multiple exception types, like so:

```
}catch(int x){

}catch(double x){

}catch(etc){}
```

Uncaught exceptions:

If an exception gets thrown and you don't catch it, something will happen like so:



To handle that, we use a "catch all handler" shown below:

```
}catch(...){
      cout << "Caught unknown exception!"
}
```

There is a keyword for functions for dealing with exceptions…

"noexcept" << This means the function will not throw an exception.

Basically just put it next to something when you say it will not throw an exception.

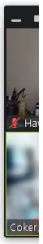
You should only put try blocks around things that are *expected* to throw an exception. You don't want unexpected exceptions.

5/7/21 - Lecture 19 - More exceptions

Exception Classes

- Basic data types as exception types are vague
- Use exception classes to handle exceptions from various sources differently
  - A normal class designed specifically to be thrown as an exception
- Exception handlers should catch exception class objects by reference
- Inheritance
  - Exception handlers will match classes of a specific type, but also classes derived from that type

```
class Base{
public:
    Base() {}
};

class Derived : public Base{
public:
    Derived() {}
};

int main() {
    try {
        throw Derived();
    } catch (Derived &derived) {
        cout << "caught Derived";
    } catch (Base &base) {
        cout << "caught Base";
    }
    return 0;
}
```



'Cause I wasn't paying enough attention to type this out right away...

Std::exception

- Interface class designed to be a base class to any exception thrown by C++ standard library
- what()
  - Virtual member function, returns c-style string descriptions of exception
  - Derived classes override it to change message appropriately
  - Include noexcept specifier in C++11

```cpp
#include <iostream>
#include <exception> // for std::exception
#include <string> // for this example

int main()
{
    try
    {
        // Your code using standard library goes here
        // We'll trigger one of the exceptions
        std::string s;
        s.resize(-1); // will trigger a std::bad_alloc
    } catch (std::exception &exception) {
        // This will catch exception and all derived exceptions too
        cerr << "Standard exception: " << exception.what() << endl;
    }

    return 0;
}
```

```cpp
class Swim_Exception : public exception {
private:
    string name;
    string description;
public:
    Swim_Exception(string, string);
    const char* what() const noexcept;
};
#endif
```

```cpp
#include <iostream>
#include "swim_exception.h"

using namespace std;

// define a constructor that sets the following variables:
// name - name of the swimming animal
// description - explanation of the exception
Swim_Exception::Swim_Exception(string name, string tmp) : name(name), description(tmp) {
}

const char* Swim_Exception::what() const noexcept {
    string error_message = name + " encountered a critical exception: ";
    error_message += description;
    return error_message.c_str();
}
```

So you can make your own exceptions using the exception class. Each one has to redefine the what function (since the what function is a pure virtual function).

So then you can call it, (**AND THE WAY YOU CALL IT IS BY throwING the constructor! IE:)**
throw Swim_Exception(“”,””); (I didn’t get a screenshot of the actual syntax…)

WARNING! DERIVED CATCHES MUST BE BEFORE BASE CATCHES!

```cpp
class Base {};
class Derived: public Base {};
int main() {
    Derived d;
    try {
        throw d;
    } catch(Base b) {
        cout<<"Caught Base Exception";
    } catch(Derived d) {
        cout<<"Caught Derived Exception";
    }
    return 0;
}
```

So here it will print the base exception first because it sees some base class in the derived class. Put derived ones first!

Introduction to Hunt the Wumpus

- Text based adventure made in 1973
- Go through series of caves and hazards to find and kill the Wumpus and escape with the gold
- Use polymorphism and template class
- Two modes:
    - Normal
    - Debug **(Build debug first!!)**

Normal:

- **Player can't see, can't see what they have, etc**

Debug:

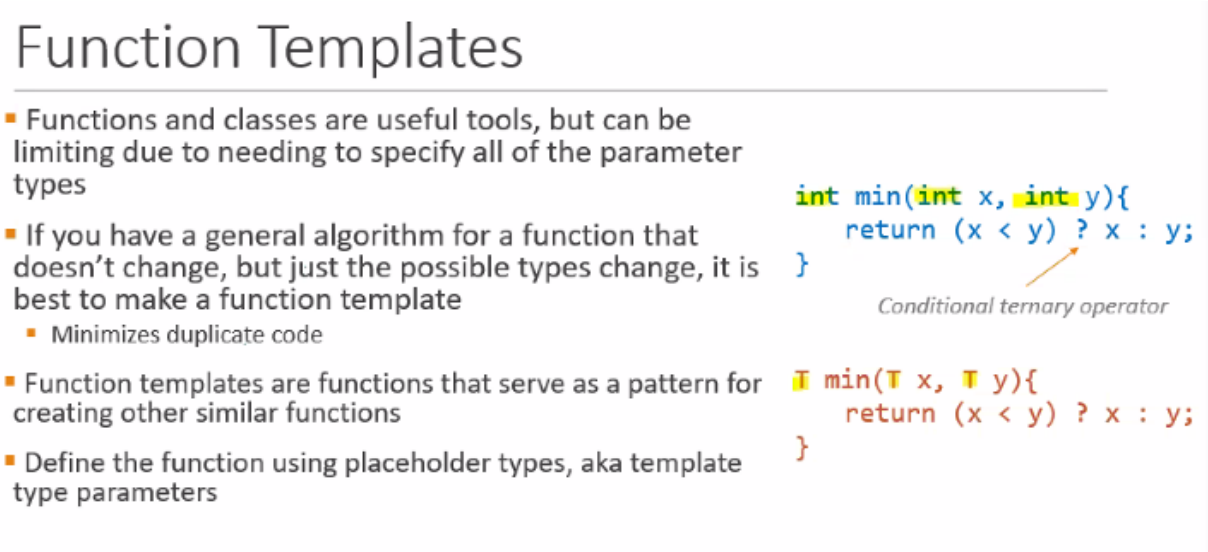
- **Write this first**
- **You can see everything**
- **You can see items and such**
- **Etc. Anything else that is helpful.**

Here's an online version of hunt the wumpus:
https://osric.com/wumpus/

Tertiary Operator:

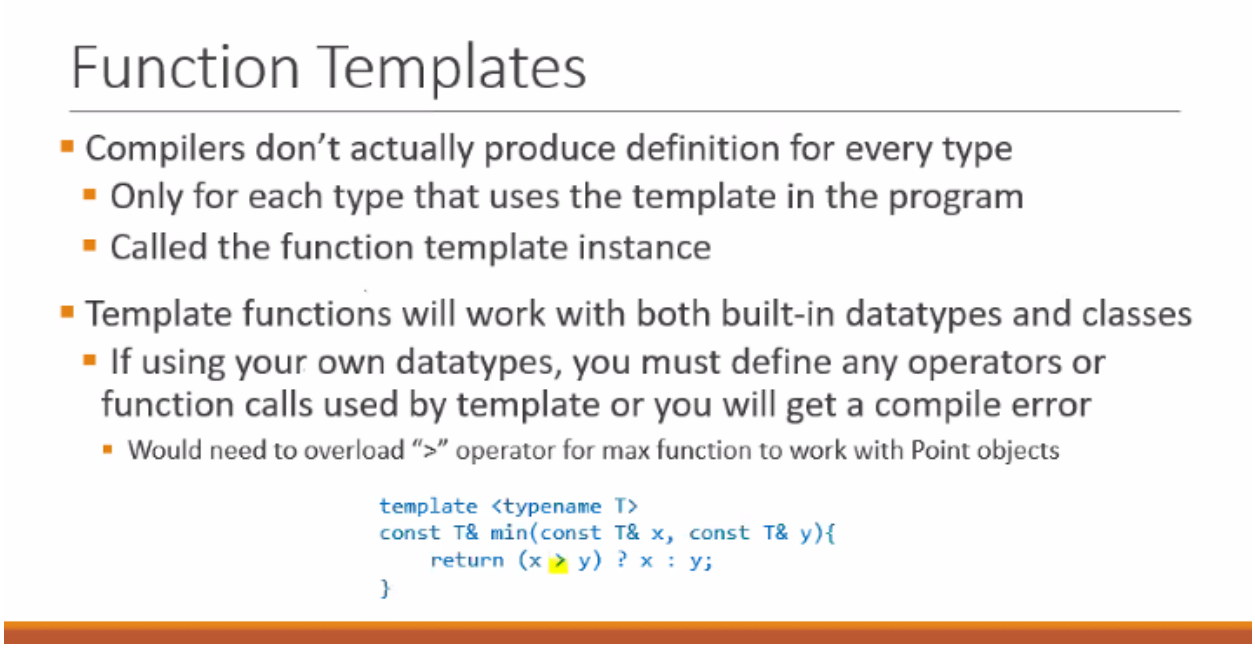
```
Return (x < y) ? x : y;
"Evaluate what's in the (), if it's true, return x, else return y"
```

Function templates:



If you have a user-defined type and want to use a template function, you'd better make sure any operators you use have been overloaded for your user-defined datatype.

But will work by default for user-defined functions



So the compiler basically just turns the basic template function into as many as it needs specific to the datatypes the program uses.

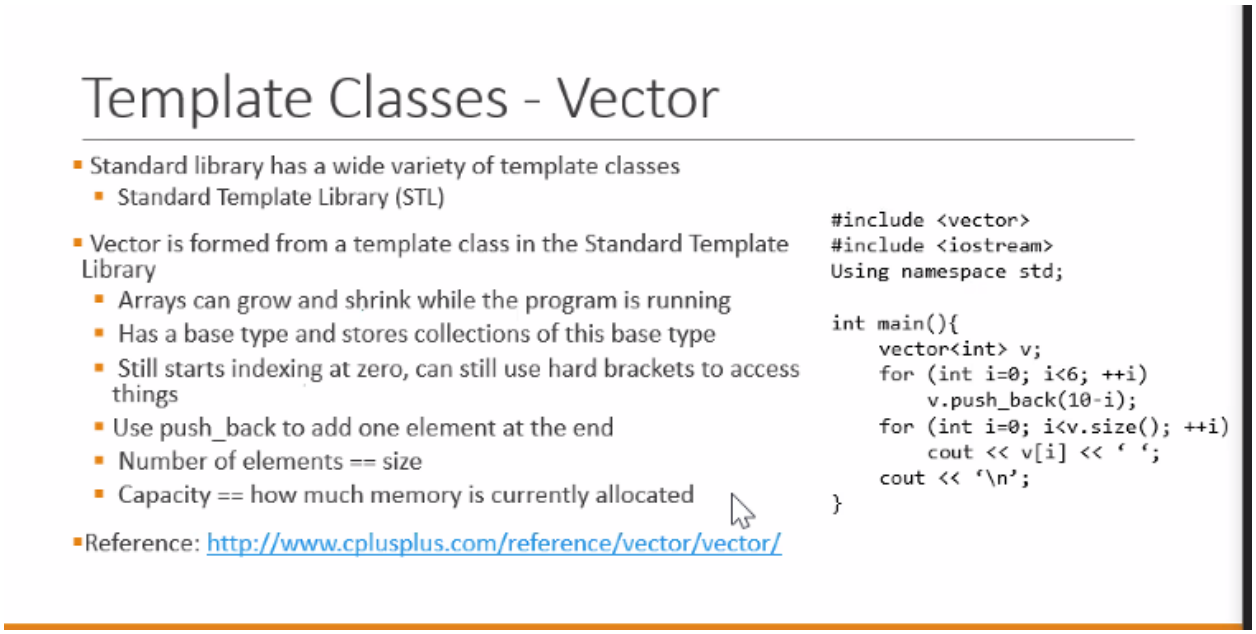
Pros:

- **Reduce code maintenance**
- **Can be safer**

Cons:

- **May not work perfectly with older compilers**
- **Error messages are harder to read**
- **Increase compile time and code size**

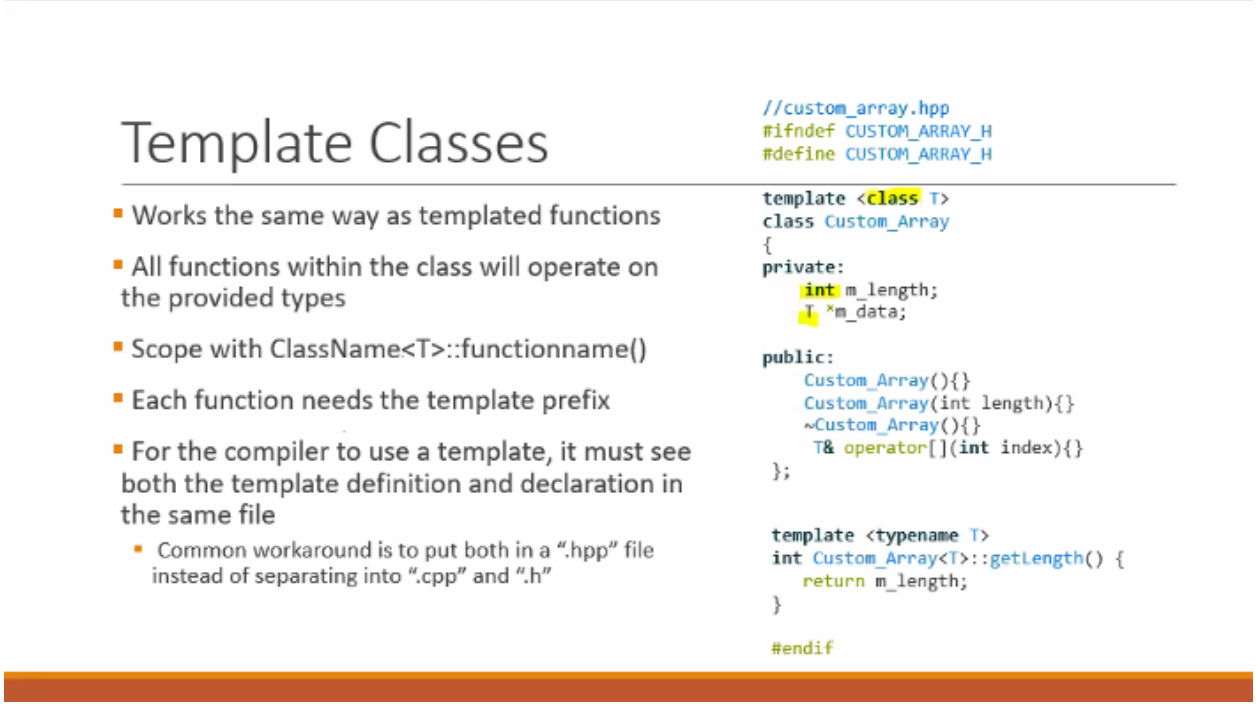
Due to compiler inconsistency, you need to define the template in the same file it is invoked.



Preview! Much much easier than dynamic memory, take preview if you want to start on assignment 4.

5/14/21 - Lecture 20 - Vectors & Template classes!

Function template **instances** -- these are what the specific examples of template functions that the compiler generates are called




```
template <class T> //This is maybe unneeded but should be put anyways
Class Custom_Array{
      //blah blah normal class definitions
      private:
            int m_length;
            T *m_data;

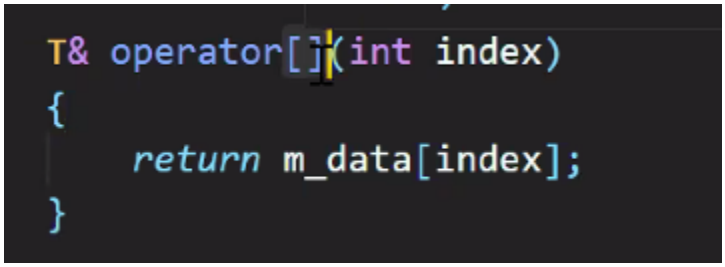
      public:
            Custom_Array(){
                  m_length = 0;
                  m_data = nullptr;
            }

            Custom_Array(int length){
                  m_data = new T[length];
                  m_length = length;
            }
            ~Custom_Array(){
                  delete[] m_data;
```

```
        }

        void Erase(){
            delete[] m_data;
            m_data = nullptr;
            m_length = 0;
        }

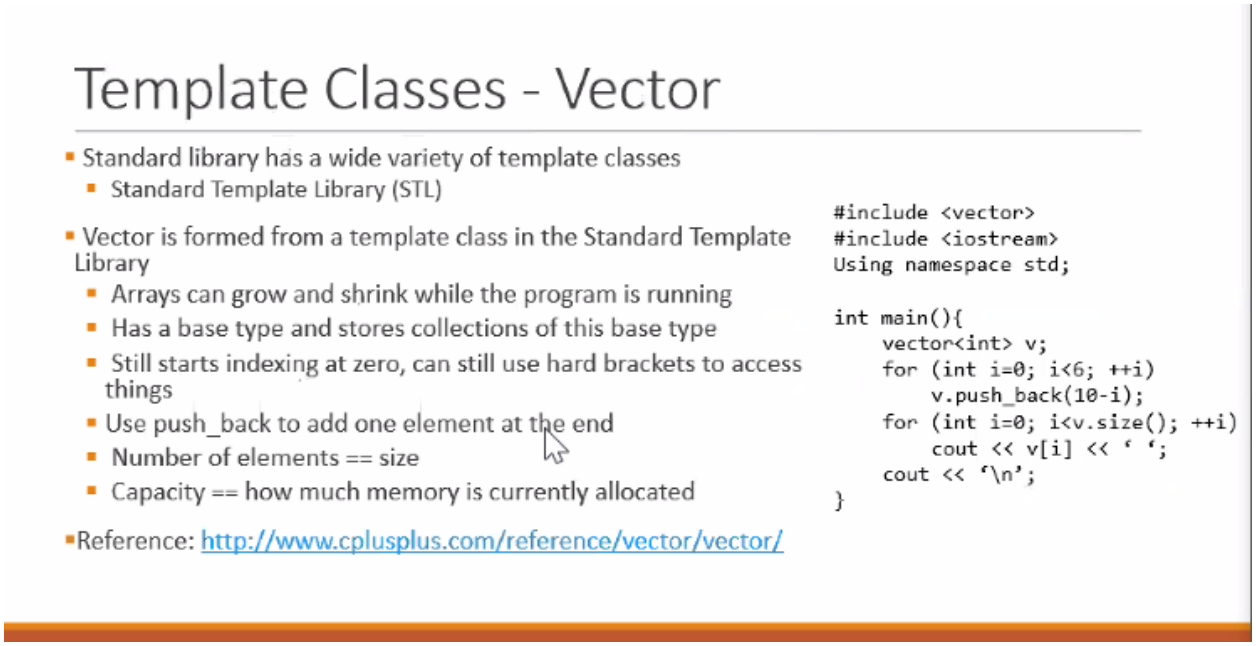

};
```



You can overload bracket operators -- it must accept an int, and then return a value at some index.

Don't do file separation for template functions! Put them all in one file. The file extension should also be .hpp.

That's just another .h file extension. .h and .c are from C, .hpp and .cpp are from c++



VECTORS:

```
vector<int> v; //vector of type v. No size allocated, it has a
nullptr happening, its size is 0.
```

Looking at the reference documentation will be helpful at this point! If there's something we want, it probably exists already.

```
push_back // adds an element to the end
pop_back // removes the last element

vector<int> v;
for(int i = 0; i < 6; i++){
      v.push_back(10-i);
}

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

cout << "\n"; //Fun fact about \n vs endl -- the \n works in both c++
and c, but what it doesn't do is flush the buffer that things get
stored in. endl does flush. flushing takes time, so using \n may be
faster.

endl is better for print statement debugging.
```

There's a difference between the size function and the capacity function.

vector<int> v2(10, 2);//creates a vector of size 10, each value begins at 2.

```
vector<int> v2(10, 2);
cout << "Size :" << v2.size() << endl;
cout << "Capacity :" << v2.capacity() << endl;

v2.push_back(7);
cout << "Size: " << v2.size() << endl;
cout << "Capacity: " << v2.capacity() << endl;
```

```
minimum(3, 9) is:
3
minimum(3.14, 2.718) is:
2.718
Foo value at 12: 24
10 9 8 7 6 5
Size :10
Capacity :10
Size: 11
Capacity: 20
```

One thing the vector class does to try to be more efficient is to add more memory than it thinks it needs, so that it doesn't need to find a new chunk of memory every single time you added something.

For program 4, there are a lot of ways to handle making the grid. commonest is making a 2D vector.

//For 2D vectors. This will create a vector of vectors of type room.
vector<vector<Room>> cave(4, vector<Room>(4)); // 4, x 4

Creates a 4x4 grid of rooms.

```
vector<vector<Room>> cave(4, vector<Room>(4));

for(int i=0; i < square_size; i++){
    for (int j=0; j < square_size; j++){
        cout << cave[i][j].get_name() << endl;
    }
}
```

Here's examples of accessing stuff.

```
cave[2][2].set_name("Death");

cout << cave[2][2].get_name() << endl;
```

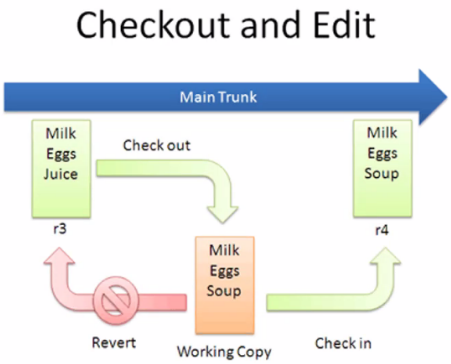
5/17/21 - Lecture 21 -

Week 7 quiz do Wednesday night!

The AI for Hunt the Wumpus does not need to be smart at all. So literally a random one.

Version Control Systems (VCS)

- Let you manage and record changes to a file (or set of files) over time
  - Can recall specific version later – keeps a history of all past versions
  - Commonly used for multiple people collaborating on the same file
- Subversion (SVN)
  - Easier to learn
  - Centralized VCS
  - TortoiseSVN is most popular free Windows SVN client (SmartSVN for Mac)
- Git
  - Most popular for open source projects
  - Distributed VCS
  - GitHub is popular tool for managing your code online



Version control systems? **Software tools that intelligently let you work on code with others, save multiple versions as you work to avoid random stuff.**

Git is a popular one, GitHub is probably based on it. Subversion is another one. Good to know for working on a project with multiple people.

The Standard Library

- Programs use the same sort of concepts over and over
  - Loops
  - Strings
  - Arrays
  - Sorting
- Standard Library is a collection of classes that provide templated containers, algorithms, and iterators
  - Pro: Templates for common programming needs
  - Con: Can look complex due to everything being templates
- Refer to documentation before a project to see if you can use something from the STL
  - http://www.cplusplus.com/reference/stl/

La

Container classes

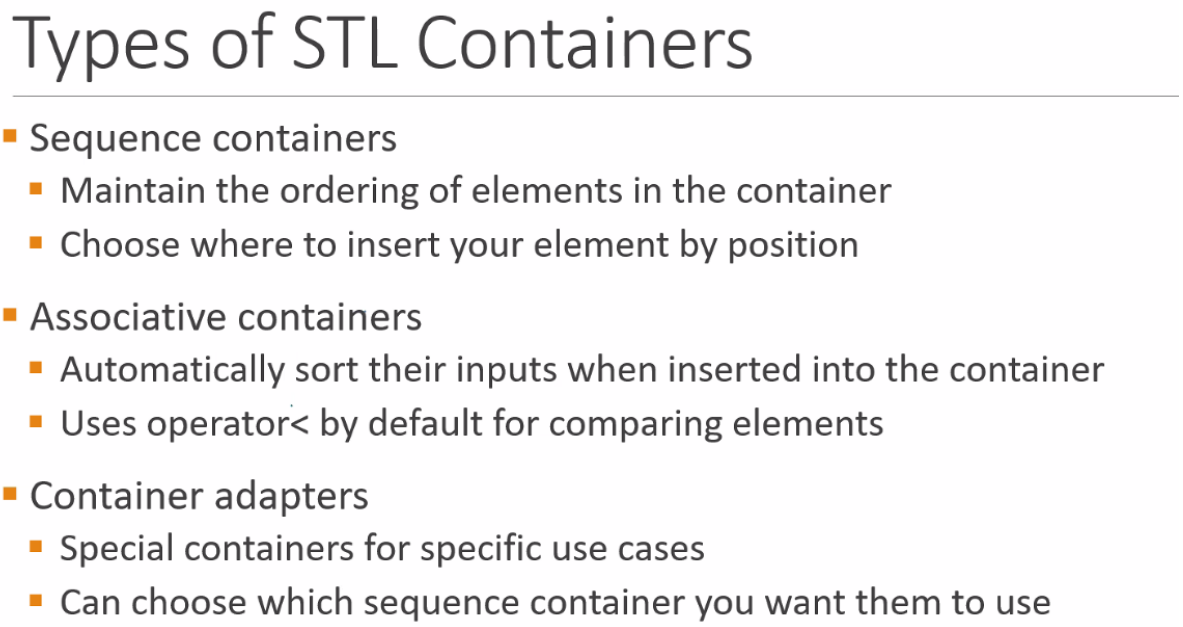
- Class designed to hold and organize multiple instances of another type
  - "Member-of" relationship
- Many different kinds of container classes
  - Most common is array (std::array or std::vector)
- Well defined containers include functions for the following:
  - Create an empty container (constructor)
  - Insert a new object into the container
  - Remove an object from the container
  - Report the number of objects currently in the container
  - Empty the container of all objects
  - Provide access to the stored objects
  - Sort the elements (optional)

Container classes:

- has to do all of the things listed above
- *May* be able to sort the elements in the container.

STL Containers:

- Most used functionality of STL
- Templayes that can store b asically any datatype
- Which container to use is determined by the application requirements (IE you decide)
    - Should data be sorted?
    - How will it be accessed? (Front to back? Randomly?)
    - Will additional data need to be added or removed?



In sequence containers, you need to retain the order of the elements.

Associative containers automatically sort the things put into them. It's "automatically" handles for you -- but it just uses the less-than comparison operator to figure out where things are. **So when you use your own datatypes you need to overload the comparison operator.**

Container adaptors: They just sit on top of some of the basic containers.

Sequence Containers

- <vector> - dynamic array class allowing random access via []
- <deque> - double-ended queue class, a dynamic array that can grow from both ends
- <array> - basic static array class, allows random access
- <list> - doubly linked list, each element has pointers that point at the next and previous elements in the list; no random access
- <forward_list> - singly linked list, each element has a pointer to the next element in the list; no random access

deque seems cool and potentially useful in the future.

Associative containers:

Associative Containers

- <set> - stores unique elements, sorted by value
- <multiset> - allows duplicate elements, sorted by value
- <map> - each element is a pair, called a key/value pair
  - Key is used for sorting and indexing data, must be unique
  - Value is the actual data
  - Also called an associative array
- <multimap> - allows duplicate keys

Will automatically sort what you put into them.

set will **not** allow non-unique elements, but *not sure what happens when you do add a duplicate element*.

Map is kinda confusing, for some reason it has two variables per index it seems. The second is called the key. OK, so it would be

useful for like a dictionary, IE a word is the key, the definition is the value.

Multimaps would actually be more correct, since it allows the same key for multiple values.

Container Adapters

- <stack> - elements operate in a LIFO (last in, first out)
  - Elements both inserted/pushed and removed/popped from the end of the container
  - Defaults to deque, but can also use vector or list
- <queue> - elements operate in a FIFO (first in, first out)
  - Elements are inserted/pushed to the back, and removed/popped from the front
  - Defaults to deque, but can also use list

- <priority_queue> - a queue where the elements are kept sorted
  - When elements are pushed, they get sorted in the queue. When an element is popped, it returns the highest priority item in the queue

How to access all of these elements?

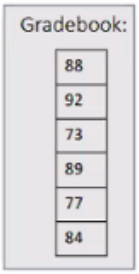
- Traditional for loops
  - Also for each loops (what are these?)
- For loops often contain the most concise code.

for loop

- Let's look at a fictional gradebook example

```
array<int, 6> gradebook = {88, 92, 73, 89, 77, 84};
cout << "Gradebook:" << endl;
for (int i = 0; i < gradebook.size(); i++) {
  cout << gradebook[i] << endl;
}
```

This approach is concise and helps to clearly communicate your intention.



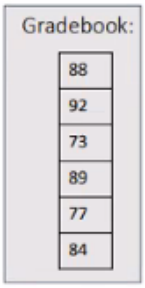
for each loop

- Let's look at a fictional gradebook example

```
array<int, 6> gradebook = {88, 92, 73, 89, 77, 84};
cout << "Gradebook:" << endl;
for (int grade : gradebook) {
  cout << grade << endl;
}
```

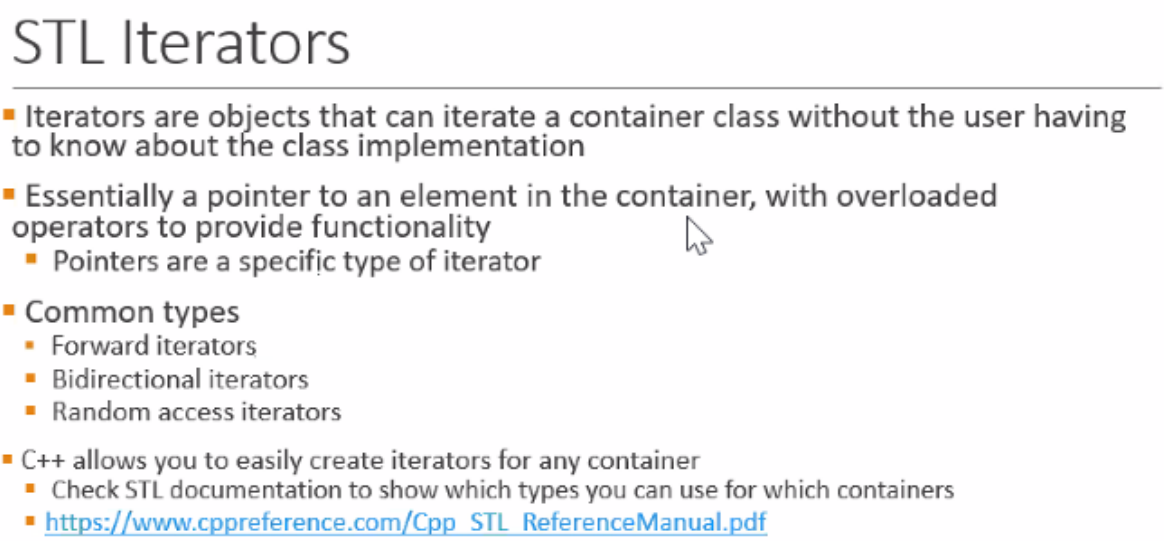
Note:
for-each loops were added in C++11 so old compilers will not work.

A *for-each* loop is a useful technique when you want to access **each** element in a container.



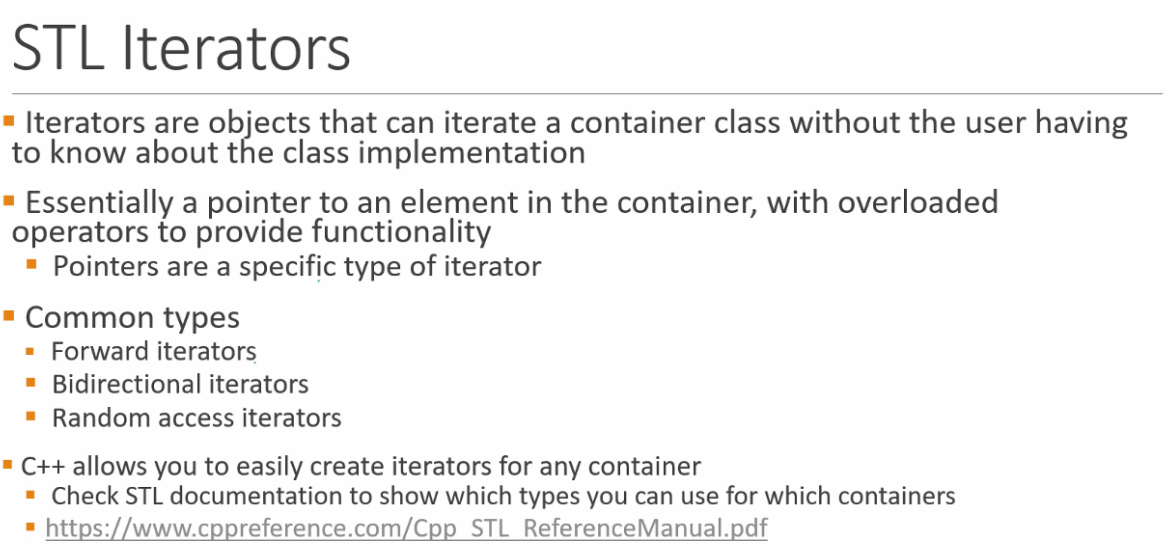
So this basically, for each element in something, does something. You just don't need an index variable, but its limitation is that it **must hit all values** (well, unless you use a break probably)

We'll talk about the below on Friday? She meant Wednesday probably.



We don't have to write constructors, destructors, and assignment operators for the vector class.

5/17/21 - Lecture 22 - STL Iterators



At their most basic, you could think of an iterator as a fancy pointer. In fact, pointers are actually a type of iterator, but very simple. They're just an address, they have no intelligence or member functions. But iterators do have that sort of thing, I guess.

If we were to use a class solely designed to make life with containers easier, there are a few types of iterator classes that are designed for specific types of containers.

So I guess iterators are kind of like if you had an array (or container) and then the iterator just allows you to access elements in those, without having to do things like for loops and stuff.

Do linked lists not know their size? Depends on implementation. They generally don't I guess. But they could, somehow…

https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf

STL Iterators

- Operator* - Returns the element the iterator is currently pointing at
- Operator++ - Moves the interator to the next element in the container
- Operator== and Operator! - Determines if two iterators point at the same element (have to dereference the iterators first)
- Operator= - Assigns the iterator a new position (usually the start or end or end of the container). To assign the value that the iterator is pointing at, dereference first and then assign.

All iterator classes will have at least these operators overloaded.

Dereference: object*
Moves to next element: object++
Operator== and Operator!: determines if two iterators point at the same thing (but dereference them first)
Operator= Assign the iterator a new **POSITION**.

This stuff isn't needed on assignment 4. Is it useful? Some of these may be helpful. Especially algorithm functions which we'll look at today.

STL Iterators

- Each container includes four basic operations to use with assignment operator
  - begin() – returns an iterator for the first element in the container
  - end() – returns an iterator for the element just past the last element in the container
  - cbegin() – returns a const iterator for the first element
  - cend() – returns a const interator for the element just past the last element
- Each container provides at least two types of iterators
  - container::iterator (begin/end)
  - container::const_iterator (cbegin/cend)

begin(): returns an ITERATOR to the first element

end() returns an ITERATOR to the last + 1 iterator of all the elements -- used to check if the iterator you're currently using is equal to this one, then you know you've looped all the way through the elements.

We don't have subtraction operators for every single type of iterator. THis table is from cplusplus.com, shows which kind of iterators you can use with which containers / which operators for which iterators.

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

For each loops have iterators hidden underneath the scenes.

Annoying thing for iterators, to actually define and scope them, it can get pretty long.

DEFINING AN ITERATOR

```
//it commonly used for iterator names
vector<int>::iterator it;
```

auto i = 1; OR auto i = 1.0; //The auto keyword will give it a type given context. First example is int, second is double.

Since putting iterators inside for loops is super long, we can use the auto keyword inside it instead:

```
for(auto it = gradebook.begin(); it != gradebook.end(); it++){
      cout << *it << endl;
}
```

Gradebook Example

- What if we wanted a sorted printout of student grades?
- Can use the multiset container
  - http://www.cplusplus.com/reference/set/multiset/
- Code can then simply iterate over the container and display each grade
  - for-each loop will automatically use iterators for us

```
multiset<int> gradebookSet;
gradebookSet.insert(88);
gradebookSet.insert(92);
gradebookSet.insert(73);
gradebookSet.insert(89);
gradebookSet.insert(77);
gradebookSet.insert(84);

cout << "Gradebook:" << endl;
for (int grade : gradebookSet) {
  cout << grade << endl;
}
```

Gradebook:

73
77
84
88
89
92

```
multiset<int> gradebook;
gradebook.insert(88);
gradebook.insert(92);
gradebook.insert(73);
gradebook.insert(89);
gradebook.insert(89);
gradebook.insert(84);
```

Using a multiset because it allows repeats of element values. Normal set doesn't.

Gradebook Example

- What if we wanted to only display the top 3 grades?
  - for-each won't work anymore
- Use a reverse iterator
  - rbegin()
  - http://www.cplusplus.com/reference/set/multiset/rbegin/

```
multiset<int> gradebookSet;
gradebookSet.insert(88);
gradebookSet.insert(92);
gradebookSet.insert(73);
gradebookSet.insert(89);
gradebookSet.insert(77);
gradebookSet.insert(84);

cout << "Top Grades:" << endl;
auto it = gradebookSet.rbegin();
for (int i = 0; i < 3; i++) {
  cout << *it << endl;
  it++;
}
```

Top Grades:
92
89
88

```
multiset<int> gradebook;
gradebook.insert(88);
gradebook.insert(92);
gradebook.insert(73);
gradebook.insert(89);
gradebook.insert(89);
gradebook.insert(84);

auto it = gradebook.rbegin();

for (int i = 0; i < 3; i++){
    cout << *it << endl;
    it++
}

return 0;
```

STL algorithms:

STL Algorithms

- Generic algorithms for working with the elements of container classes
  - Search
  - Sort
  - Insert
  - Reorder
  - Remove
  - Copy
  - Etc.

5/21/21 - Lecture 23 -

Discussing containers and iterators… Remember those

- Containers
- Iterators

Discussing STLs:

- Generic algorithms for working with the elements of container classes
    - Search
    - Sort
    - Insert
    - Reorder
    - Remove
    - Copy
    - (And more)

So if you need to do something like that, there's an algorithm for you.

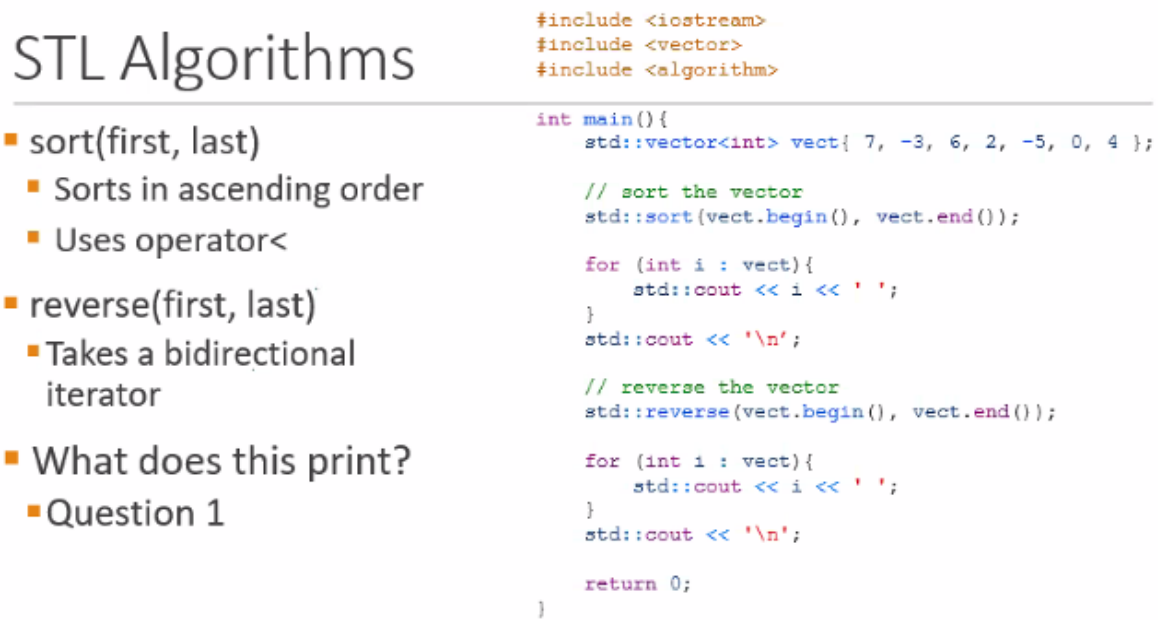
STL algorithms: https://www.cplusplus.com/reference/algorithm/

- IMplemented as **functions** that operate using **iterators**.
- Range of elements provided is (first, last) where last refers to the element *past* the actual last element.

some other stuff she skipped the slide

Common algorithms:

- sort




```
#include<vector>
#include<algorithm>
using namespace std;

int main(){
     vector<int> vect{7, -3, 6, 2, -5, 0, 4};

//if you were gonna start somewhere else, you'd want to make a vector iterator that points to where you want it to start sorting / end sorting.

     sort(vect.begin(), vect.end());
```

```
for (int i : vect){
    cout << i << ' ' << endl;
}
cout << '\n';
```

```
//This will permanently change vector. It will sort it and not return anything.

     reverse(vect.begin(), vect.end());
```

```
for (int i : vect){
    cout << i << ' ' << endl;
}
cout << '\n';
```

So what will print out?

It'll print the values least to greatest, then greatest to least since it reverses them.

You can provide a function for sort!

```
sort(vect.begin(), vect.end(), myFunction);
```

HAS to be a bool function. It will basically use that function instead of a less-than operator.

You don't need to include the brackets--algorithm stuff is smart enough, and it will pass things in itself. If you were to include the () it would then need parameters and that's not what we want.

We could play with templates here, but…

```
bool myFunction (int i, int j){
    return (i > j);
}
```

STL Algorithms

- iota(first, last, starting value)
  - Stores increasing sequence into range provided
- find(first, last, value)
  - Uses operator== for value
- list::insert(iterator, value)
  - Uses bidirectional iterator
- What does this print?
  - Question 2

```
#include <algorithm>
#include <iostream>
#include <list>
#include <numeric>

int main()
{
    std::list<int> li(6);
    std::iota(li.begin(), li.end(), 0);

    // Find the value 3 in the list
    auto it{
        find(li.begin(), li.end(), 3)
    };

    // Insert 8 right before 3.
    li.insert(it, 8);

    for (int i : li) // for each loop with iterators
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}
```

```
//Question 2
list<int> li(6);
```

//create a linked list that will hold 6 ints for us.

```
iota(li.begin(), li.end(), 0);
```

//Fill up the whole list with values, starting at 0 and increasing by ++ (in this case, one). We can overload the increment operator if we need.

```
auto it{find(li.begin(), li.end(), 3)};
```

//FIrst, why in iterator? Well, find returns an iterator to where the value is found.
//So we have to have an iterator to store it in, we have auto so we don't have to type out the auto type.

```
li.insert(it, 8);
```

//Inserts (before?) the iterator. So it sticks an 8 right in front of the 3.

```
for (int i : li)
    cout << i << ' ';
cout << '\n';
```

//This will print out every element in our list, which goes from 0 - 5, and it will have an 8 in front of the 3.

Thus: 0 1 2 8 3 4 5 is what it prints.

STL Algorithms

- find_if(first, last, function)
  - Accepts a unary function that will accept an element in the range and return a bool
  - http://www.cplusplus.com/reference/algorithm/find_if/
- What does this print?
  - Question 3

```
// find_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::find_if
#include <vector>        // std::vector

bool IsOdd (int i) {
  return ((i%2)==1);
}

int main () {
  std::vector<int> myvector;

  myvector.push_back(10);
  myvector.push_back(25);
  myvector.push_back(40);
  myvector.push_back(55);

  std::vector<int>::iterator it = std::find_if
(myvector.begin(), myvector.end(), IsOdd);
  std::cout << "The first odd value is " << *it << '\n';

  return 0;
}
```

```
//Question 3
vector<int> myvector;
myvector.push_back(40);
myvector.push_back(55);
myvector.push_back(10);
myvector.push_back(25);
```

//make a vector, add some values

```
auto it2 = find_if(myvector.begin(), myvector.end(), isOdd);
//isOdd returns true if the value is odd.

cout << *it2 << endl; //Will print out 55.
```

STL Algorithms

- min_element(first, last)
  - Uses operator<
- max_element(first, last)
  - Uses operator<
- What does this print?
  - Question 4

```
#include <algorithm> // std::min_element and std::max_element
#include <iostream>
#include <list>
#include <numeric> // std::iota

int main()
{
    std::list<int> li(6);

    std::iota(li.begin(), li.end(), 0);

    std::cout << *std::min_element(li.begin(), li.end()) << ' '
              << *std::max_element(li.begin(), li.end()) << '\n';
    return 0;
}
```

Pretty simple to understand. Uses < comparison operator, and **THESE FUNCTIONS RETURN ITERATORS.** So we need to dereference it, OR store in an iterator if we want to use it later.

5/17/21 - Lecture 24 - Sorting algorithms

Program 5 look-ahead:

Big things:

- Implementing linked lists
- Sorting algorithms

The number of lines of code and number of files for this assignment will be significantly smaller than the others, **BUT** that being said, it can get you a bit because you have to implement a recursive algorithm for sorting. Even if it seems simple, don't put it off. VERY conceptual heavy.

FOR NEXT LAB: Do it using merge-sort. This is useful for next assignment.

Readings this week may be useful… It's more of an interactive lesson to think about concepts and stuff! Use these if I need help with concepts, or maybe just use them anyways.

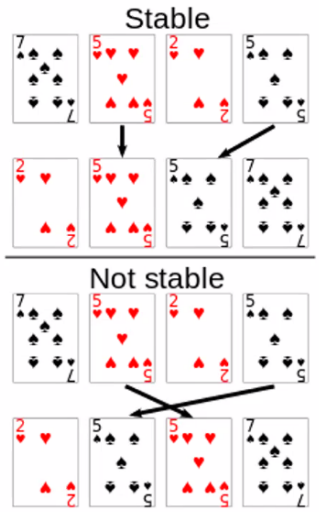
Sorting:

- Sorts in ascending order or descending
- Uses a comparison operator, which can be overloaded
- Can be used to optimize code, since sorted containers are easier to find data inside
- There are lots of sort methods:
    - Bubble, selections, insertion, merge, quick, heap, counting, shell, comb, bucket, radix, cocktail, cycle, sleep, tim…

How to choose a method?

How do we choose a method?

- Runtime
  - Best, average, and worst case
- Space usage
  - Can it be sorted in-place, internally or does it require external memory
- Stability
  - If two elements are equal, they appear in the same order in the output as they were in the input
  - Needed because data is often sorted based on only part of the data



STABLE VS UNSTABLE:

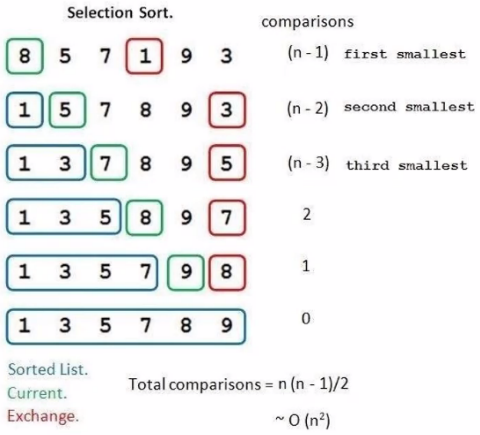
Stable sort means a minimum that occurs first will still occur first even if there's another minimum that's the same value.
Unstable means those two may be swapped, so first-minimum order may not be preserved.

https://visualgo.net/en/sorting

Good visualization of sorting algorithms.

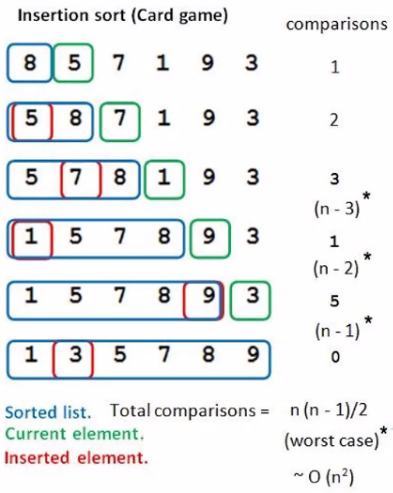
Selection sort

- Data has two sections
  - Already sorted
  - Remaining unsorted
- Repeatedly finds the minimum element from the unsorted portion of the data and puts it at the beginning
- Not stable by default
- Consistent runtime
- In-place, no significant extra space needed



Insertion sort

- Somewhat similar to selection
  - Divides the data into sorted and unsorted
- Elements in the sorted portion are only sorted with respect to each other (not the entire array)
  - An unsorted element could still be inserted somewhere in the middle of the sorted portion
- Think of how you organize a hand of playing cards
- Stable
- Best used for smaller datasets

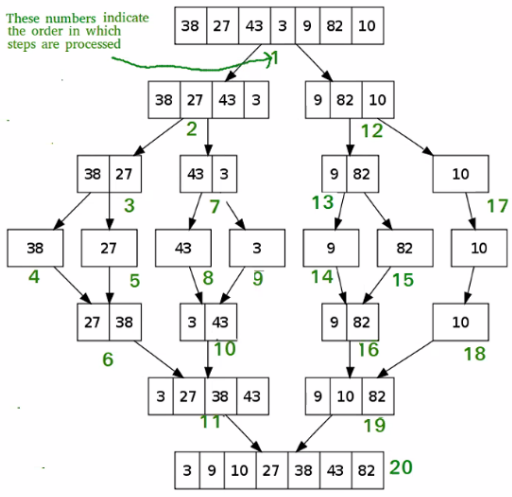


Very similar to selection, we're still looking at data as a part that's already sorted and a part that's unsorted

Instead of continuously looking for the minimum value and then putting it in the sorted array, the elements in the sorted portion are **only sorted in reference to themselves.**

Merge sort

- A divide and conquer algorithm
  - Divide – Break the given problem into subproblems of the same type
  - Conquer – Solve the subproblems
  - Combine – Appropriately combine the answers
- Key concepts
  - 1) When two arrays are already sorted, it is easy to combine them into one sorted array
  - 2) An array of length "1" is already sorted
- Stable
- Often used for large datasets, and externally



So basically divide into a set of pieces we can sort

Then merge each set of two into one set and resort as it merges

Repeat until you only have 1 set.

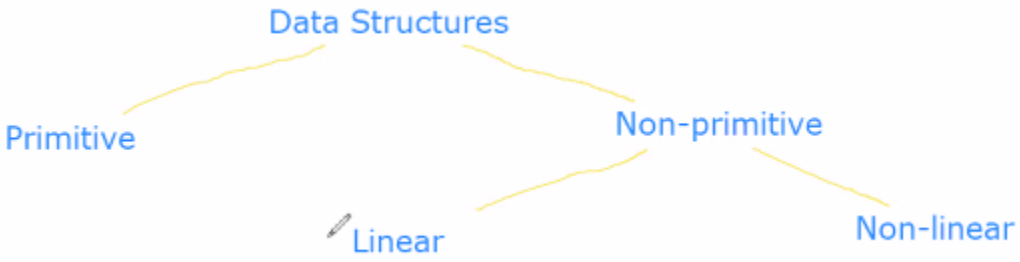
Concepts or rules:

- When two arrays are already sorted, it's easy to combine them into one array
- An array of length 1 is already sorted.

5/26/21 - Lecture 25 - Data structures

Data structures:

- A particular way of organizing data in a computer so it can be used effectively.
- Some examples:
    - Array
    - Linked list
    - Binary tree
    - Heap



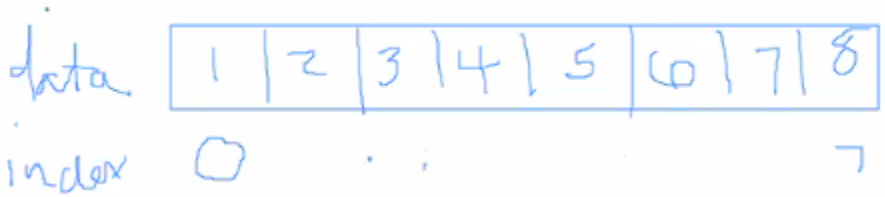
- Primitive data structures: primitive datatypes
- Non-primitive (Arrays and stuff?)
    - Linear:
        - Static structures
            - Static
        - Dynamic structures
            - Vectors
            - Stacks, queues, etc
    - Non-linear:
        - Graphs, trees, etc

Linear Data Structures

- Data structure where elements are arranged sequentially or linearly
- Elements are attached to the previous and next elements
- Single level
  - Can traverse through all elements in a single run through
- Examples
  - Array
  - Stack
  - Queue
  - Linked list

Just start at one end and then increment through them without worrying about its shape.

Each element is connected to the next
Single level, 1d is a linear data structure.



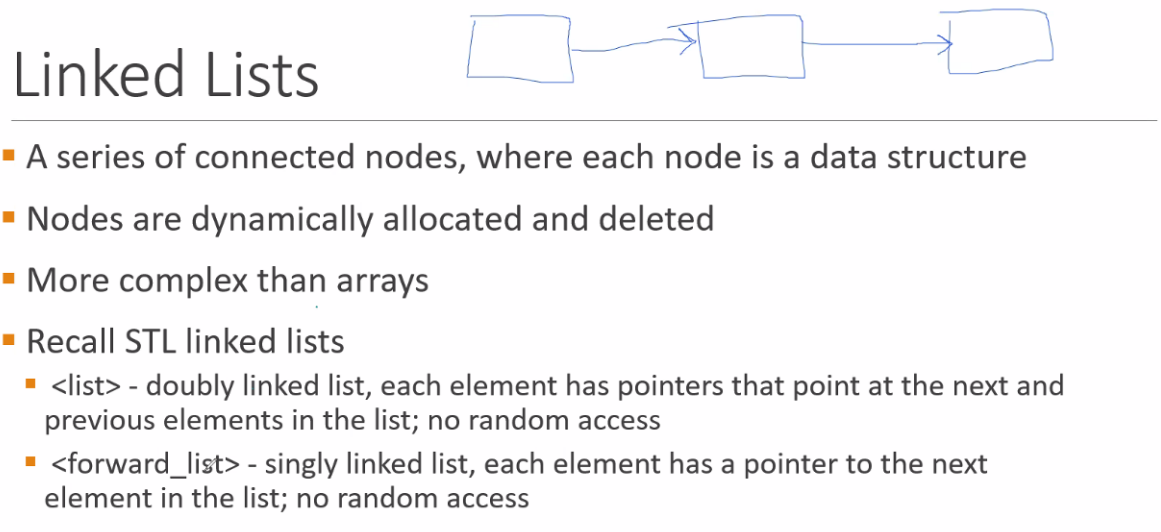
Cons of arrays:

- Inserting things is resource-intensive
- Hard to resize
- Doesn't handle people trying to access stuff outside the structure

Stack cons:

- To get down to a certain thing, you have to push/pop things a bunch.
- But it is convenient when you want First in is First out behaviour.

Linked lists:

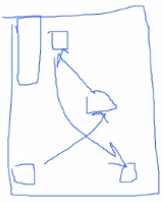


Each node knows who comes after for singly-linked lists, doubly-linked lists know who comes before as well.

Pros of linked lists versus arrays/dynamic arrays:

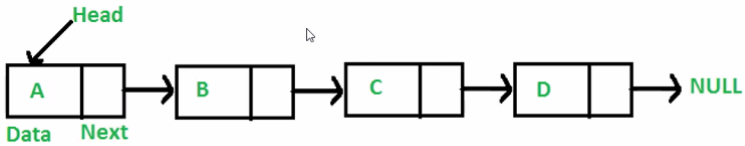
- Easier to resize and delete things
- They don't require contiguous memory blocks like arrays do (so more memory efficient?)
- They do take more memory **overall**, because they also store a pointer to memory
- But they may be more memory efficient because they can fill in the gaps in memory versus a vector, for example. But comparing a straight array to a straight linked list, the linked list takes more memory.
- We **don't get** random access, not like we're used to with indexes. You *have* to start at whatever pointer you're at, and increment through. On average, it's longer to access elements.
- The linked-list doesn't know the length of itself.

- Why wouldn't we just use vectors?
  - Linked lists are faster
  - Easier to insert/delete elements
- Cons of linked lists
  - No random access
  - Extra memory space for a pointer is required for element



Linked list structure:

- Each node contains data members and pointer
  - "Reference" or "successor" pointer
- The head points to the first node
- The last node points to null

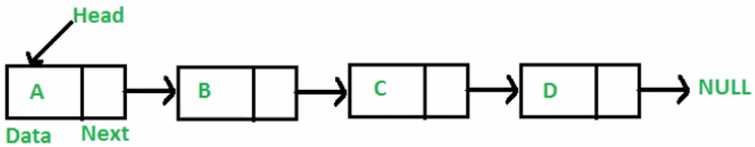


So once you hit null on the pointers, you can't go anywhere else. No ability to go out of bounds. The last node is often called the tail.

- Need a datatype for the node
  - Can use a struct or class depending on your implementation
  - Self-referential data type/structure
- Need a class for managing the nodes

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


class LinkedList{
private:
    node *head;
public:
    LinkedList(){
        head=NULL;
    }
};
```

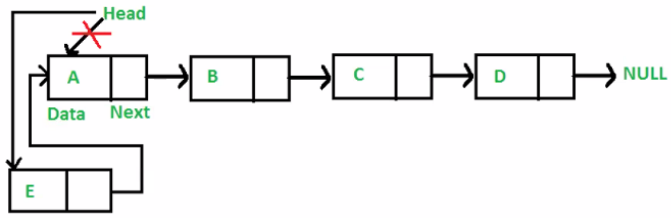


So it looks like we can actually define our own linked list? Like, it's not a template data structure, we just make it ourselves?

Two key operations any structure for organizing data:

- **Adding data**
- **Removing data**

- Inserting nodes
  - At the front
    - Make a new node
    - Put in data
    - Set next of new node as head
    - Set head to point at new node




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

class LinkedList{
private:
    node *head;
public:
    LinkedList(){
        head=NULL;
    }
};
```

Three places to look at adding things:

- At the front
  - Make a new node in space
  - Set its node's pointer to the head of the linked list
  - Redefine the pointer to head to point at the new node's address
- In the middle at n
  - Start at the beginning (head)
  - traverses until it gets to the n-1th element
  - temporarily store the n-1th element and the nth element pointers
  - create a new node in space, make it point at the nth element
  - then make the previous element point at the nth element.
- At the end
  - Make a node in space
  - Set the old tail's nullptr to instead point to the new node's address
  - Set the tail's pointer to point at the new node's address

Node can be class or struct -- linkedlist has to be a class.

We'll learn about removing stuff on Friday.

Presentation/Plug about InnovationX entrepreneurial stuff at OSU, they have workshops and classes and speakers and actual help for launching a product. Launch academy class helps you launch the class.

Chat with a consultant about all things "entrepreneurial" in the Business Idea Development Studio:
https://oregonstate.qualtrics.com/jfe/form/SV_3w9oSPjPNGbL2S1

Join Launch Academy:
https://oregonstate.qualtrics.com/jfe/form/SV_1yOXbN8UL5EIl49

Get info about all the things in the weekly Launch newsletter:
https://oregonstate.us16.list-manage.com/subscribe?u=734437f5e8313d1df2c14ff2c&id=72eb106db2

Event information:
https://business.oregonstate.edu/faculty-and-research/centers-strategic-initiatives/innovationx/events

Or, email any of us:
Michelle.Marie@oregonstate.edu
okazamax@oregonstate.edu

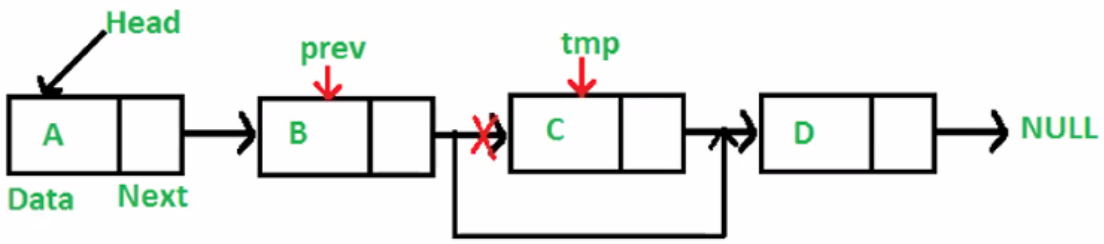
5/28/21 - Lecture 26 - Linked Lists & Recursion Review

Refresh from last class…

Linked list structure:

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

Or could also have a
      node *previous
```



Keep very careful track of your pointers. If you lose them, they're GONE FOREVER, AAAAAAAAUGH

Removing (the head) from a list:

1. Check if there is anything at all in the linked list
2. Temporarily store the pointer that the head points to
3. Then, set the *head* of the linked list to the pointer we stored temporarily.
4. Then we can delete the temporary pointer variable.

Big O of 1, so it's the fastest it can be.

It's a good idea to add a function for our linked-list class that checks for nothing before trying to sort things.

Removing one from the end:

1. Check if empty
2. Create a temp pointing to the pointer at the head
3. while(!nullptr), we just go to the next pointer

4. When it *is* null, then set the previous node's pointer to nullptr
5. Delete the tail
6. Set the tail pointer to the temp thing that held the previous pointer
7. delete temporary things

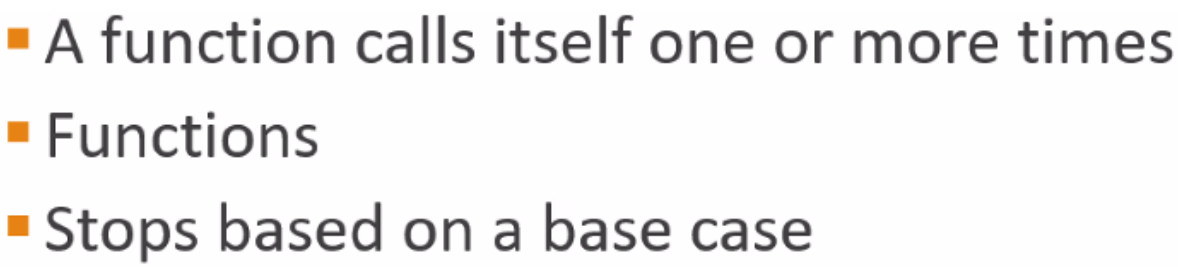
For removing one from somewhere in the middle:

1. Same thing, check if empty then make a temp pointer to the first element
2. loop through setting the temp pointer to the last pointer until we get to the index before we want
3. Set a temp to the pointer for the one deleting
4. Set a temp to the pointer for the one after deleting
    a. (Now we have 3 pointers: before, the deleted one, the after one)
5. Then set the before one's pointer to the after one's pointer
6. Then delete the deleted one, and the temp ones.

In c++, you could *technically* write an overloaded operator for a struct as a member function. But you *shouldn't*. I think it's kinda a sacred thingy, but probably has good reason too.

How would we find the length of a linked list?

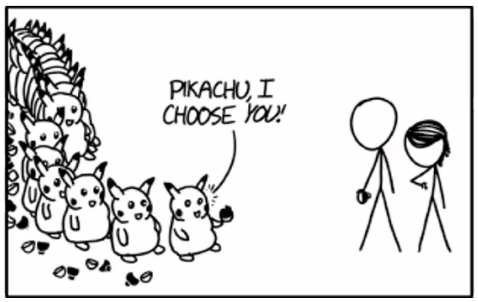
- While loop that iterates through until nullptr and has a counter to keep track of it
- **OR a recursive function:**



Recursion

- When a function calls itself one or more times
- Form of repetition
- Typically used to perform the same operation on a smaller subset and then build the result based on what is returned from the smaller case
- Normally has at least one base case for stopping
- Based on inductive logic
  - Specific instances lead to general conclusion

"Win by Induction"



https://xkcd.com/1516/

Iteration vs. Recursion

- Anything that can be done iteratively can be done recursively and vice versa
  - Some problems naturally lend themselves towards one mode of thinking vs. the other
- Recursion is more costly in terms of processor time and memory space
  - Uses stack to store the set of new local variables and parameters every time the function is called
  - Infinite recursion can lead to a system crash, vs infinite iteration just consumes CPU cycles
- Recursion makes code smaller/more readable

Definite pros and cons to each method. Some things will be more easy or readable to do via recursion rather than iteration.

But recursion is more processor-consuming and memory intensive. Each time you call a recursive function, it pops onto the stack, cause each function has its set of variables. So until you get to the base case, you'll have a whole bunch of variables for every single instance of the function.

So why would we use recursive functions??

Besides being more readable, it can save you coding time.

A good example is factorials:

- n! = n * (n-1) * (n-2) * … * 2 * 1 for all n > 0.
- Base case is when n=1.

Iterative solution (one example):

… oops

Recursive solution:

```
int factorial(int n){
    if (n==1)
        return 1;
    return n*factorial(n-1); //recursive calling
}
```

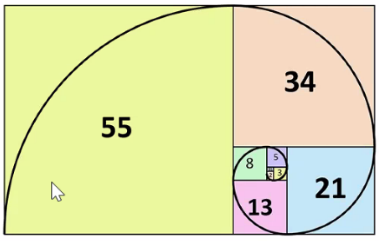
```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

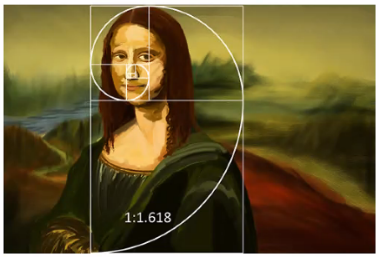
This also fits very well how we would look at the factorial problem in math.

IMPORTANT: Since there aren't live demos for program 5, make sure there are **A TON OF COMMENTS** in code explaining things for the TAs, since they can't ask things about it.

Another Example – Fibonacci Series

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…
- F(0) = 1, F(1) = 1, F(n) = F(n-1)+F(n-2)
- Would you use iteration or recursion to calculate out a Fibonacci number?





- Recursion

```
long long fibonacci(int n){
    if (n>1)
        return fibonacci(n-1)+fibonacci(n-2);
    else
        return 1;
}
```

- Iteration

```
long long fibonacci(int n){
    int sum = 1;
    int temp1 = 1;
    int temp2 = 1;
    for (int i=2; i<=n; i++){
        temp1 = temp2;
        temp2 = sum;
        sum = temp1 + temp2;
    }
    return sum;
}
```

Another Example – Fibonacci Series

- Recursion

```
long long fibonacci(int n){
    if (n>1)
        return fibonacci(n-1)+fibonacci(n-2);
    else
        return 1;
}
```

- Iteration

```
long long fibonacci(int n){
    int sum = 1;
    int temp1 = 1;
    int temp2 = 1;
    for (int i=2; i<=n; i++){
        temp1 = temp2;
        temp2 = sum;
        sum = temp1 + temp2;
    }
    return sum;
}
```

Versus:

```
long long fibonacci(int n, long long prev1, long long prev2) {
    if (n<2)
        return 1;
    else if (n==2)
        return prev1 + prev2;
    else
        return fibonacci(n-1, prev2, prev1+prev2);
}
```

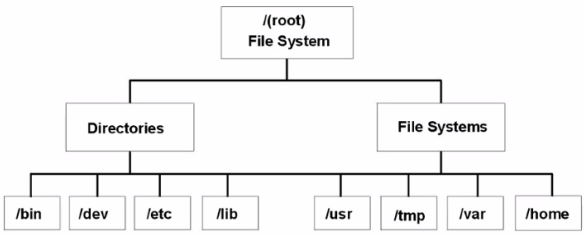
Another Classic Example

- Towers of Hanoi
- Rules
  - Only one disk can be moved among the towers at any given time.
  - Only the "top" disk can be removed.
  - No large disk can sit over a small disk.

```
solve_tower(int n, int source, int dest, int other){
    if (n<=1)
        cout << "Move from " << source << " to " << dest << ".";
    else {
        solve_tower(n-1, source, other, dest);
        solve_tower(1, source, dest, other);
        solve_tower(n-1, other, dest, source);
    }
```

Recursion

- Common applications
  - Language translators
  - Compilers
  - Sorting & searching
    - Especially with non-linear data structures
- When should I use it?
  - When you have a problem that can be solved by breaking it up into smaller repetitive problems that could have too many special cases for a clean iterative approach
    - And when you are OK with using more memory



So, recursion isn't always best.
Common applications of recursion:

- Translators for language
- Compilers
- Sorting & searching
  - Especially with non-linear data structures

When to use it?

- When you have a problem that can be solved by breaking it up into smaller repetitive problems that could have too many special cases for a clean iterative approach
- *and when you are OK with using more memory*

5/28/21 - Lecture 27 - Big O and runtime complexity

Runtime Complexity

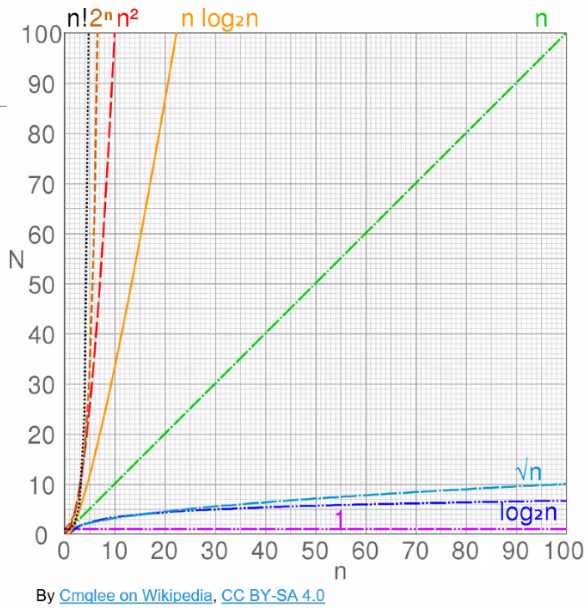
- Algorithms take time to run
- We need a way to characterize algorithms or data structures that is completely platform independent
  - AKA, doesn't depend on hardware, OS, language, etc.
  - Algorithms themselves are platform independent
- Typically talk about runtime in an abstract sense
  - Can evaluate based on input, number and kind of steps used
  - Lets us look at what happens with increasingly large datasets
  - Big O – worst case bound
  - Big Ω - best case bound
  - We ignore "constant" factors

Complexity Analysis

- Describing how runtime or memory usage changes relative to a change in input size "n"
- Particularly interested in how algorithms behave towards the limit, as "n" approaches ∞
  - Some may have expensive one-time startup costs but be efficient afterwards
    - Linear algorithm
  - Some may have inexpensive startup costs but be inefficient afterwards
    - Exponential algorithm
- Describe in terms of orders of growth
  - Grows on the order of some mathematical function if that function provides an upper bound on the runtime beyond a certain input size n

Big O Notation

- How we express an algorithm's order of growth
- Common growth order functions
  - O(1) – constant complexity
  - O(log n) – log-n complexity
  - O(√n) – root-n complexity
  - O(n) – linear complexity
  - O(n log n) – n-log-n complexity
  - O(n^2) – quadratic complexity
  - O(n^3) – cubic complexity
  - O(2^n) – exponential complexity
  - O(n!) – factorial complexity



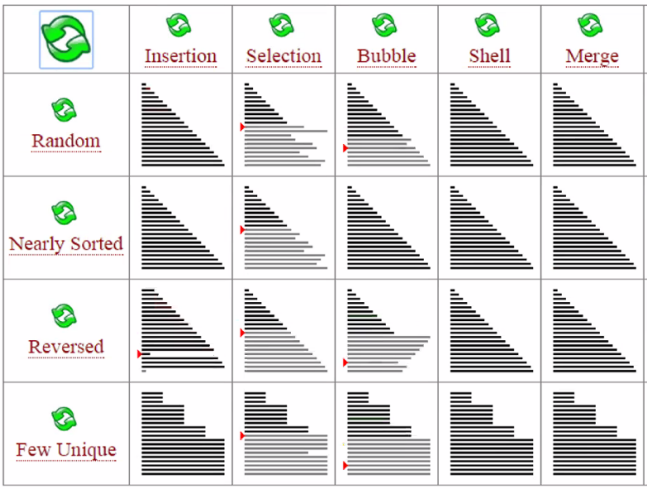
By Cmglee on Wikipedia, CC BY-SA 4.0

So this graph shows sort of complexity versus number of inputs I think

So if you want something to be fast, keep it in linear or lower if possible. But, it's hard to do if you try to do complex things.

Complexity for the sorting we've seen…

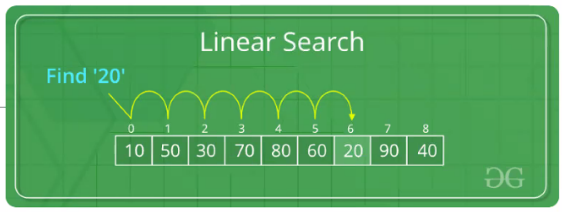
- Bubble sort
  - O(n^2) – also average
  - Ω(n) – when list is already sorted
- Selection sort
  - O(n^2)
  - Ω(n^2)
- Insertion sort
  - O(n^2) – also average
  - Ω(n) – when list is already sorted
- Merge sort
  - O(n log n)
  - Ω(n log n)

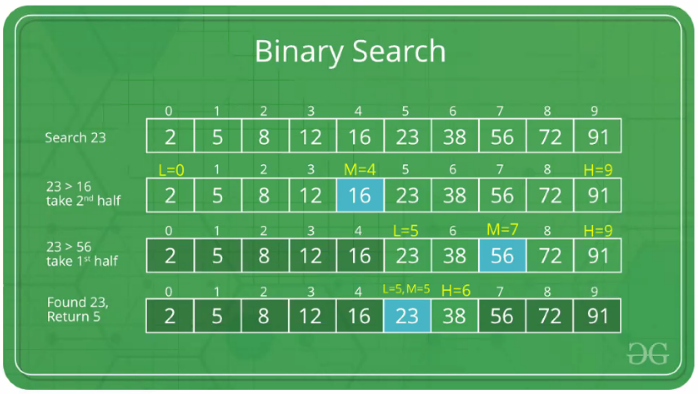


So depending on your application, you may change which sorting algorithm you use depending on the kind of data you're searching.

Searching

- Two basic types of searching
  - Linear
    - Go through sequentially one item at a time
    - Doesn't require sorting
  - Binary
    - Start at the middle of a list, comparing the value and choosing which half to look in
    - Do it over and over until found
    - Requires sorted list





Just like the basic sorting algorithms, there are also two basic searching algorithms.

- **Linear search**: exactly what you think it is, goes through one at a time.
- **Doesn't require any sorting.**

- **Binary searching: requires sorted list**
- Start in middle of list, take first or last ½ of data based on value searching for
- Do this over and over until the value is found.

- Linear search
  - O(n)
  - Ω(1)
- Binary search
  - O(log n)
  - Ω(1)

(Search complexity)

Simple example:

```
sum = 0;
for (i = 0; i < n; i++) {
   sum += array[i];
}
return sum;
```

Line by line:

1. O(constant)
2. O(constant)
3. }
4. O(constant)

Obviously though, the for loop will take time, it will take O(n), since it's dependent on n.

Whatever is inside a loop has a multiplicative effect:

So inside the loop, we have O(n)*O(1) + O(1) + O(1) for the whole algorithm.

(O(1) could also be written O(c1) for constant time c1, which is platform dependent.)

So summing is a linear function in terms of both worst case and best case scenario.

So if it takes 15 milliseconds to sum 5,000, then it will take 60 ms (it takes 0.003ms per iteration)

What about non-linear times?

- Bubble sort is $O(n^2)$
- What if we double the size of the list for a bubble sort? What happens to the runtime?
  - Say, go from 5,000 elements to 10,000 elements

- Means that runtime is proportional to n^2
- For two given sizes and their runtimes, the ratio of Big O should equal the ratio of real clock runtimes
- $\frac{n1^2}{n2^2} = \frac{t1}{t2}$
- Double input size so n2 = 2*n1
- $\frac{n1^2}{(2(n1))^2} = \frac{t1}{t2}$
- Solve for t2, we see that t2=4*t1

So if you double it, it will take 4x as long.

Determining Complexity

- Loops are one of the main determinants of complexity
- O(n)
  - `for (int i=0; i<n; i++)`
- O(log n)
  - `for (int i=1; i<n; i*=2)`
  - `for (int i=n; i>0; i/=2)`
- When loops are nested, their individual growth orders are multiplied to compute the functions overall complexity

Most of the time, loops will be our most dominant aspect of runtime complexity.

- When an O(n) loop is nested within another O(n) loop, the total complexity is O(n^2)
  - ```
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            …
        }
    }
    ```

So what is it when you have O(log n) loop inside O(n) loop?

What is it when you have O(n) loop inside O(n) loop inside O(n) loop?
    ```

So double, triple for loops can get really, really slow.

- O(n log n)
- O(n^3)

So making faster algorithms often involves making smaller, reduced, or removed loops. (Cut them down, or make them not dependant on data size)

Constants

- In the real world, constants can still impact our design choices
- Your program will only perform as well as your design
- Suppose you have two algorithms
  - Algorithm A) 1,000,000n -> O(n)
  - Algorithm B) 2n^2 -> O(n^2)
  - Which one is better?
    - It depends
    - For n< ~700,000, Algorithm 2 will actually run faster
    - Know your data in addition to knowing your algorithms

So which you use depends on your data. A is linear, but below around 700,000, the big O graph for 2n^2 is lower than linear.

5/28/21 - Lecture 28 - Final, in-class activity

What is the big O for this code?

```
void printFirstElementOfArray(int arr[])
{
    printf("First element of array = %d",arr[0]);
}
```

O(constant)

Big O is worst case scenario for as n -> infinity of the number of data you need to process.

looked at in growth-orders, not specific numbers.

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

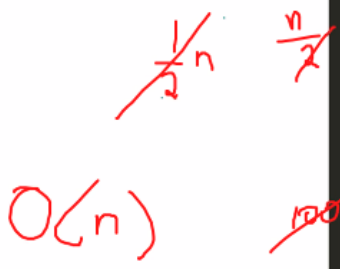
O(n) * O(n) * O(1) = O(n^2)

```
void printAllItemsTwice(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

O(2n) -- but that's not big O -- it's a linear difficulty, so we care about the dependency, not the constants. As n -> infinity, 2 becomes peanuts. So, it's just O(n).

- What is the Big O for this code?




```
void printFirstItemThenFirstHalfThenSayHi100Times(int arr[], int size)
{
    printf("First element of array = %d\n",arr[0]);

    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Still O(n). We have one n times some constants, so still just n.

```
void printAllNumbersThenAllPairSums(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d\n", arr[i] + arr[j]);
        }
    }
}
```

O(n^2)

Final stuff & things left to do:

Program 5 -- Big thing to remember:

- No live demo -- README.TXT IS IMPORTANT -- note everything here!
- Explain things you might say in person in the code.
- Tarfile! Name it XD Makefile should have this already…

Final info:

- Friday 9:30-11:20 PST
- 100 points, 60 questions, similar to the midterm (and quizzes?)
- Random, one at a time
- Zoom lecture required for attendance, updates, and questions.