CS162 Notes Spring 2021

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

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

CS162 Notes Spring 2021

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];

CS162 Notes Spring 2021

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
CS162 Notes Spring 2021 **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

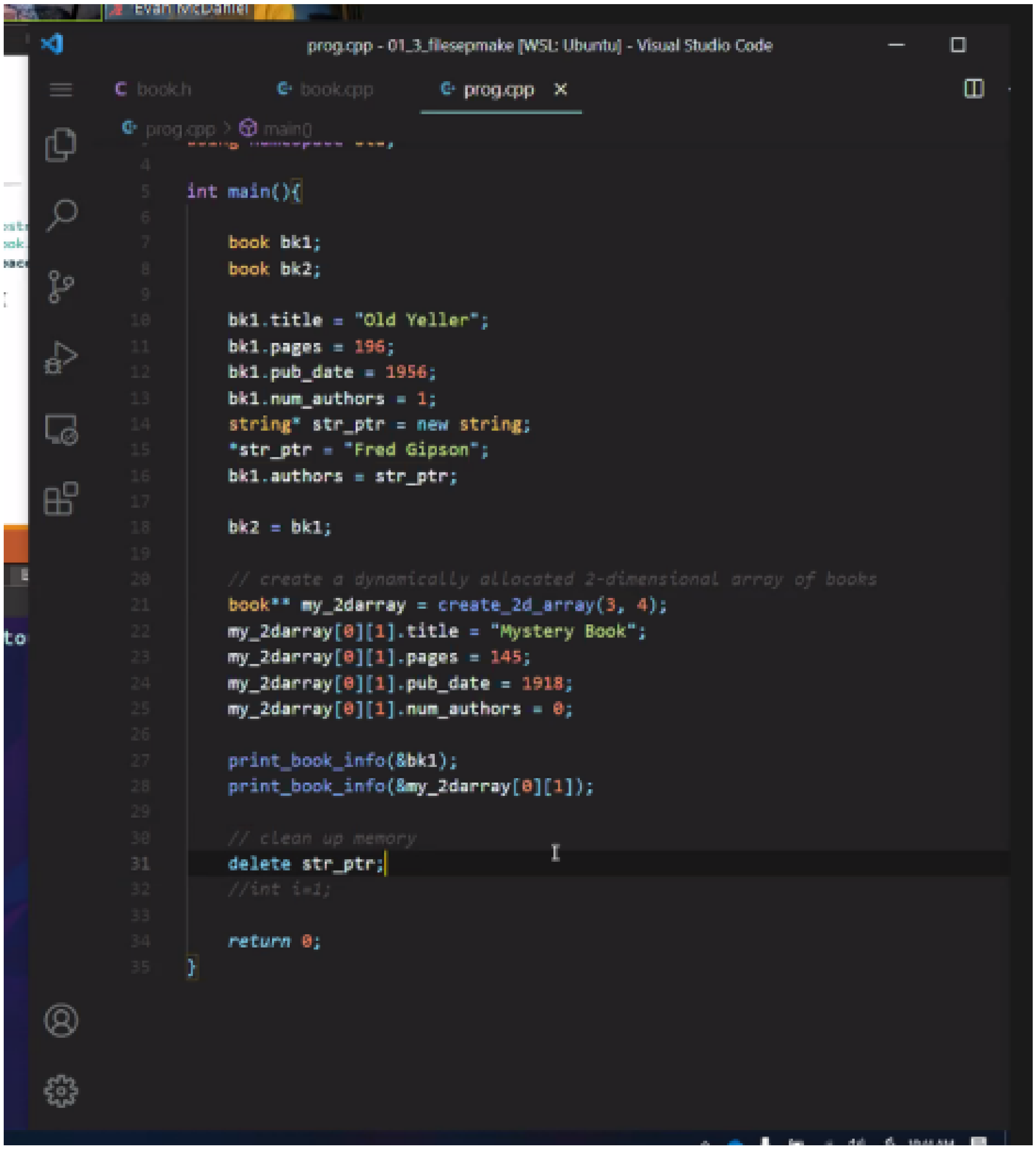
- This has the function declarations etc

Book.cpp

- This holds functions and stuff

Prog.cpp

- Prog holds int main


prog.cpp - 01_3_filesepmake [WSL: Ubuntu] - Visual Studio Code
book.h
book.cpp
prog.cpp
prog.cpp > main()

```cpp
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;
}
```

CS162 Notes Spring 2021

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

Here's an example makefile:

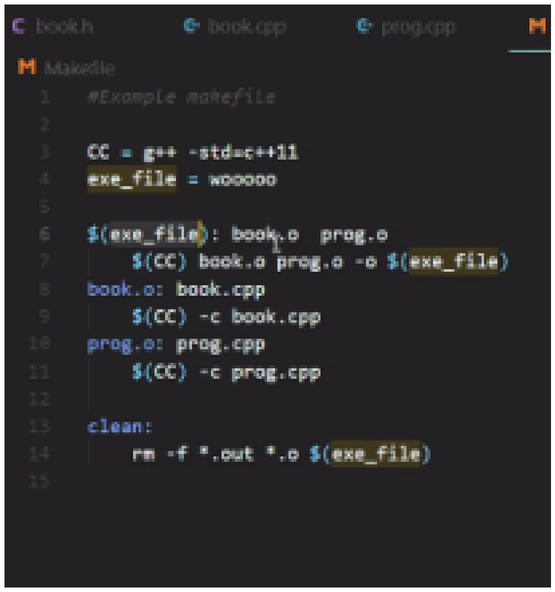
```
#Hashtags are comments

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



```
#Example makefile

CC = g++ -std=c++11
exe_file = wooooo

$(exe_file): book.o  prog.o
	$(CC) book.o prog.o -o $(exe_file)
book.o: book.cpp
	$(CC) -c book.cpp
prog.o: prog.cpp
	$(CC) -c prog.cpp

clean:
	rm -f *.out *.o $(exe_file)
```

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.

CS162 Notes Spring 2021 **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;
}
```

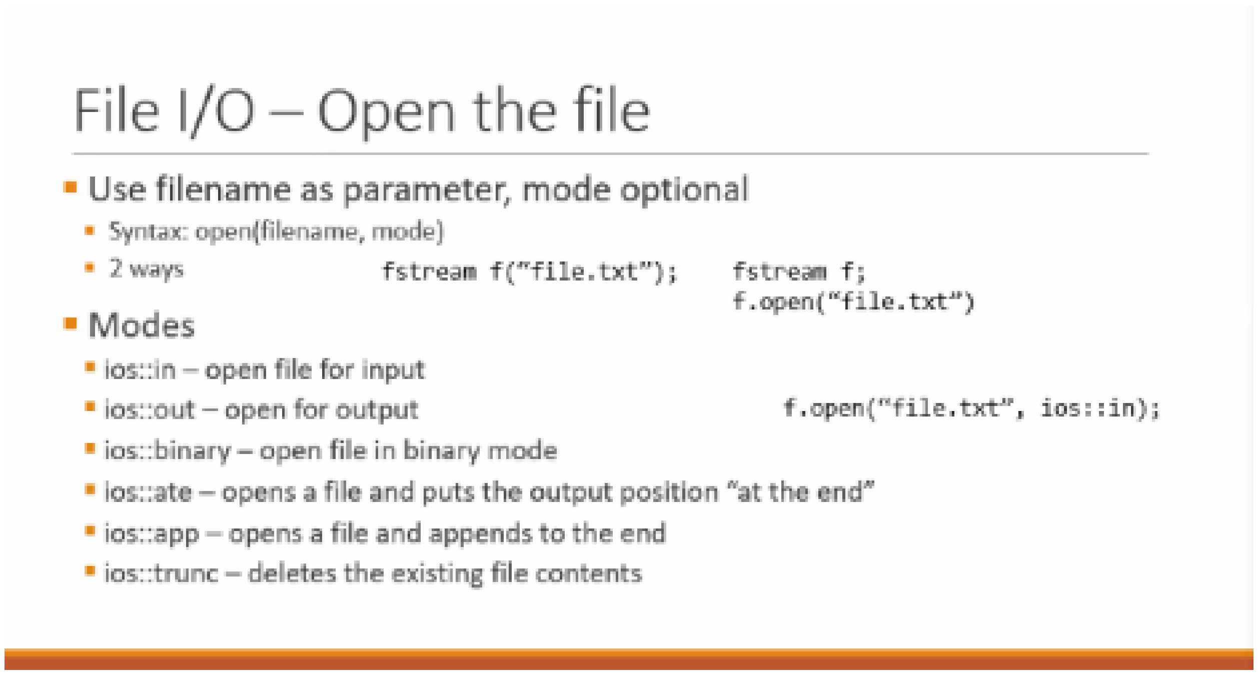
Example of using the class/filestream:

```
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

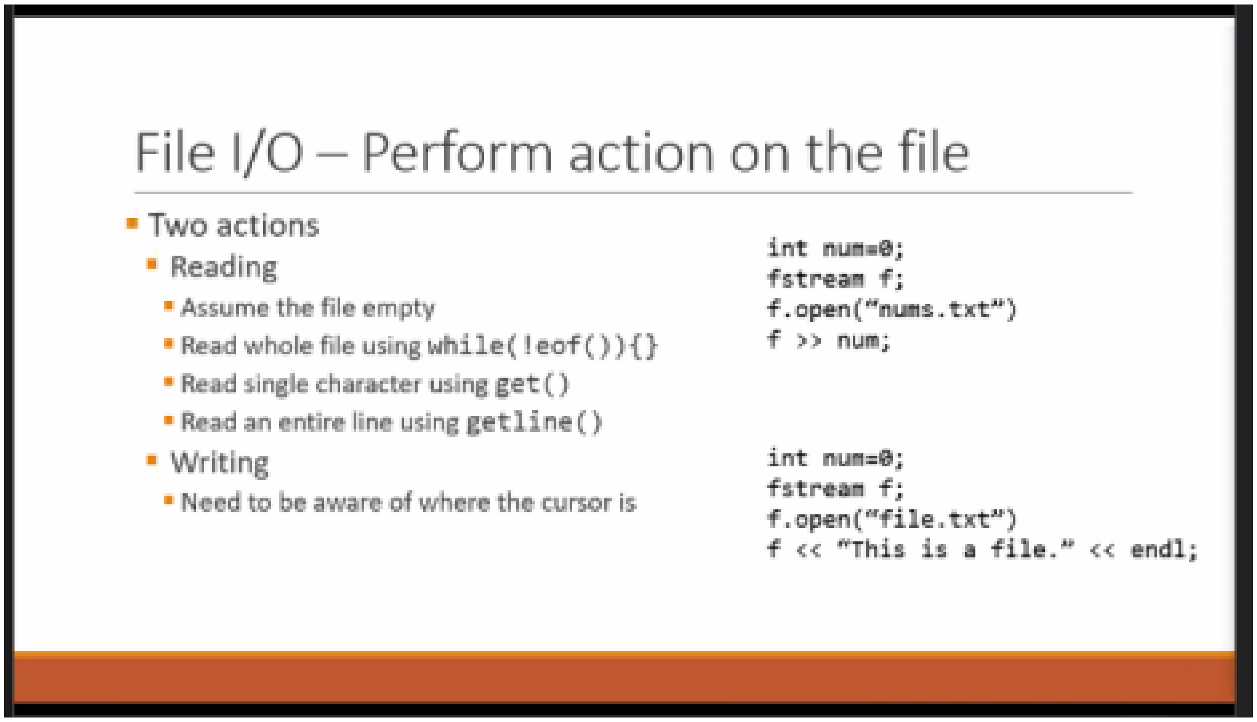
- Use filename as parameter, mode optional
  - Syntax: open(filename, mode)
  - 2 ways

```
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:



File I/O – Perform action on the file

- Two actions
  - Reading
    - Assume the file empty
    - Read whole file using `while(!eof()){}`
    - Read single character using `get()`
    - Read an entire line using `getline()`
  - Writing
    - Need to be aware of where the cursor is

```
int num=0;
fstream f;
f.open("nums.txt")
f >> num;
```

```
int num=0;
fstream f;
f.open("file.txt")
f << "This is a file." << endl;
```

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();

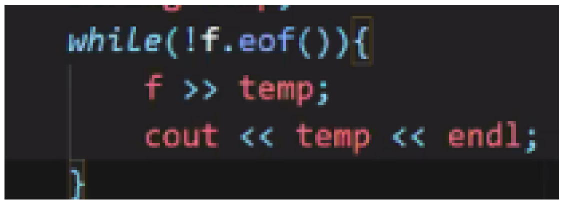
Parsing files:

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



```
while(!f.eof()){
    f >> temp;
    cout << temp << endl;
}
```

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:

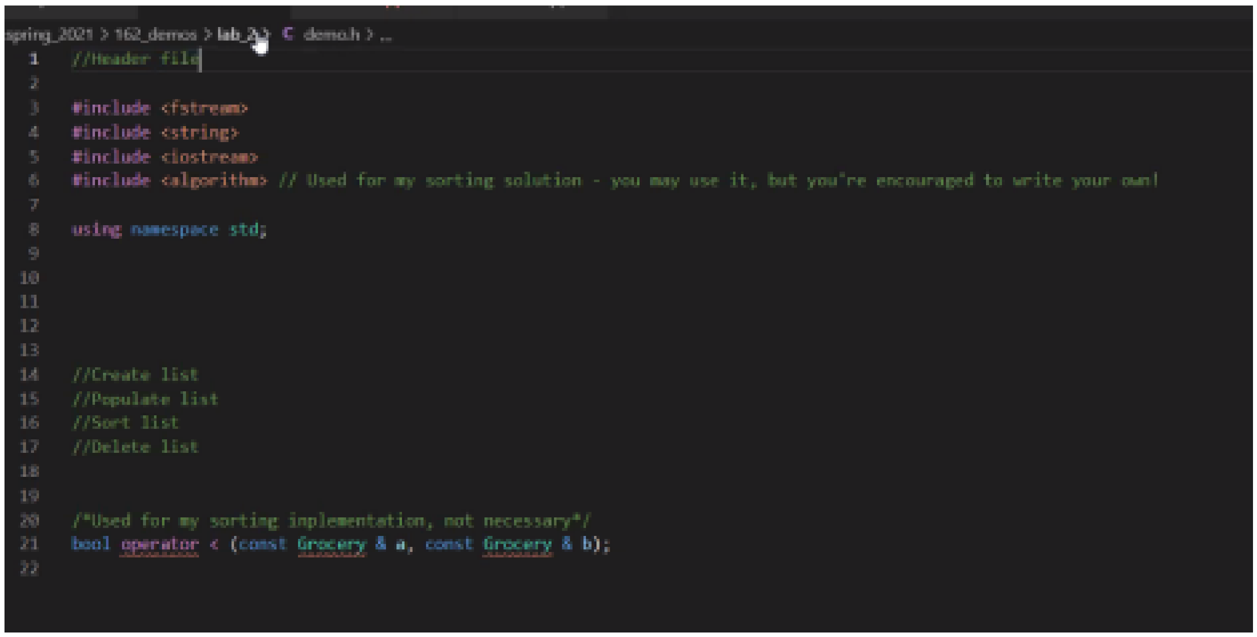
```
getline(filestreamobject, stringtoreadinto, parsechar);
//So this is used like
getline(f, title, ‘,’);

//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;
```

CS162 Notes Spring 2021

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



spring_2021 > 162_demos > lab_A > demo.h > ...

```
//Header file

#include <fstream>
#include <string>
#include <iostream>
#include <algorithm> // Used for my sorting solution - you may use it, but you're encouraged to write your own!

using namespace std;





//Create list
//Populate list
//Sort list
//Delete list


/*Used for my sorting inplementation, not necessary*/
bool operator < (const Grocery & a, const Grocery & b);

```

CS162 Notes Spring 2021 **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

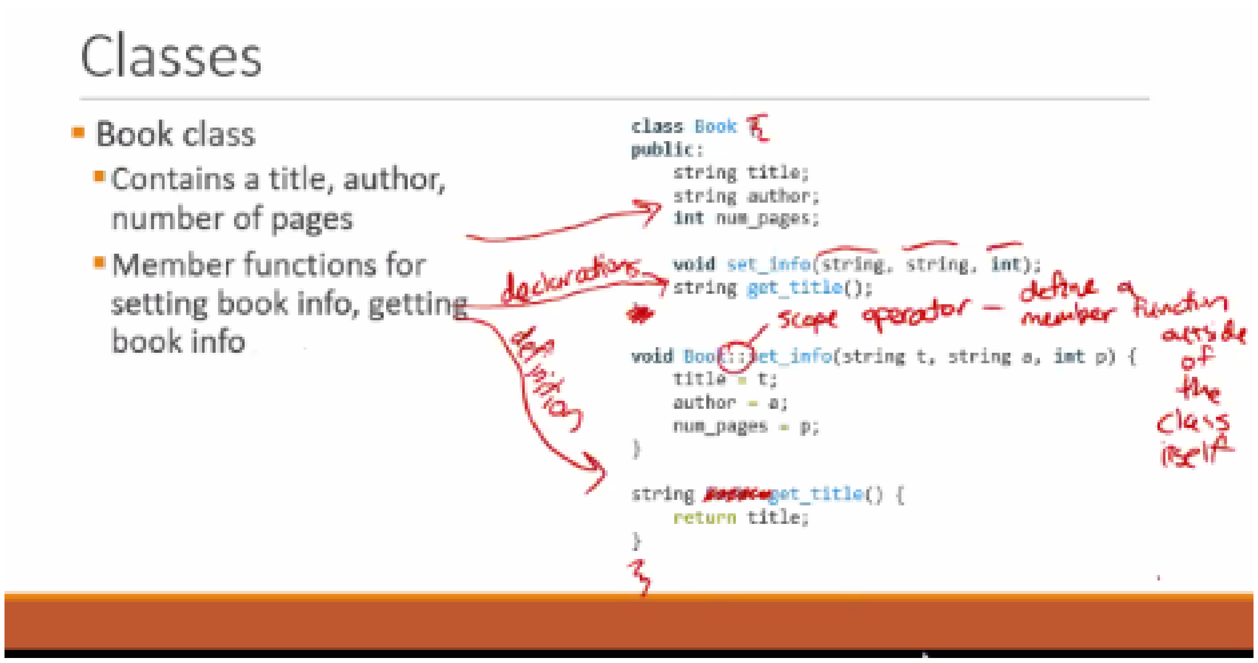
```
Class Book {
```

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);
```



Classes

- Book class
  - Contains a title, author, number of pages
  - Member functions for setting book info, getting book info

```
class Book {
public:
    string title;
    string author;
    int num_pages;

    void set_info(string, string, int);
    string get_title();

void Book::set_info(string t, string a, int p) {
    title = t;
    author = a;
    num_pages = p;
}

string [illegible]get_title() {
    return title;
}
}
```

declarations

definition

scope operator - define a member function outside of the class itself

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
```

CS162 Notes Spring 2021

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

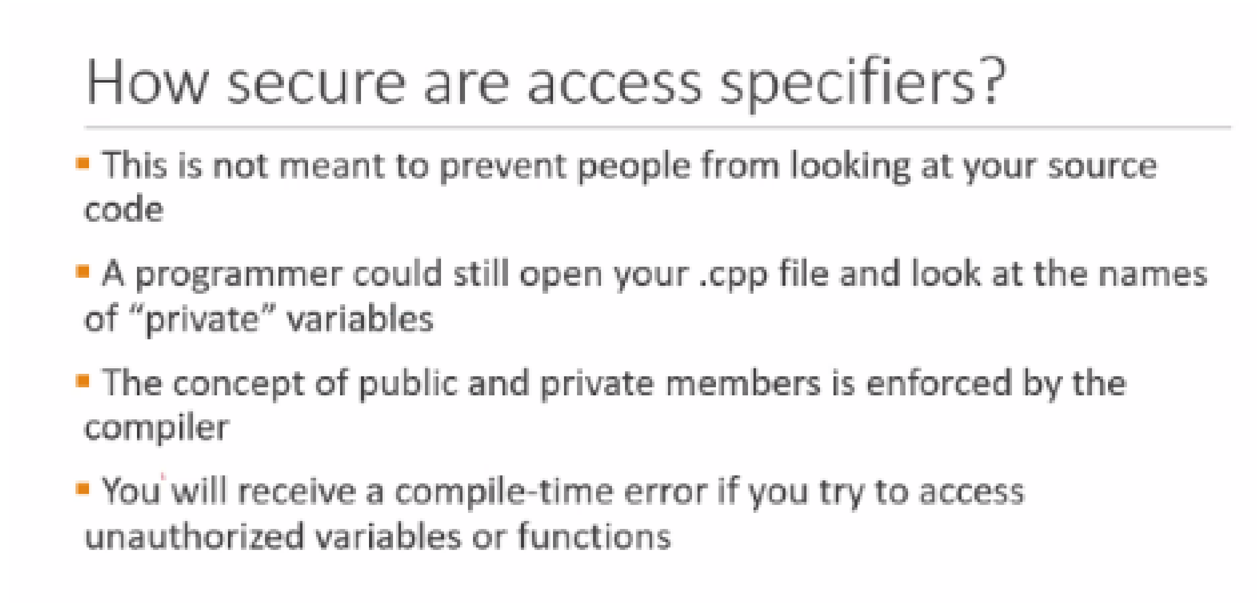
There are functions we use to retrieve or change variable values inside classes. Often called “getters” and “Setters”

```
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.

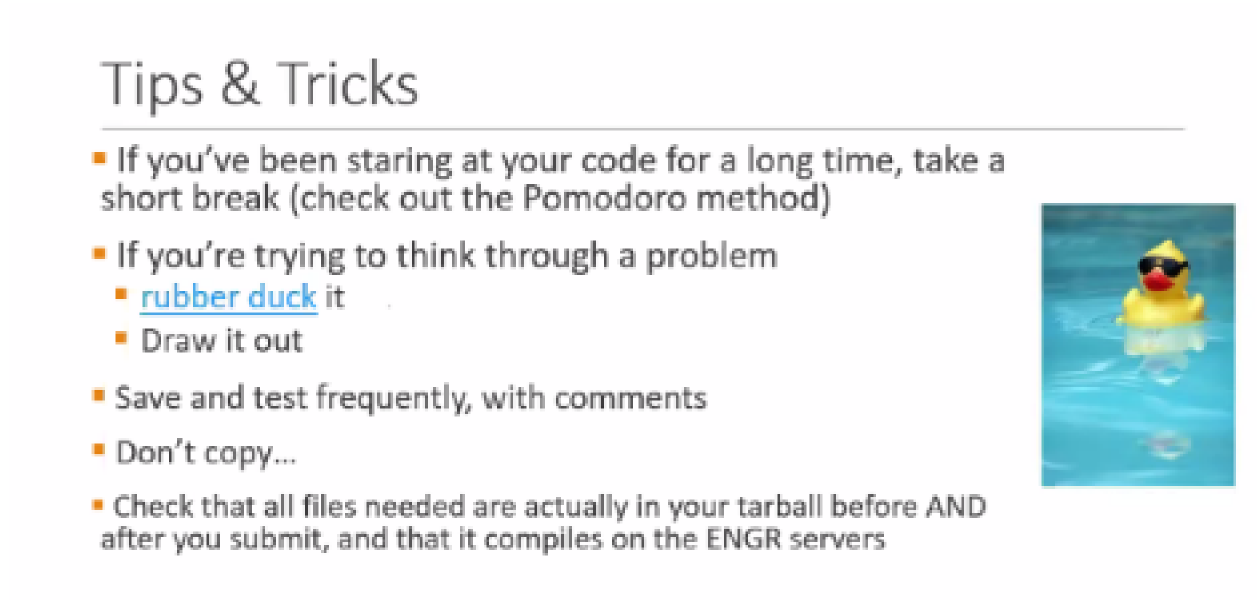
Getters and setters are properly named (formally) accessors and mutators



How secure are access specifiers?

- This is not meant to prevent people from looking at your source code
- A programmer could still open your .cpp file and look at the names of "private" variables
- The concept of public and private members is enforced by the compiler
- You will receive a compile-time error if you try to access unauthorized variables or functions

Ke
Keeeeee
CS162 Notes Spring 2021 **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

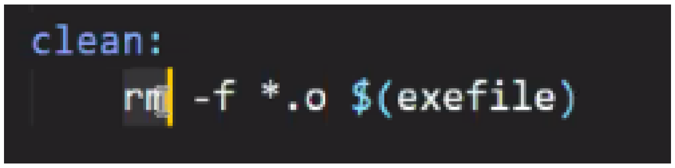
- Point.h - contains class definition and member function prototypes and variables
- Point.cpp - contains the member function definitions ●

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)
```



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

CS162 Notes Spring 2021

This says "delete anything with the extension .o"

We can add a tar command!

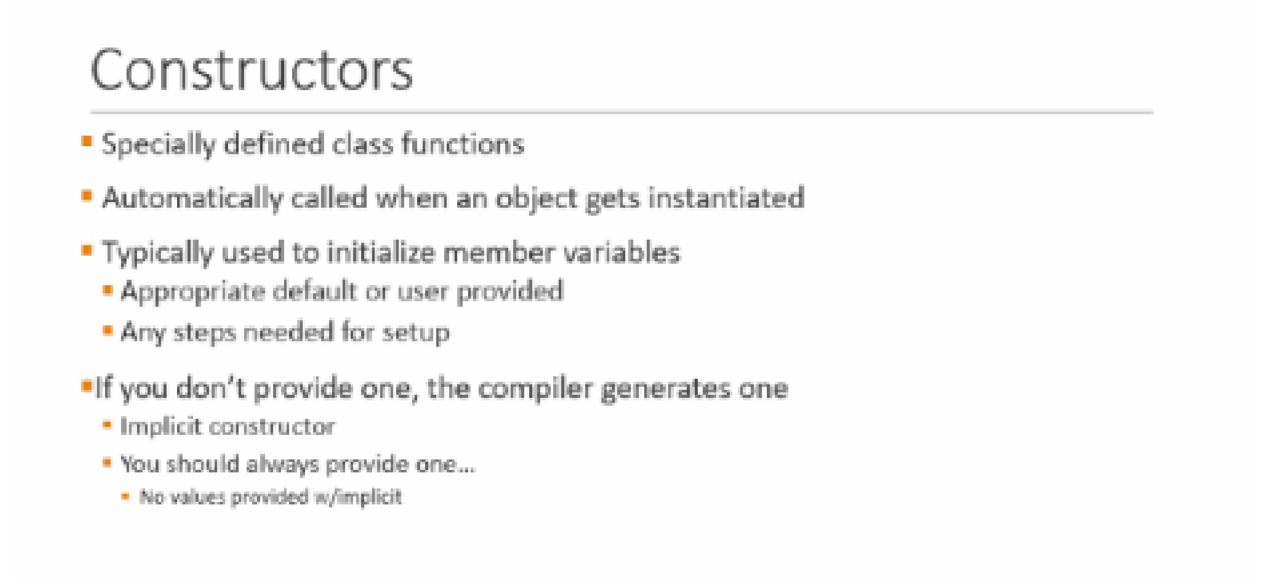
```
tar:
     tar -cvf assign1.tar *.cpp *.h Makefile
```

CS162 Notes Spring 2021 **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

- Specially defined class functions
- Automatically called when an object gets instantiated
- Typically used to initialize member variables
  - Appropriate default or user provided
  - Any steps needed for setup
- If you don't provide one, the compiler generates one
  - Implicit constructor
  - You should always provide one...
    - No values provided w/implicit

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);
```

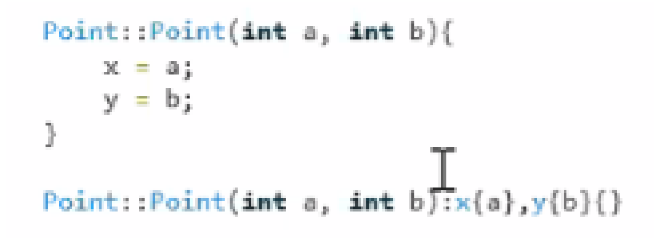
You can't call these constructors with the classic . operator, like Point p2.Point(); < That is not legal

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:

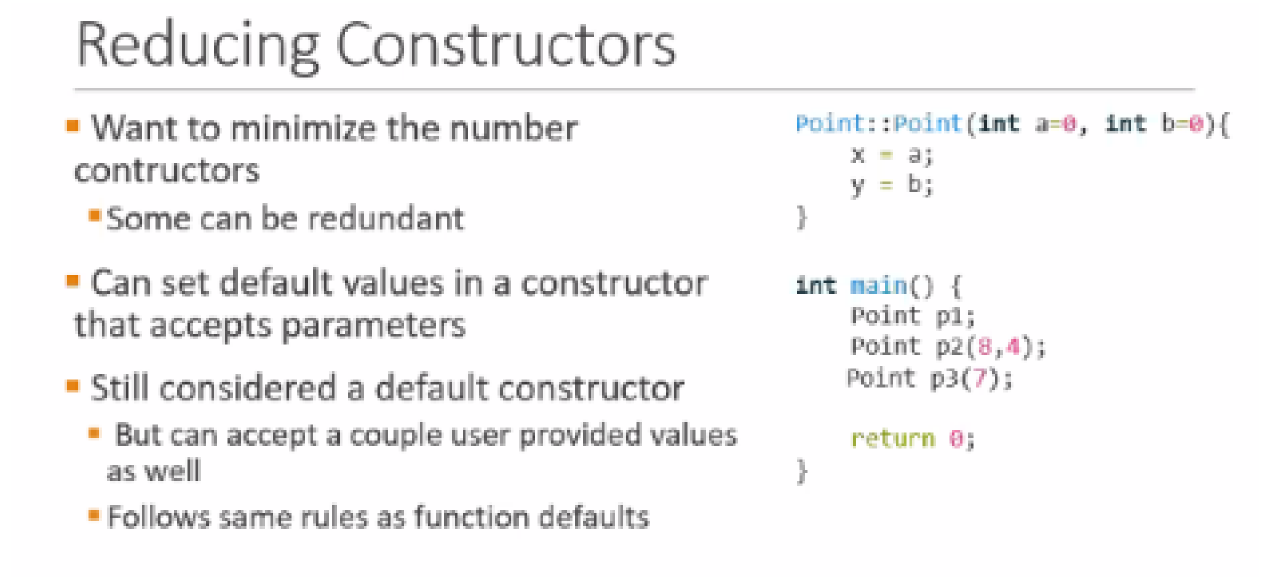
```
Point p3(3, 5, 7);
```



```
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

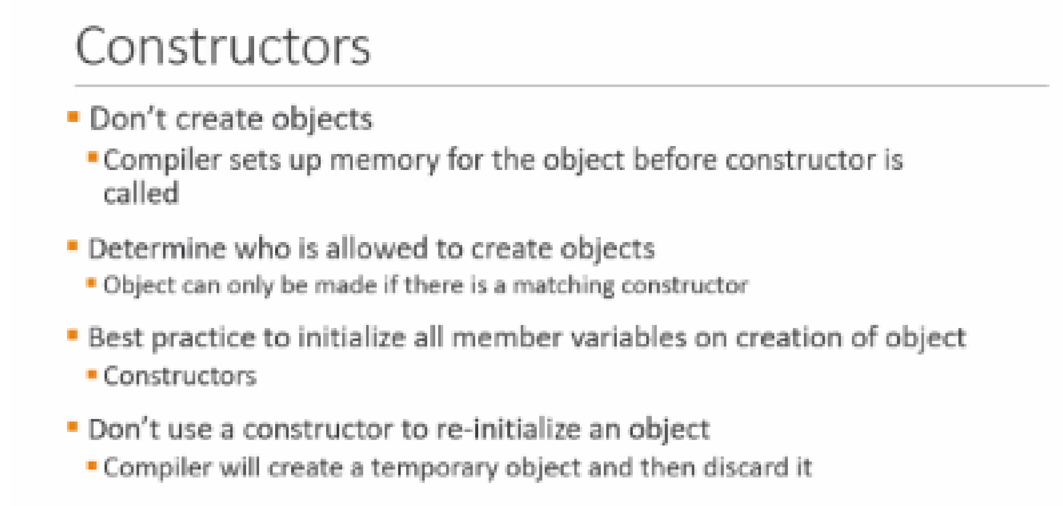
```
Point::Point(int a=0, int b=0){
    x = a;
    y = b;
}

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

CS162 Notes Spring 2021

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.

CS162 Notes Spring 2021 **4/14/21 - Lecture 8 - Const, static,**

destructors

```
#include "point.h"
#include <iostream>

Using namespace std;

Int main(){


     Return 0;
}
```

Last time… Constructors:

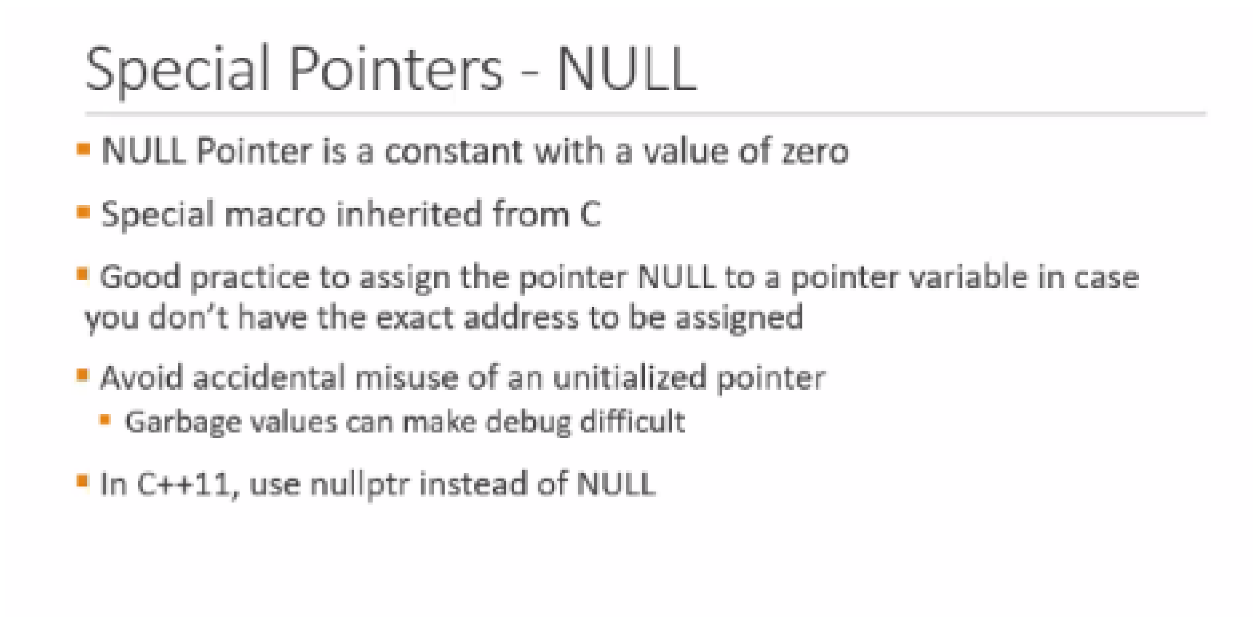
- 3 types:
    - Implicit
    - Default
    - Accepts parameters
- Special class member function that is called automatically and only once when class is creates
- … etc

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).

CS162 Notes Spring 2021



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
  -

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:

CS162 Notes Spring 2021 **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"

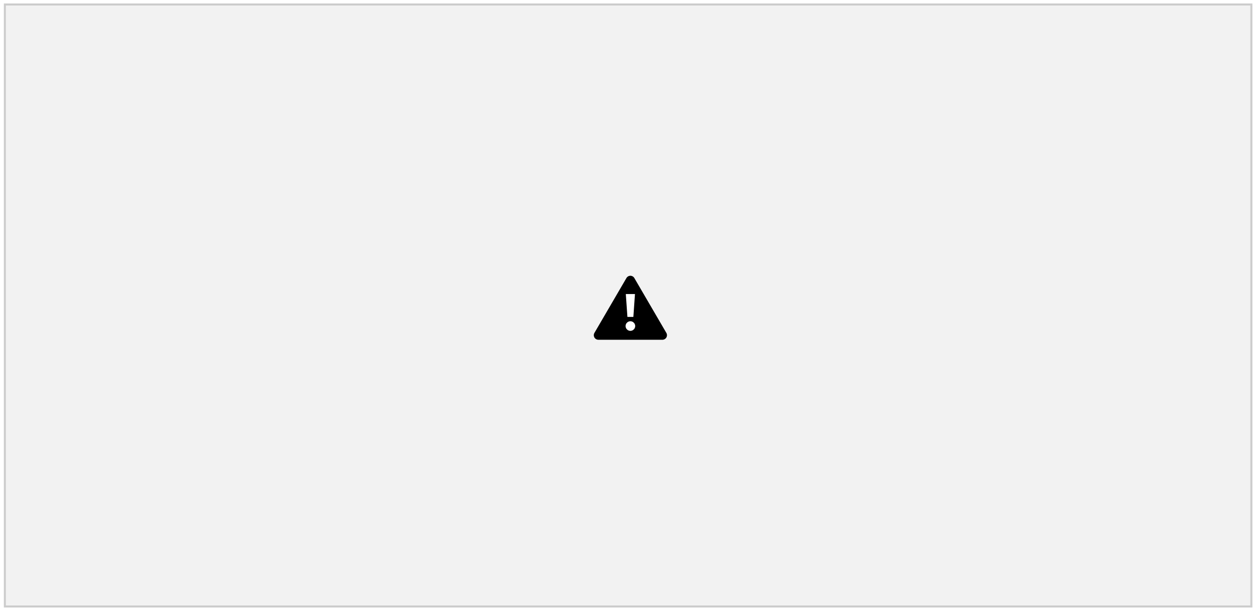
int Point::count = 0; //The static keyword *doesn't* show up here!

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

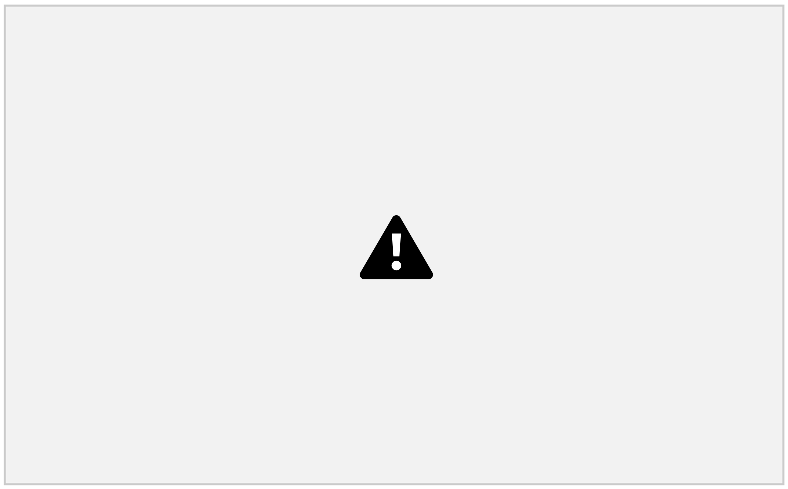

CS162 Notes Spring 2021

~~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

CS162 Notes Spring 2021 **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

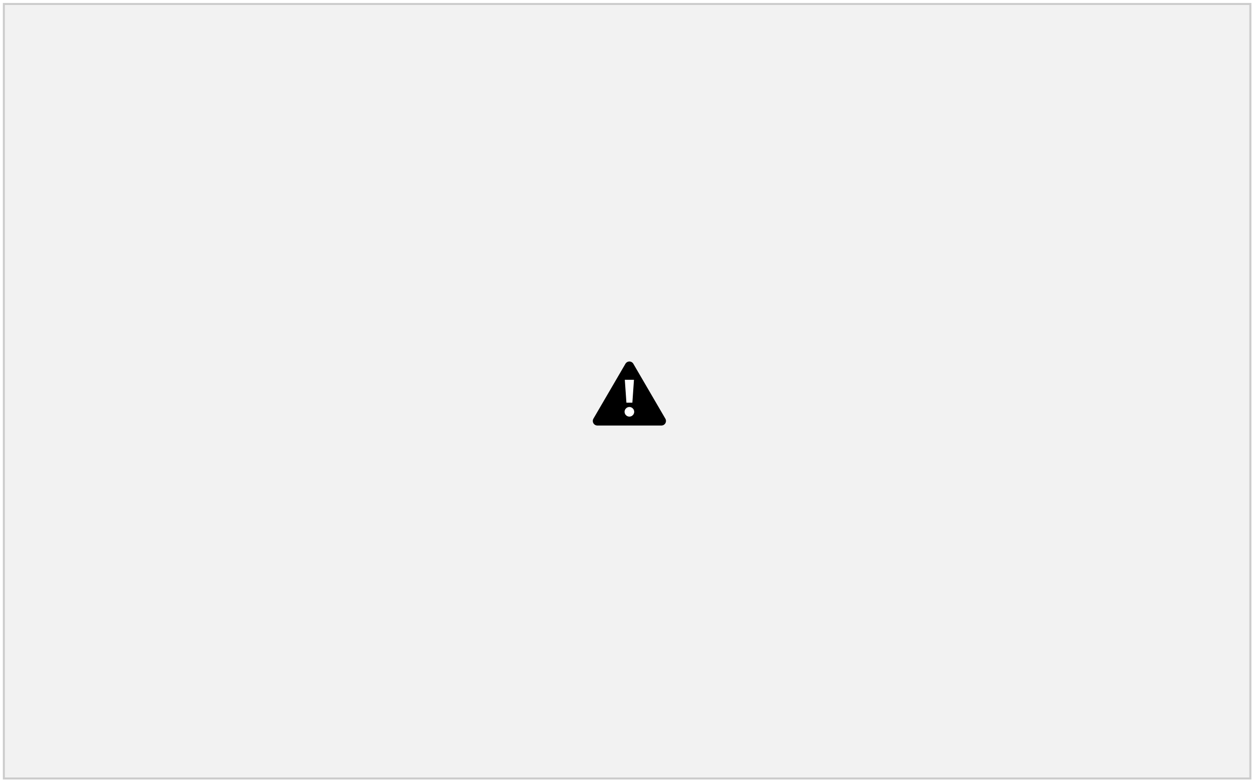
When are they useful?

```
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:



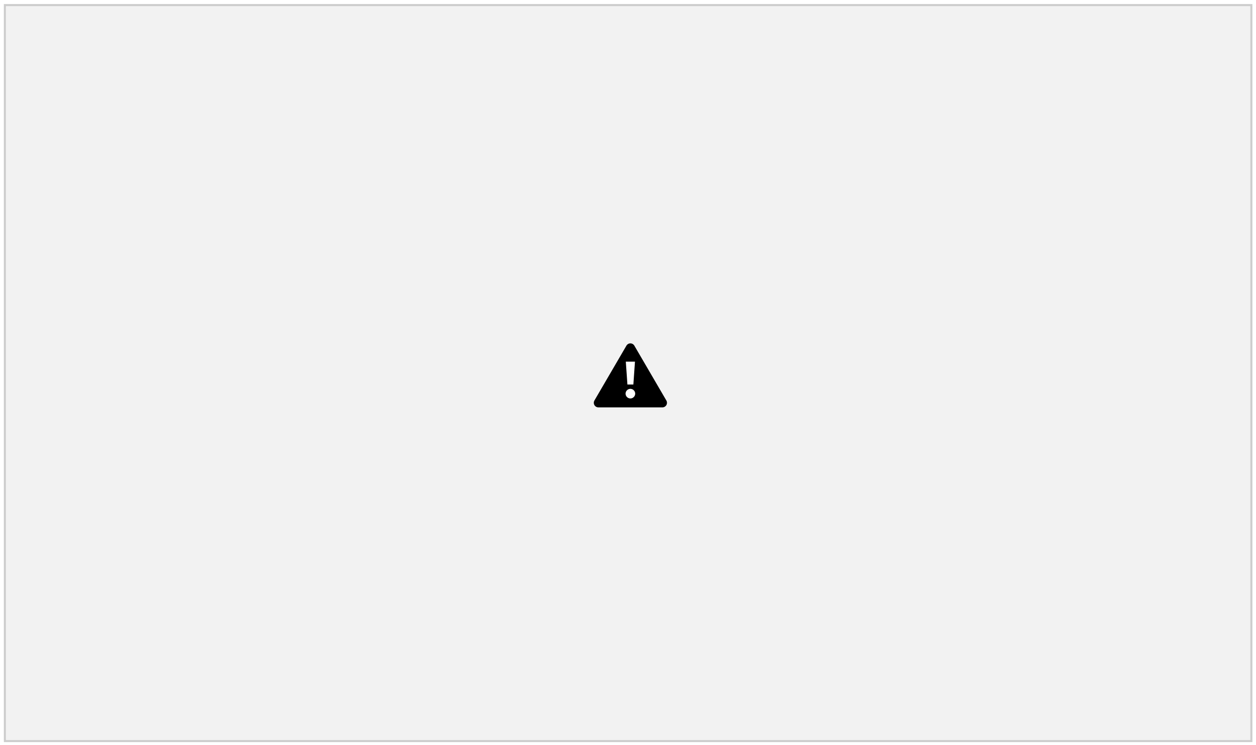
- Same name as class
- **REQUIRED** to have a parameter that is same type as the class.
- Still doesn’t return anything.
- Normally the parameter is a const
- There is one that is explicitly there -- a default copy constructor. Those are fine, AS LONG AS we aren’t dealing with dynamic memory.

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:



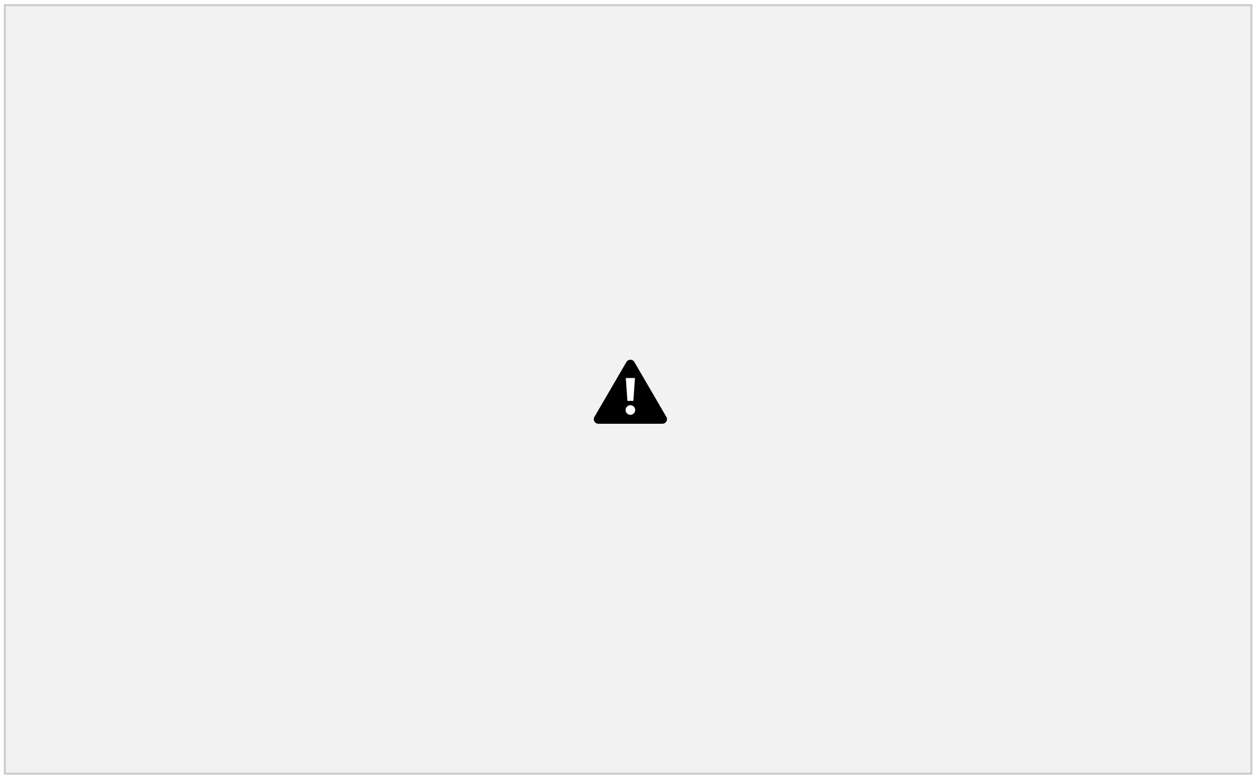
- Very specific syntax to be recognized as an overload.
- Needs to return a reference to the object
-

```
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.

CS162 Notes Spring 2021 **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
CS162 Notes Spring 2021 **4/23/21 - Lecture 12 - Relationships**

between objects

Composition and inheritance:

Composition:

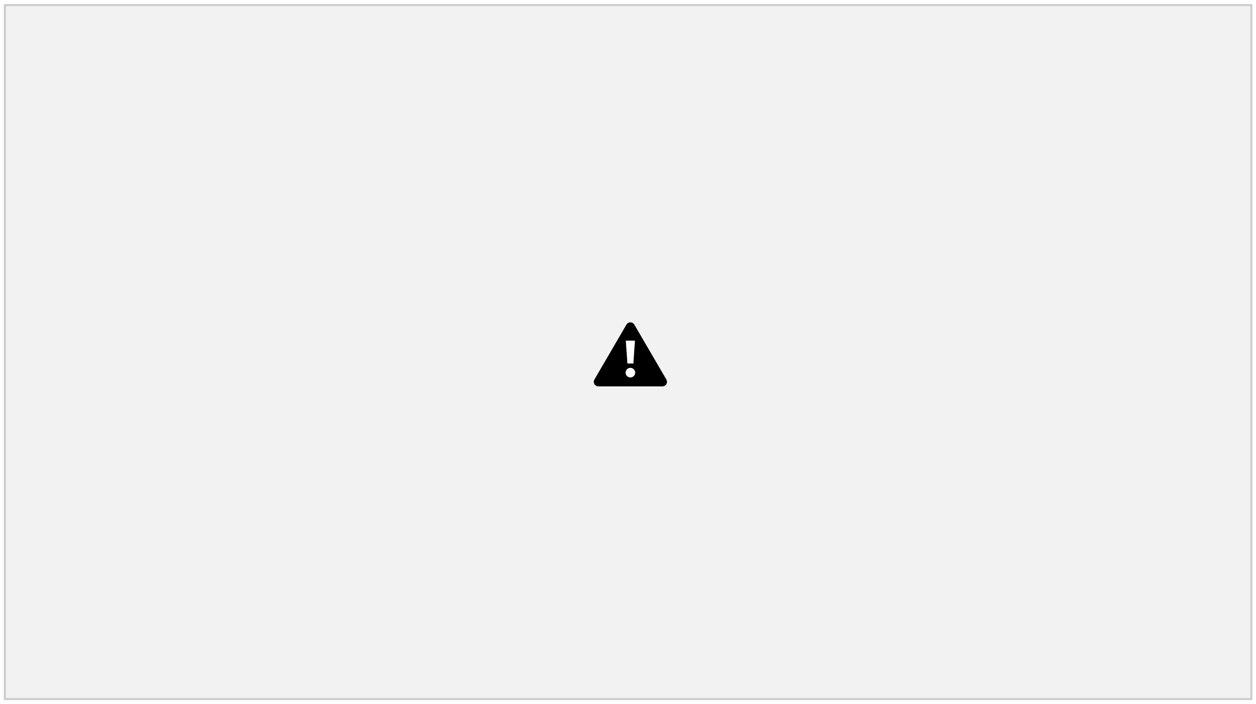
- "Part-of" relationship
- Complex object (class) is built from one or more simpler objects
- Part's existence managed by object (class)
    - Responsible for creation/destruction of parts
    - Parts' lifetime is bound to the lifetime of its object
- Part can only belong to one object at a time
    - 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

CS162 Notes Spring 2021



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

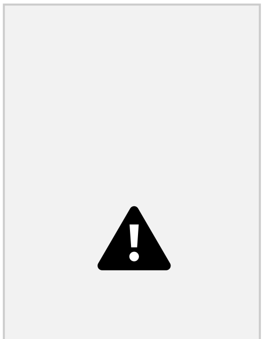
CS162 Notes Spring 2021

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)

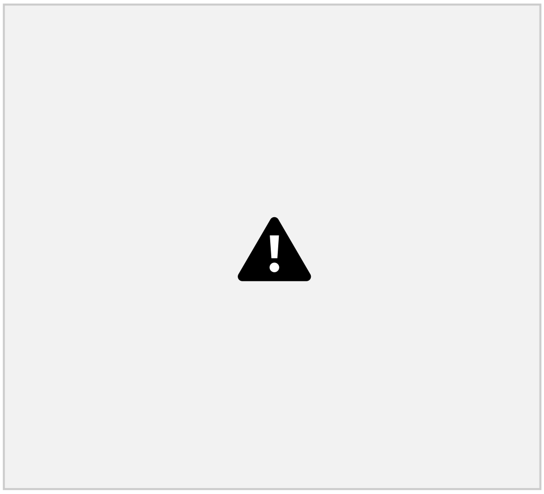
Inheritance:

- Uses hierarchies to show how objects are



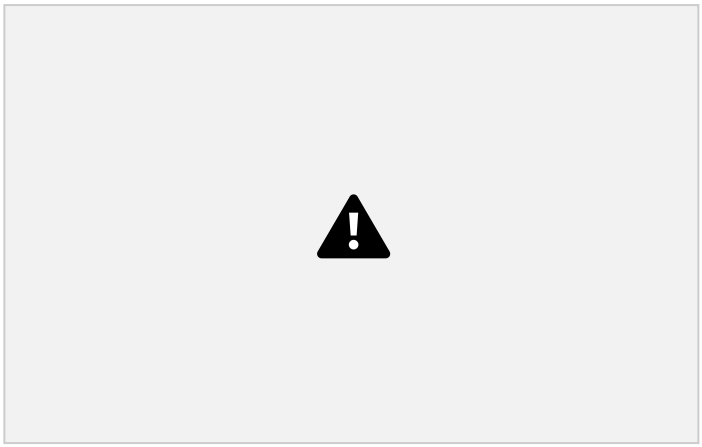
related

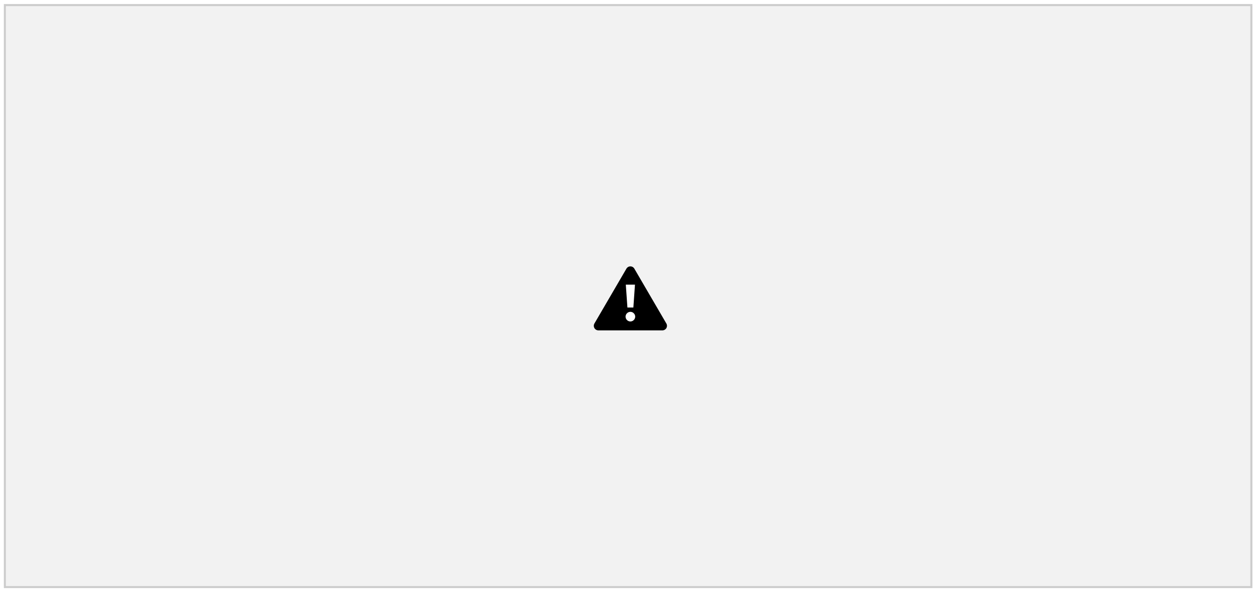
- Two common types:
    - Progression
        - Over time
    - Categorization:
        - From general to specific



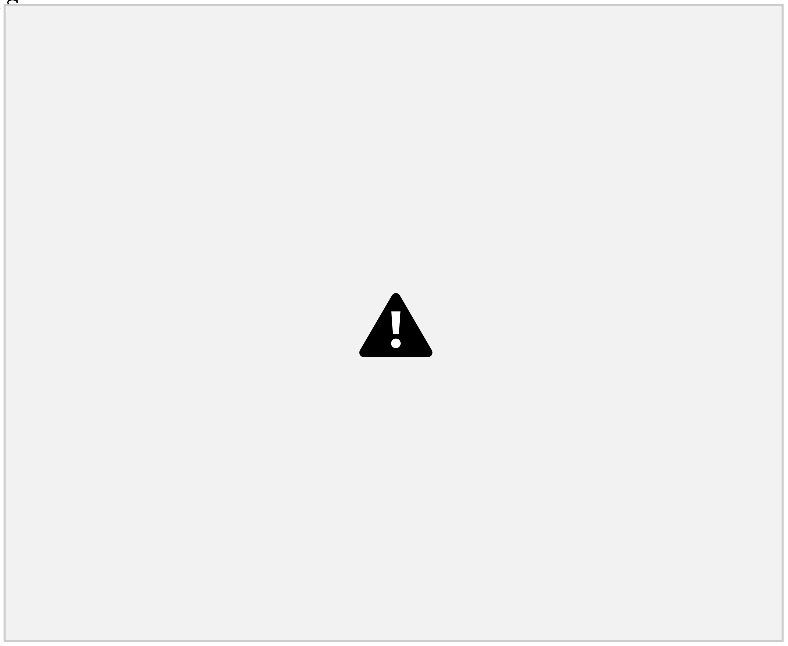
One of the most common examples is when we’re dealing with geometries and shapes:

CS162 Notes Spring 2021

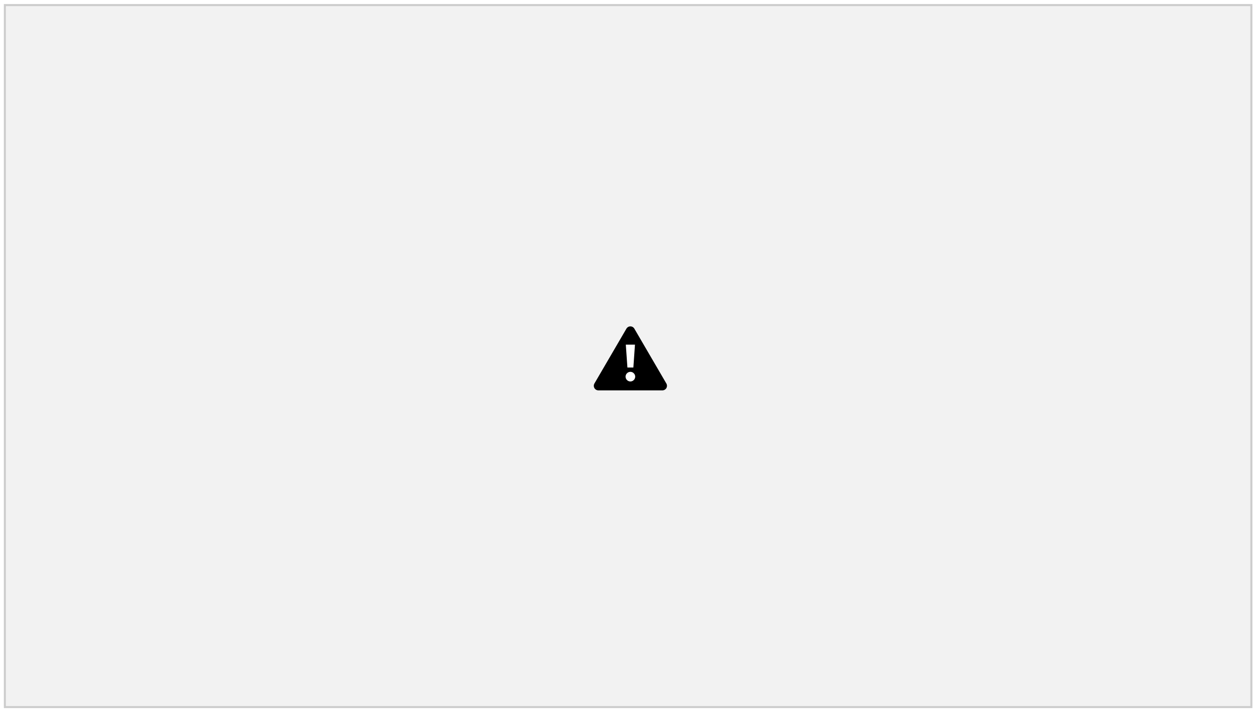




Students and teachers have a lot of things in common. If we used inheritance for this: instead of having to write all the shared things twice, we could make a person class to hold the command things:



Inheritance is not limited to
a single level:



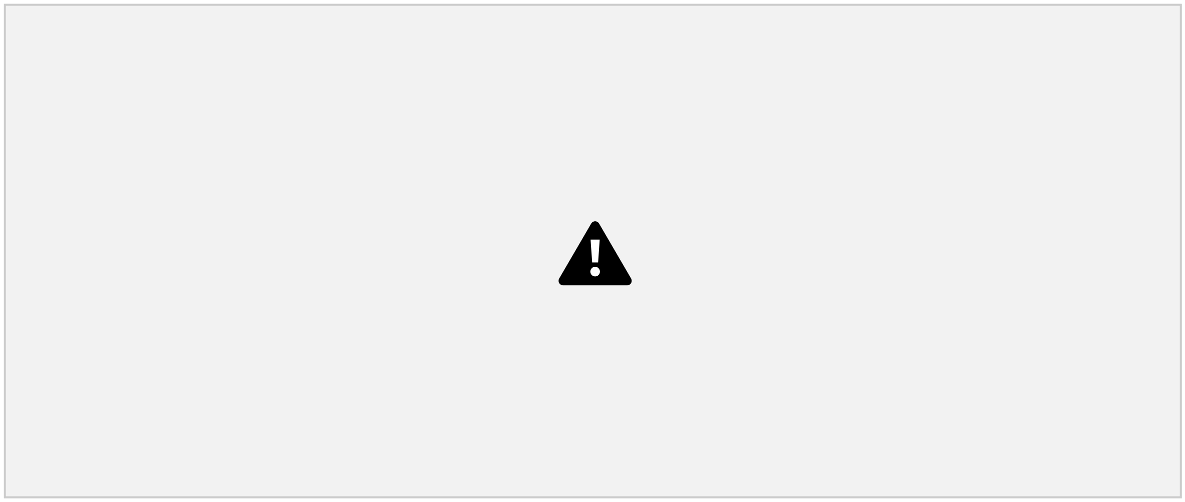
Inheritance is a **big** thing for program 3.

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.

CS162 Notes Spring 2021 **4/23/21 - Lecture 13 - Implementing**

inheritance

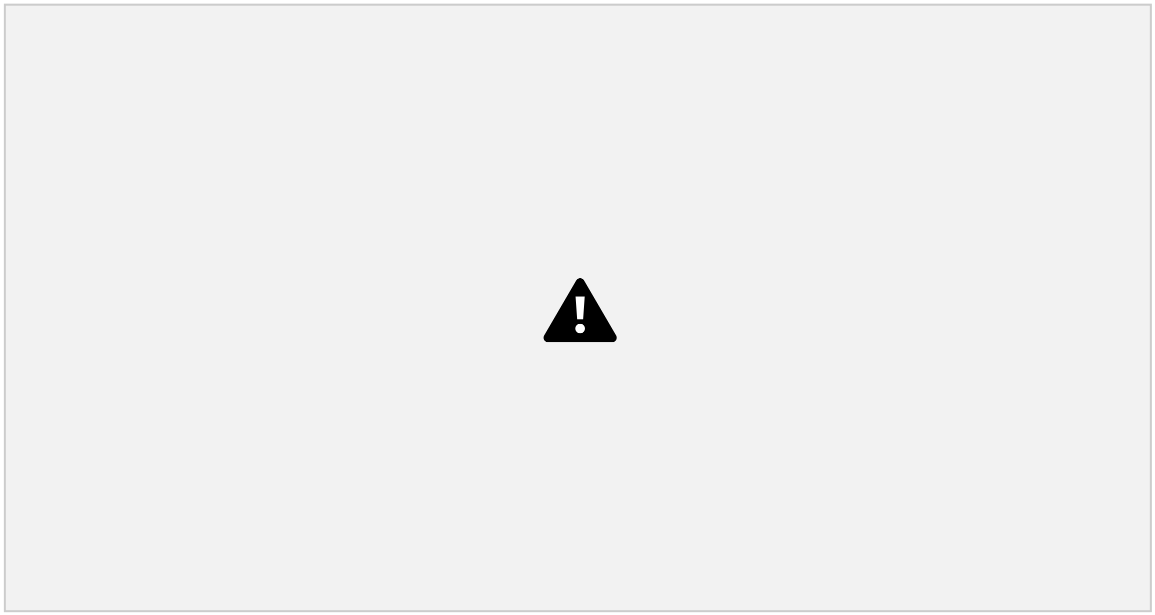


```
class Base {
Public:
      Int pub;
      Int pri;
```

```
     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

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.

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:

DESTRUCTORS:

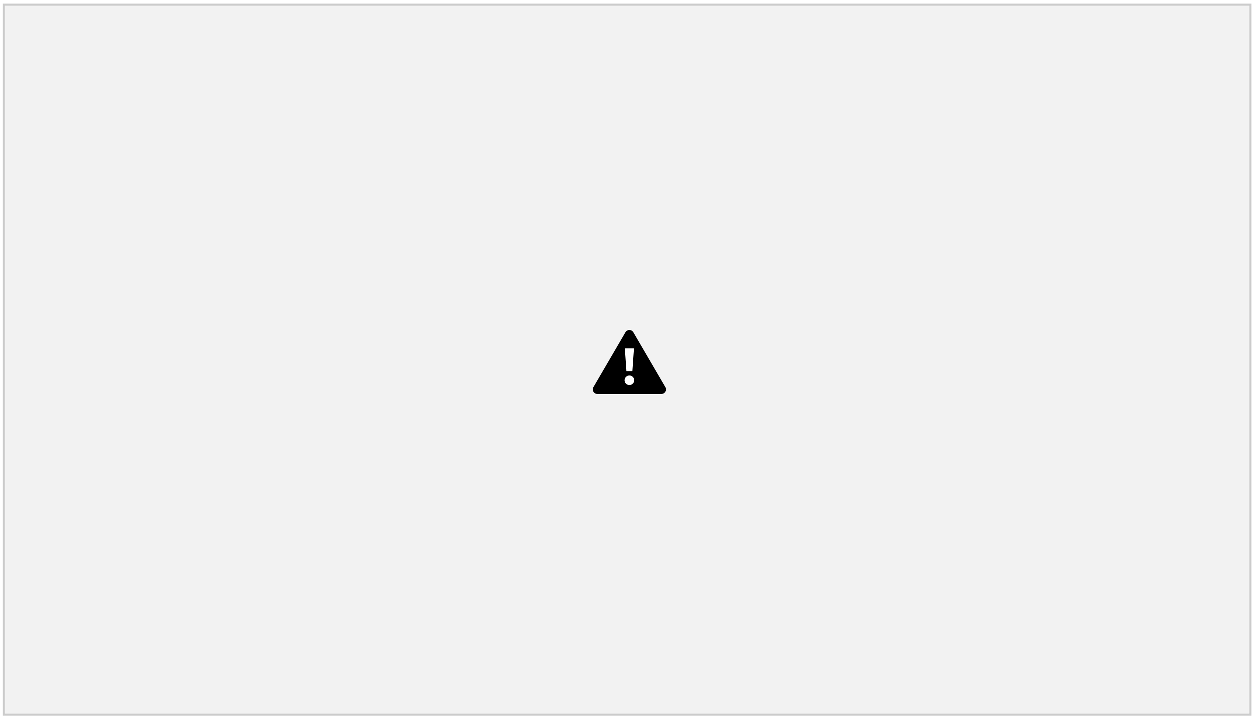
So it calls the derived destructor first, then the class destructor. (as part of destroying just the derived destructor).

So children destroyed first, then parents.

CS162 Notes Spring 2021 **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.

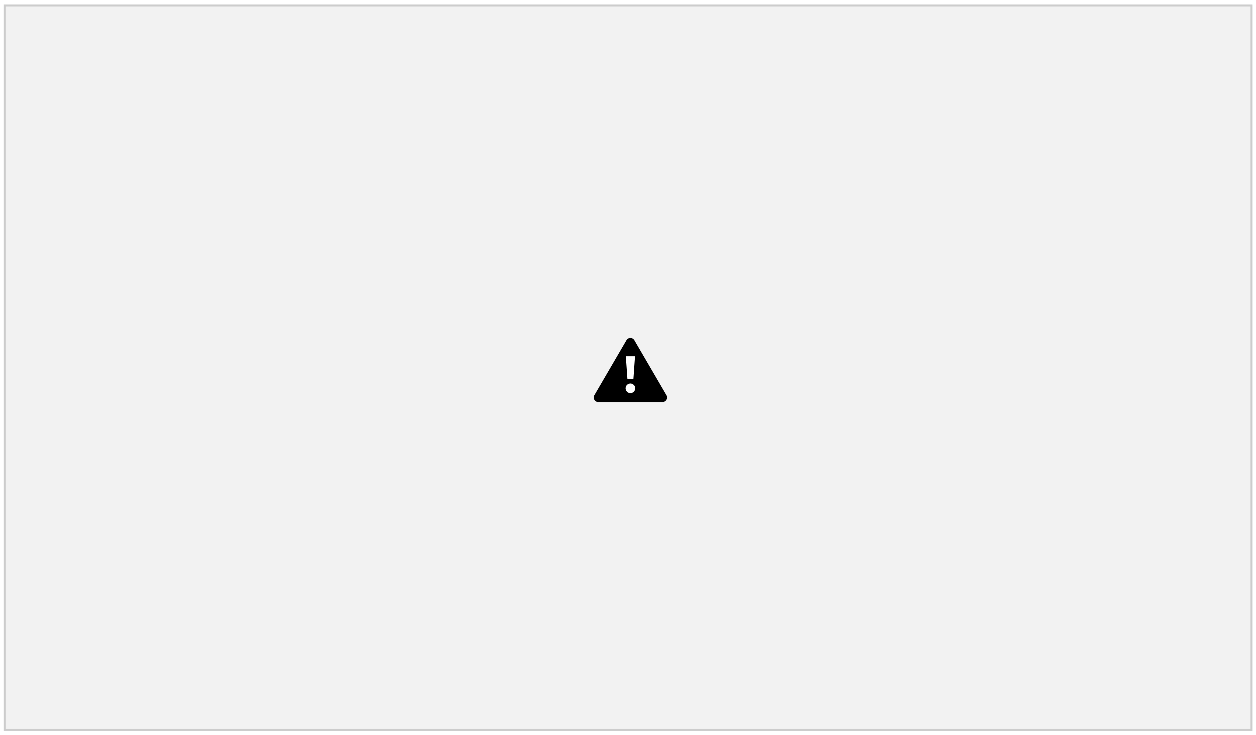
However, friendship is a one-way street. If the Line class is a friend of the point class, this is declared in both classes -- but the Point class is not a friend of the line class.

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.
CS162 Notes Spring 2021

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.

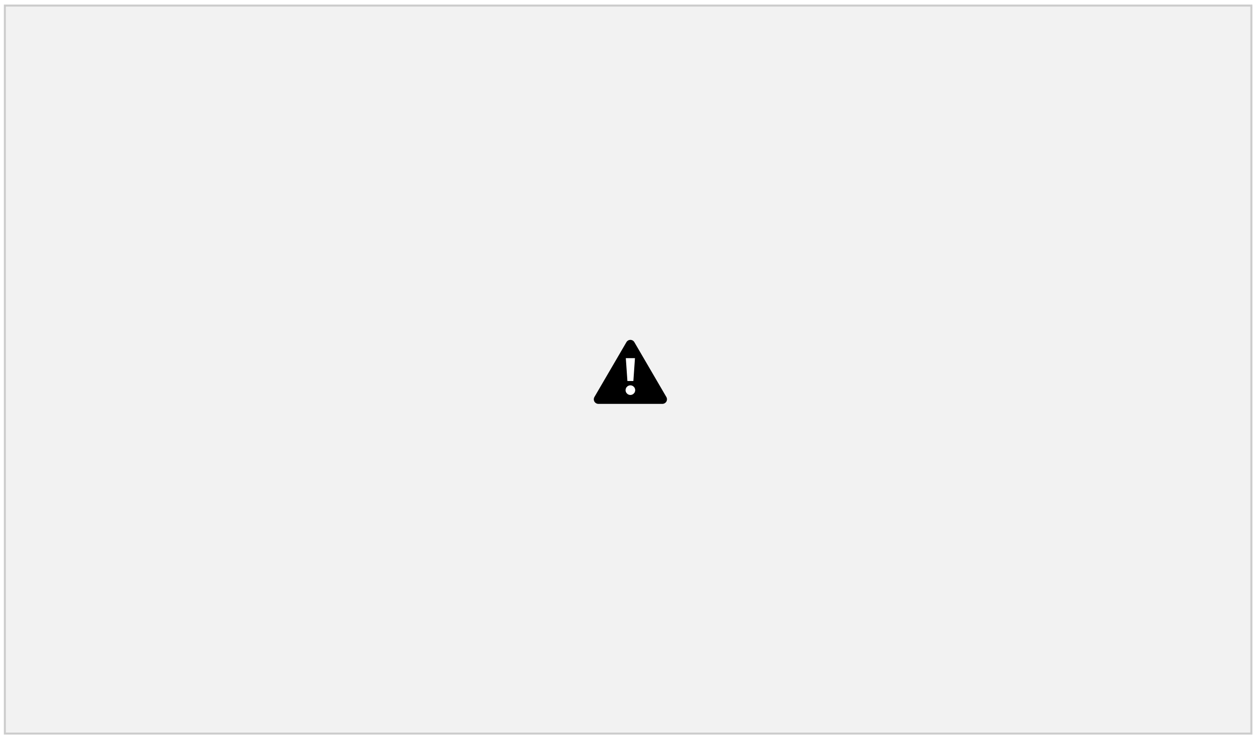
```
Point& operator+=(int x); //Returns type point, takes type int.
```

Here’s what’s in the += operator originally:
```
operator+=(const lh, const rh);
```

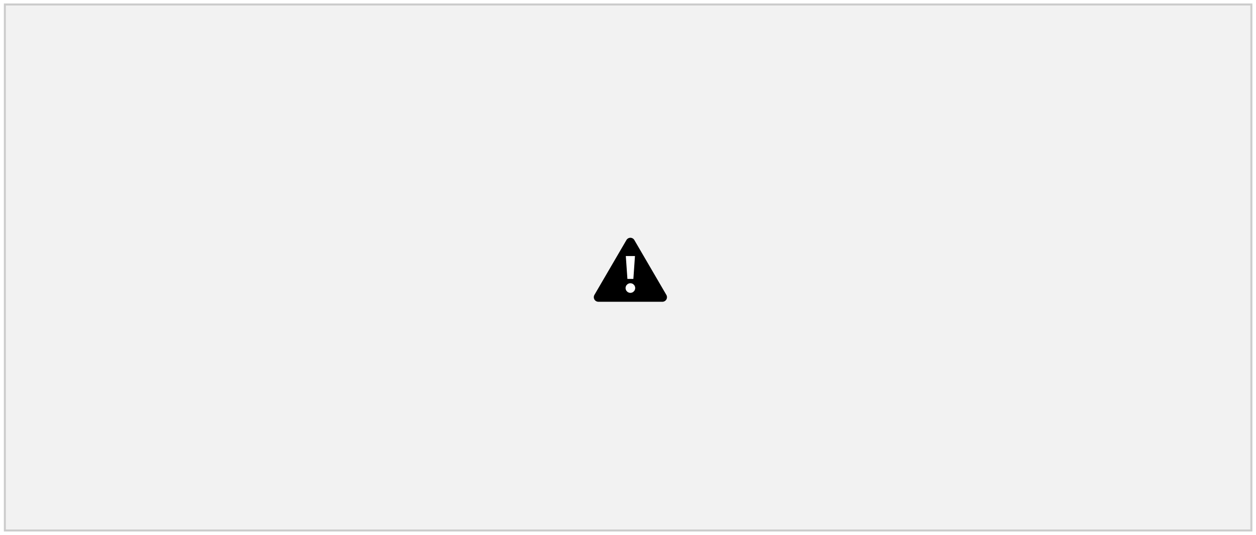
So let’s write out what this function would do.

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

So yu can essentially create a bunch of shortcuts for yourself.
CS162 Notes Spring 2021



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;
}
```

CS162 Notes Spring 2021

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.

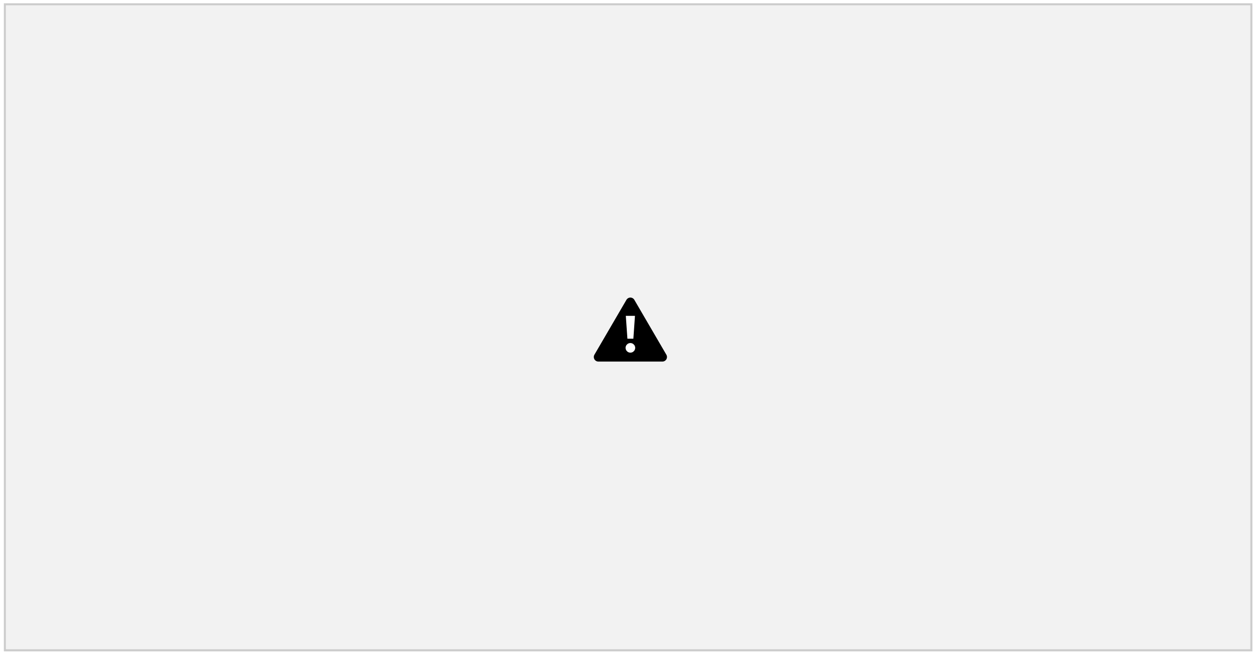
CS162 Notes Spring 2021 **5/3/21 - Lecture 15 -**

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.

CS162 Notes Spring 2021 **5/3/21 - Lecture 16 - Polymorphism**



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.
CS162 Notes Spring 2021

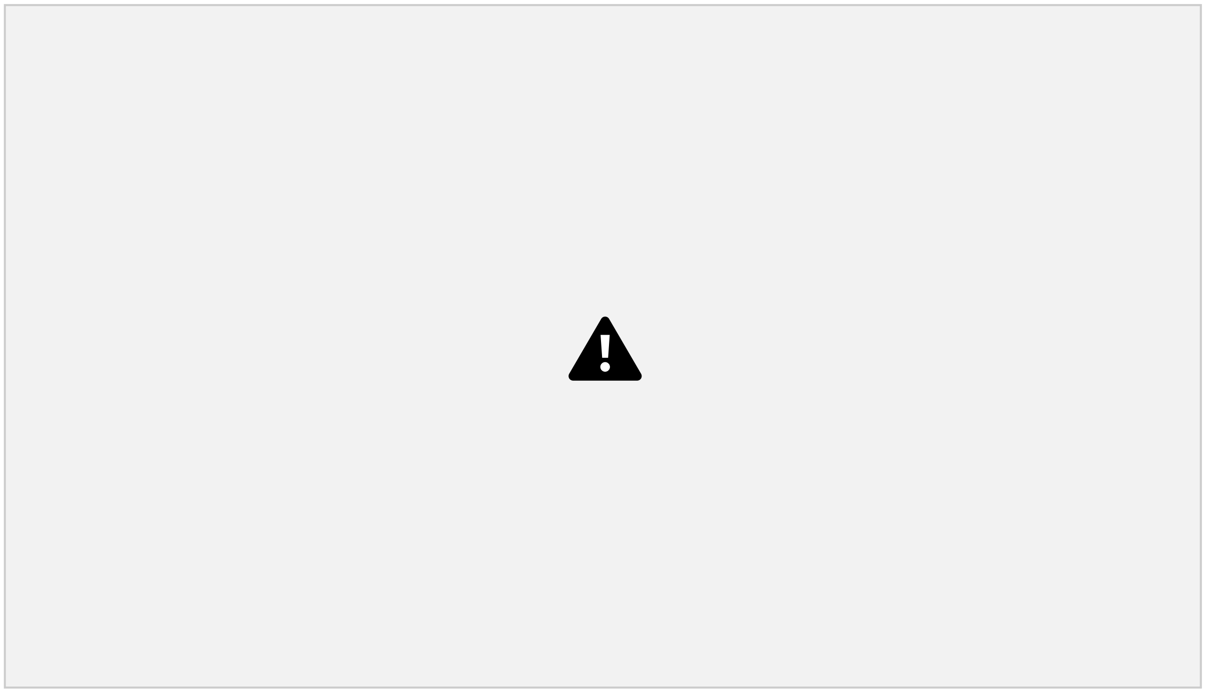
It calls the animal-general get_name -- it cannot see the monkey specific stuff. So it will print out whatever name is stored in the parent class.

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.

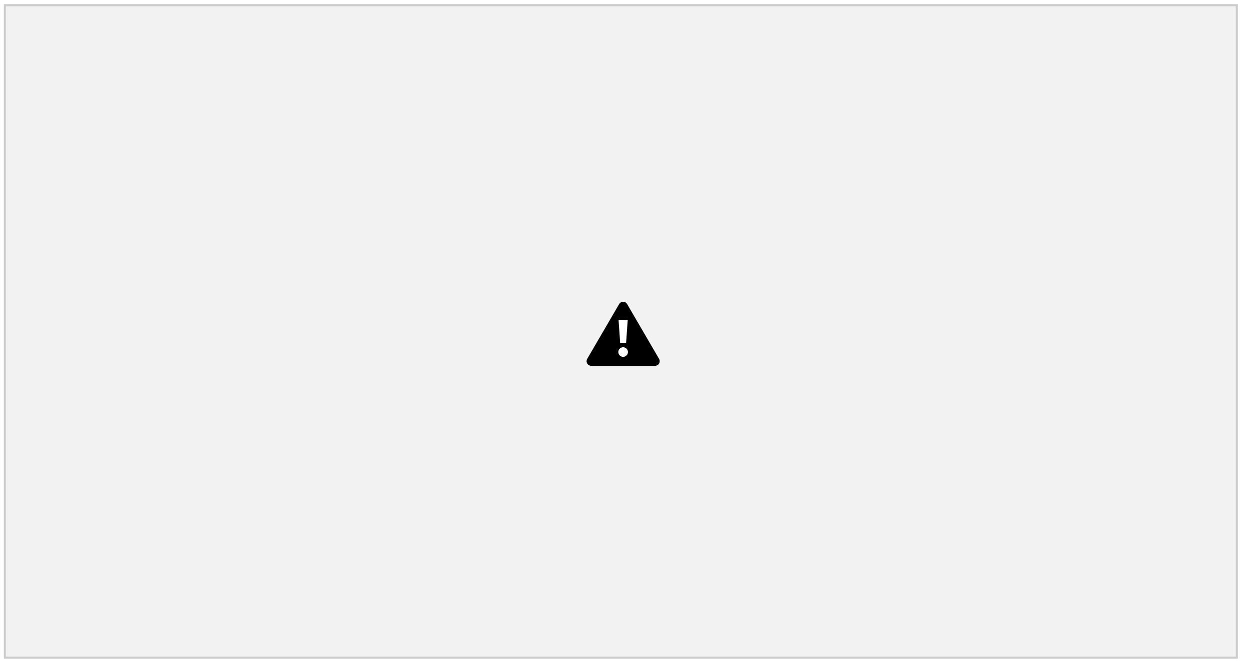
Basically, useful cause we can pass in monkeys or bears or tigers or … to a function expecting a generic animal.

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

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

CS162 Notes Spring 2021

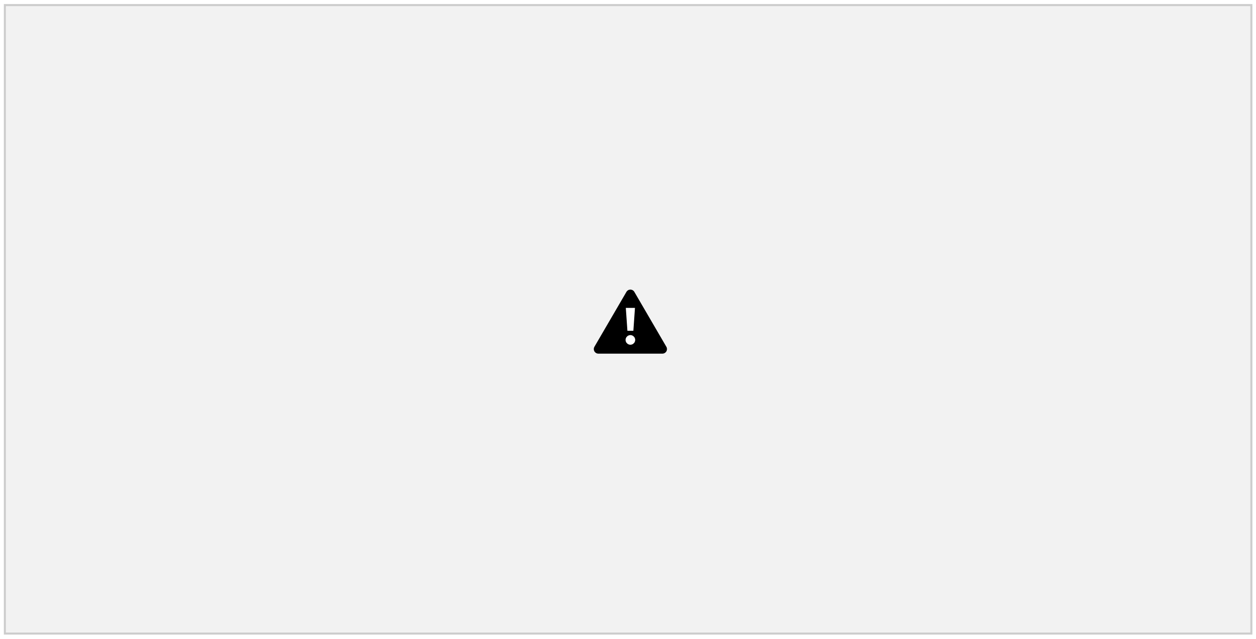


NOW WE’RE GETTING INTO RUNTIME POLYMORPHISM (also called Virtual Functions)



Useful in relation to the previous example somehow...

CS162 Notes Spring 2021

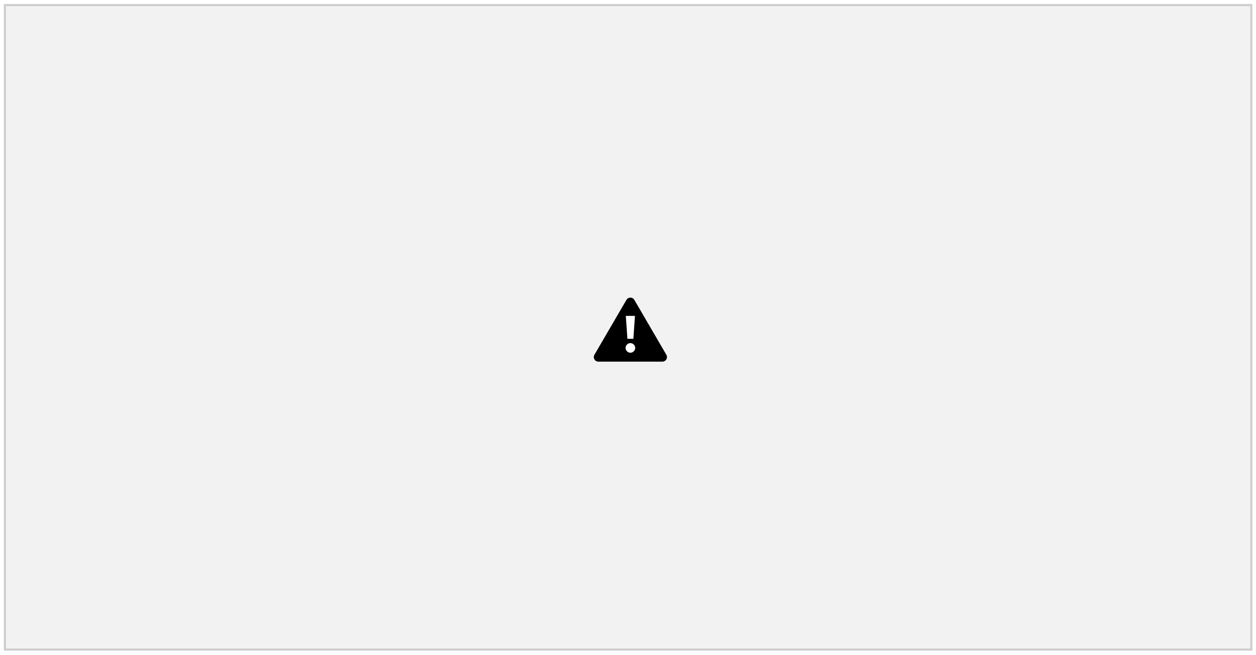


So this is useful when every single child will need to have a unique version of a function, and we don’t want to write that function for the base class.

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.
CS162 Notes Spring 2021



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**

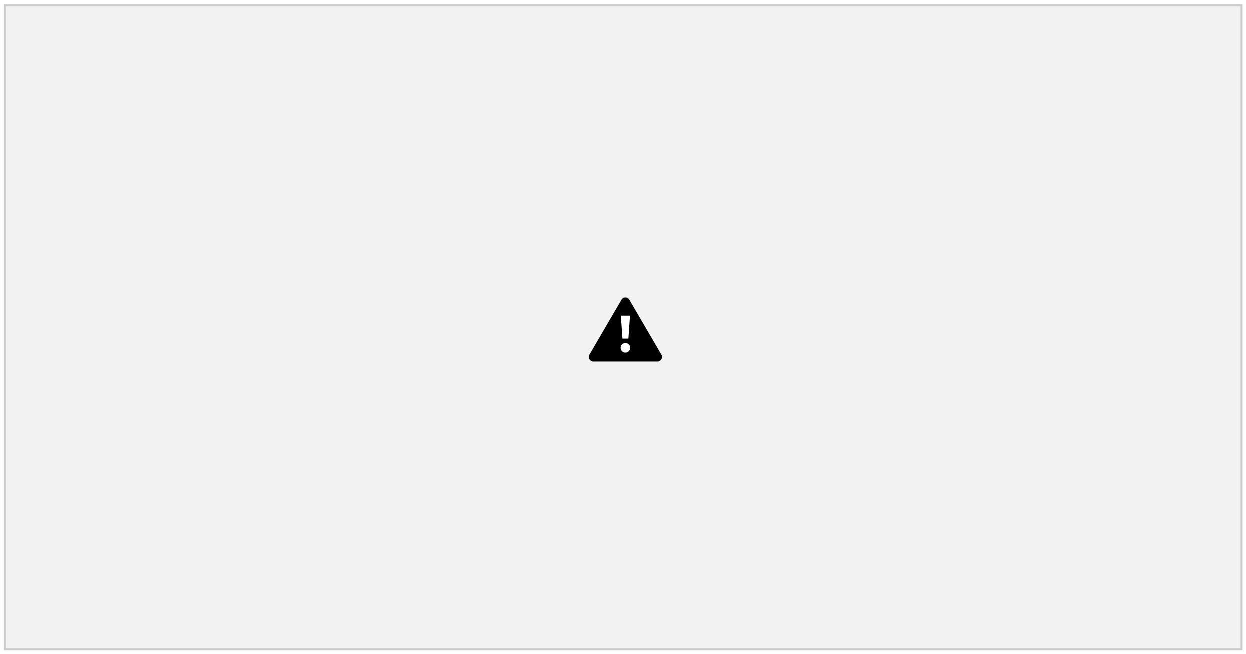
CS162 Notes Spring 2021 **5/7/21 - Lecture 17 - Polymorphism**

continued

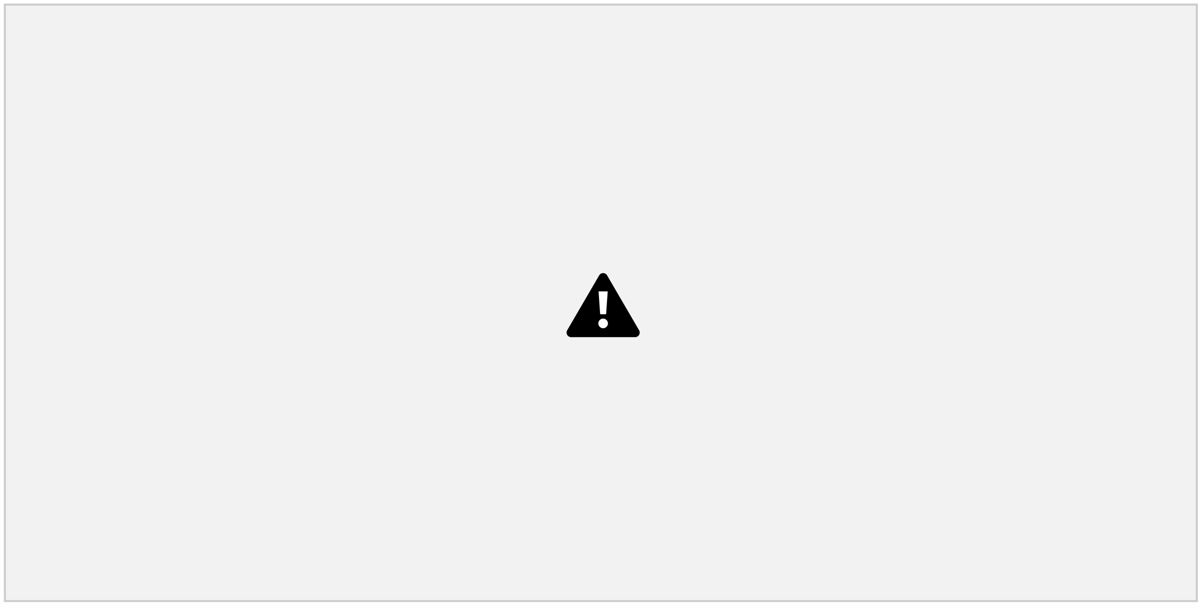
Specifiers from last time:

- 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.
CS162 Notes Spring 2021



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.

Some arguments about this -- but for the purpose of this class we will say they are not inherited. (Hint hint for exams).
CS162 Notes Spring 2021

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

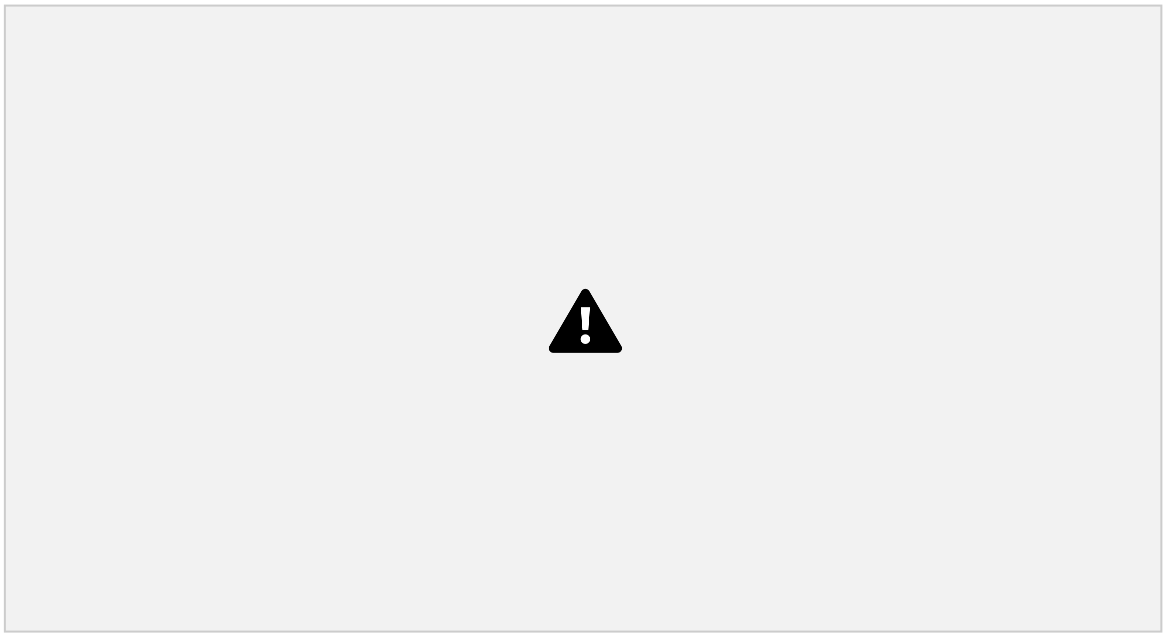
Polymorphism (runtime) is deciding which function to use during runtime -- so it cannot bind when compiling.

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?
CS162 Notes Spring 2021



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:

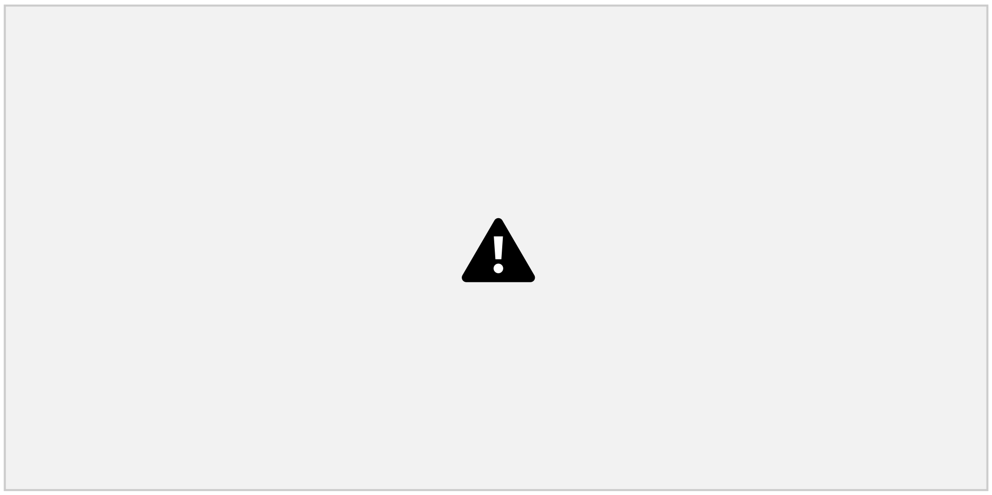
```
Animal *aptr2 = &rso;
```

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:

CS162 Notes Spring 2021



```
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:

CS162 Notes Spring 2021

CS162 Notes Spring 2021 **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

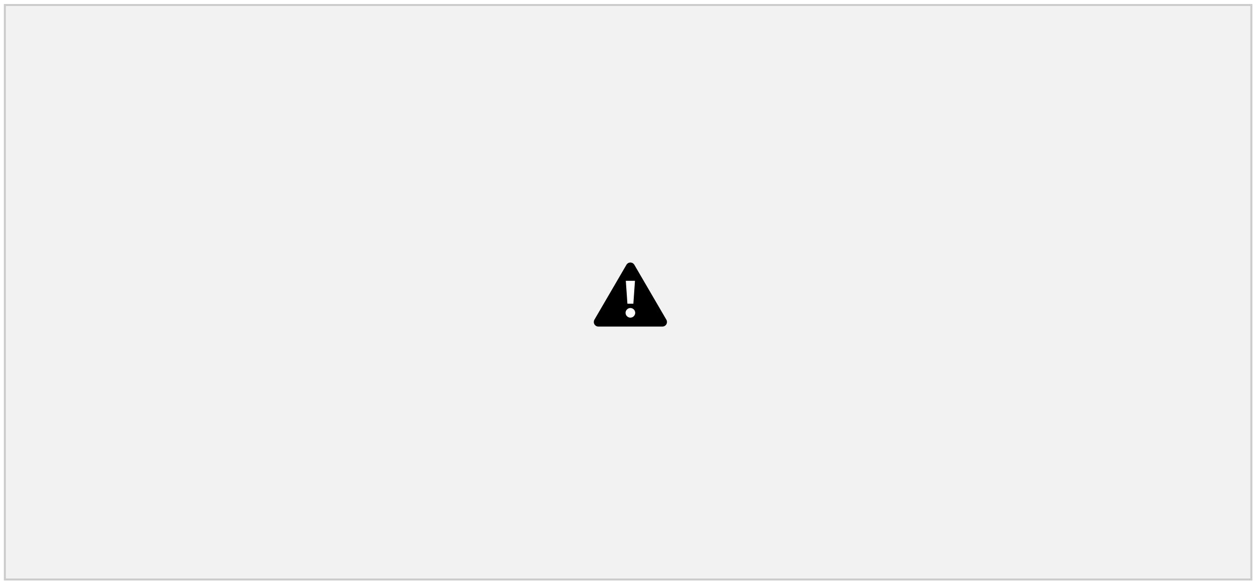
function finishes
  - ○ Check user input to make sure it matches what we want them to input

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:



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

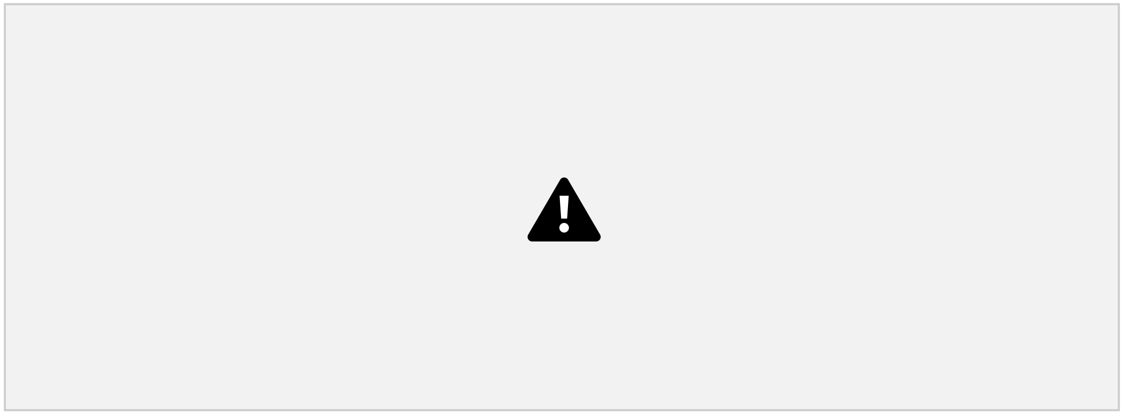
- Exception handling lets use decouple error handling from control flow of the code
  - Meant for dealing with edge cases and stuff

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.

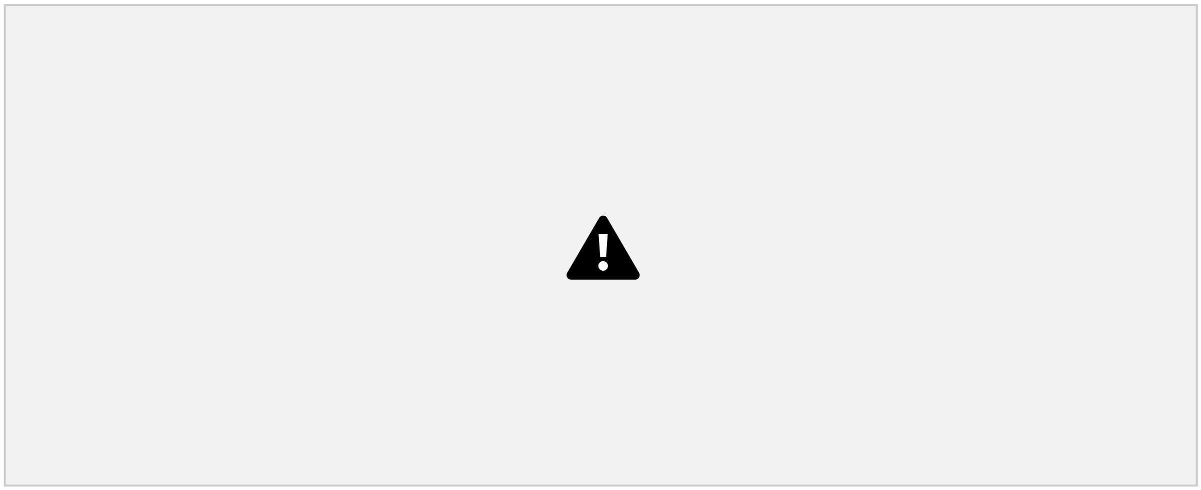
Inside main:



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.

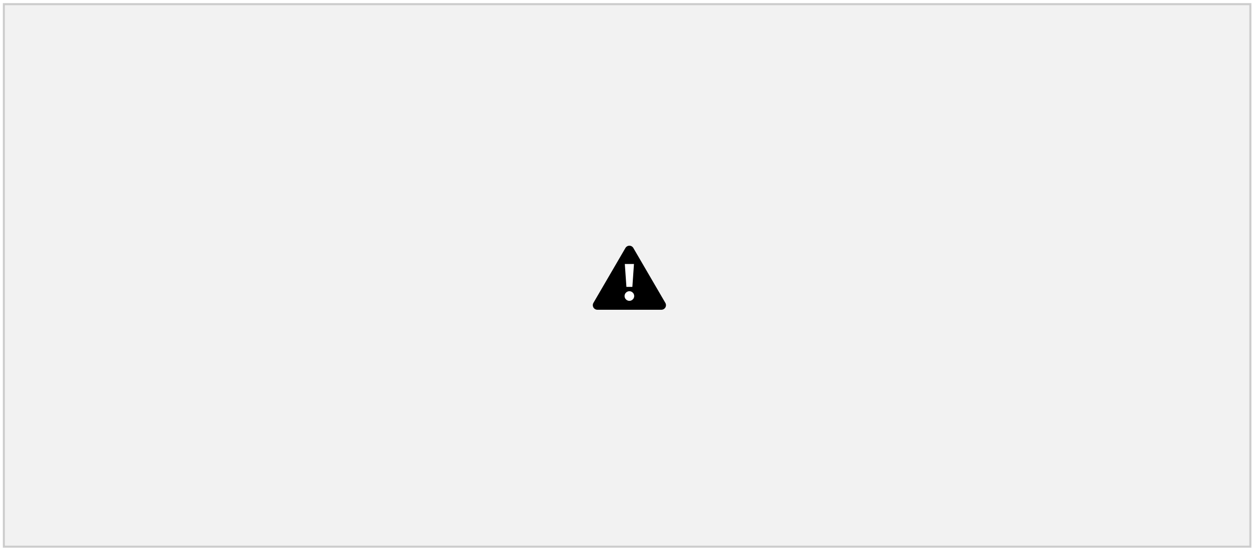
CS162 Notes Spring 2021



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.



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!"
}
```

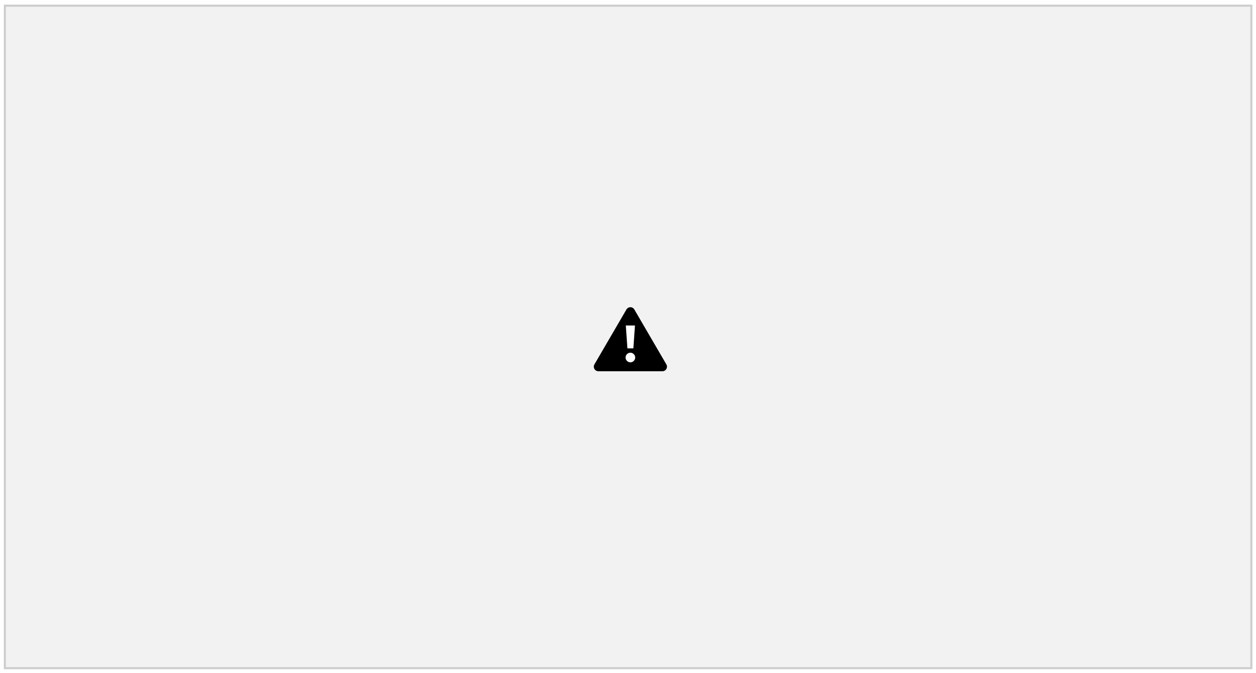
There is a keyword for functions for dealing with exceptions…

“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

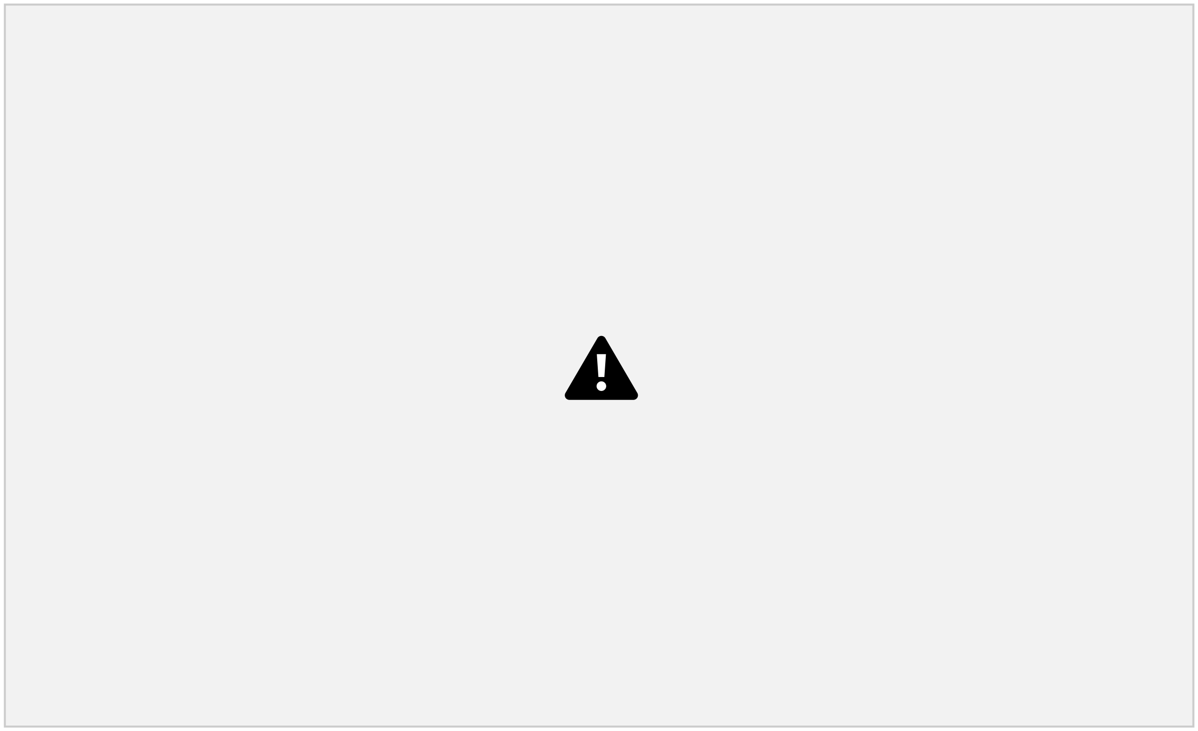


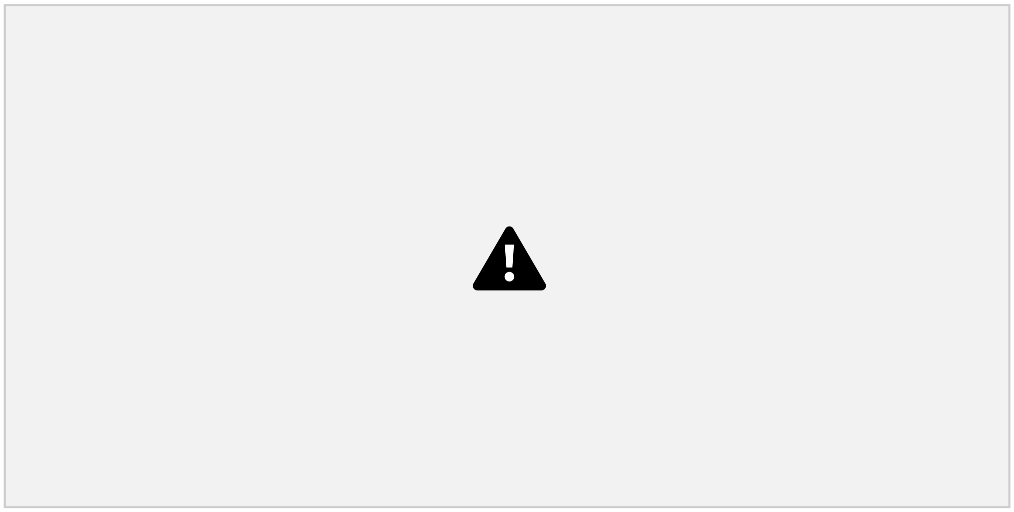
‘Cause I wasn’t paying enough attention to type this out right away...

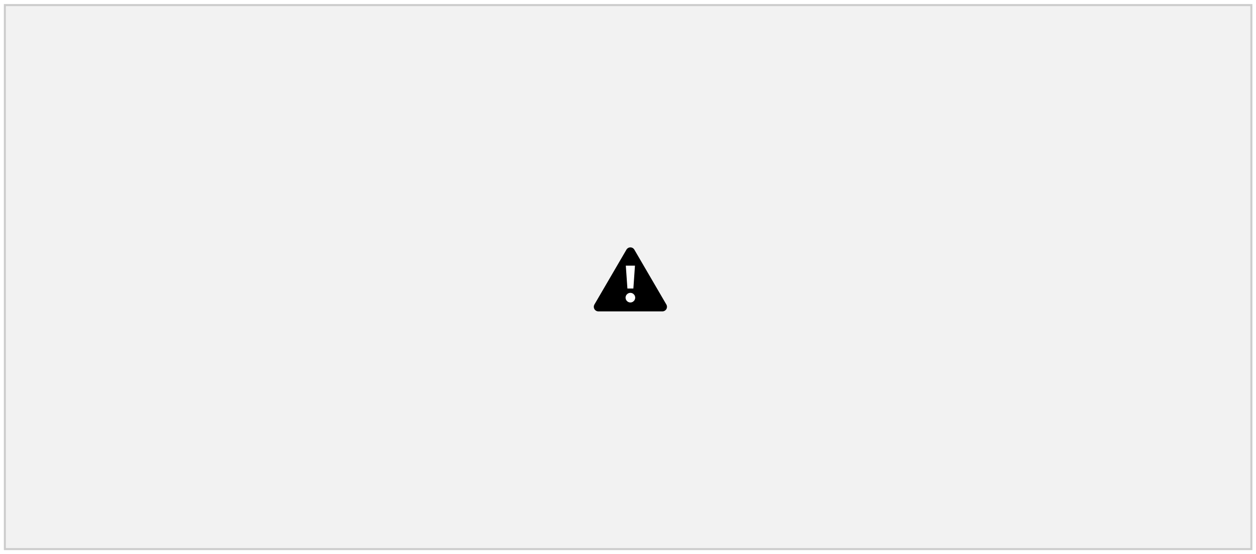
Std::exception

- 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



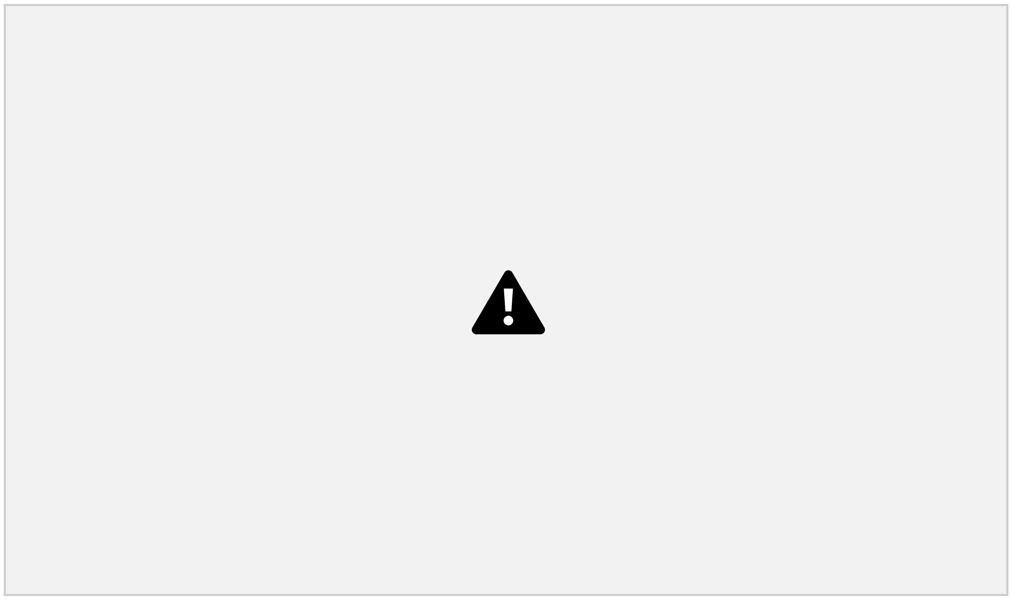




So you can make your own exceptions using the exception class. Each one has to redefine the what function (since the what function is a pure virtual function).

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!



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:

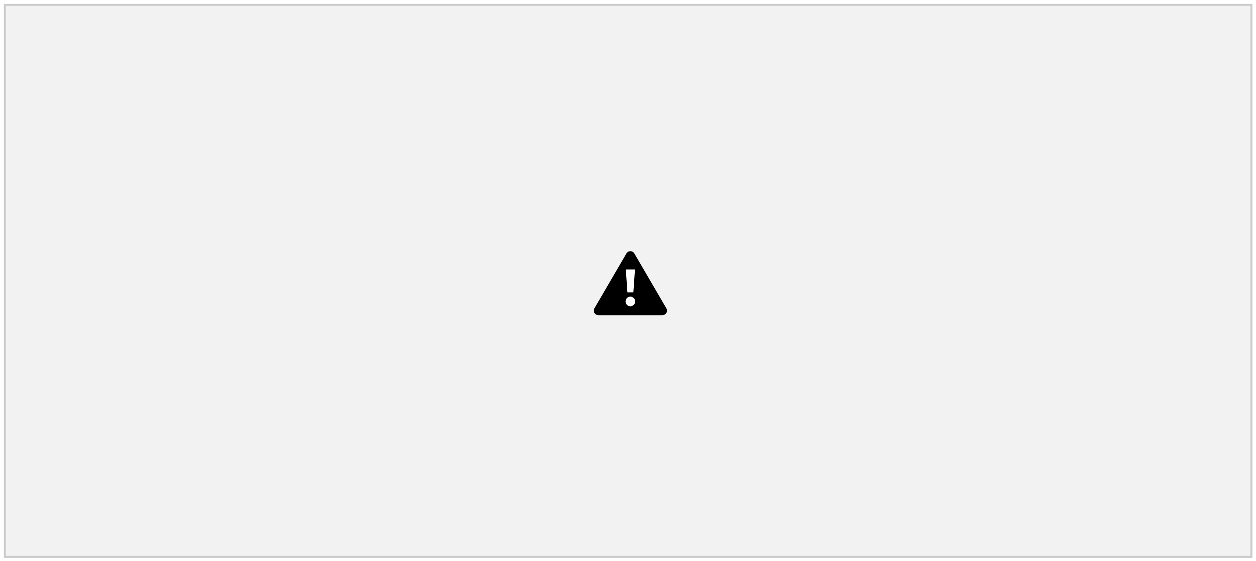
- **Write this first**
- **You can see everything**
- **You can see items and such**
- **Etc. Anything else that is helpful.**

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

Tertiary Operator:

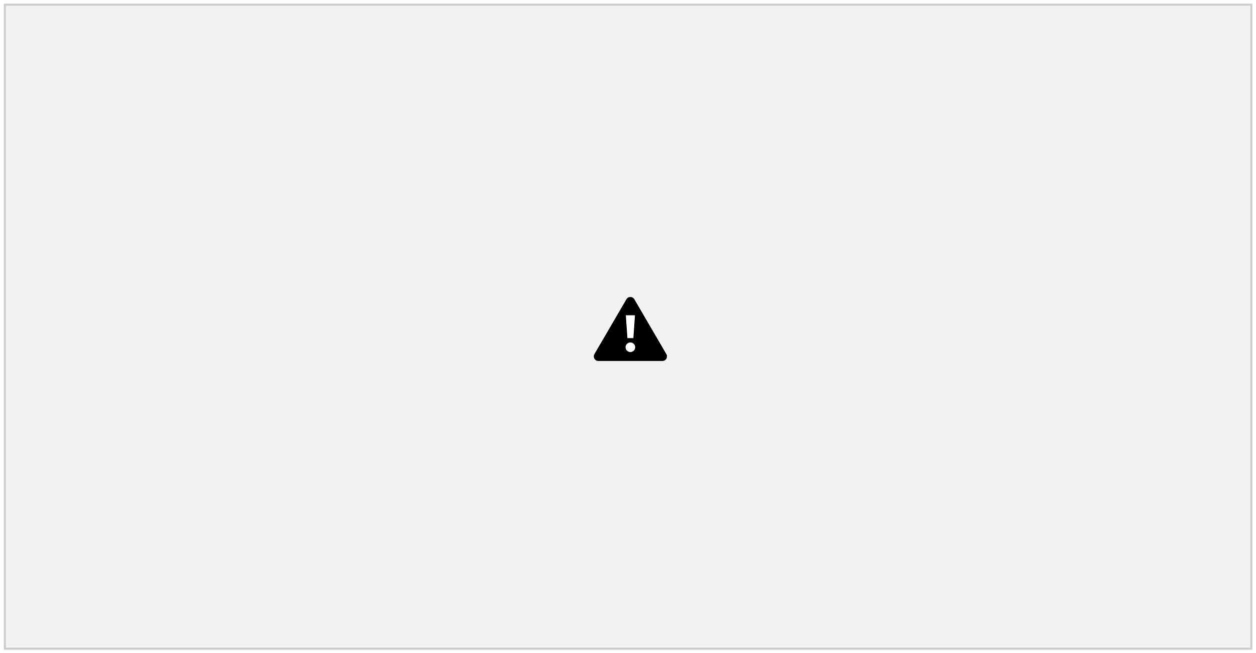
```
Return (x < y) ? x : y;
"Evaluate what's in the (), if it's true, return x, else return y"
```

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.

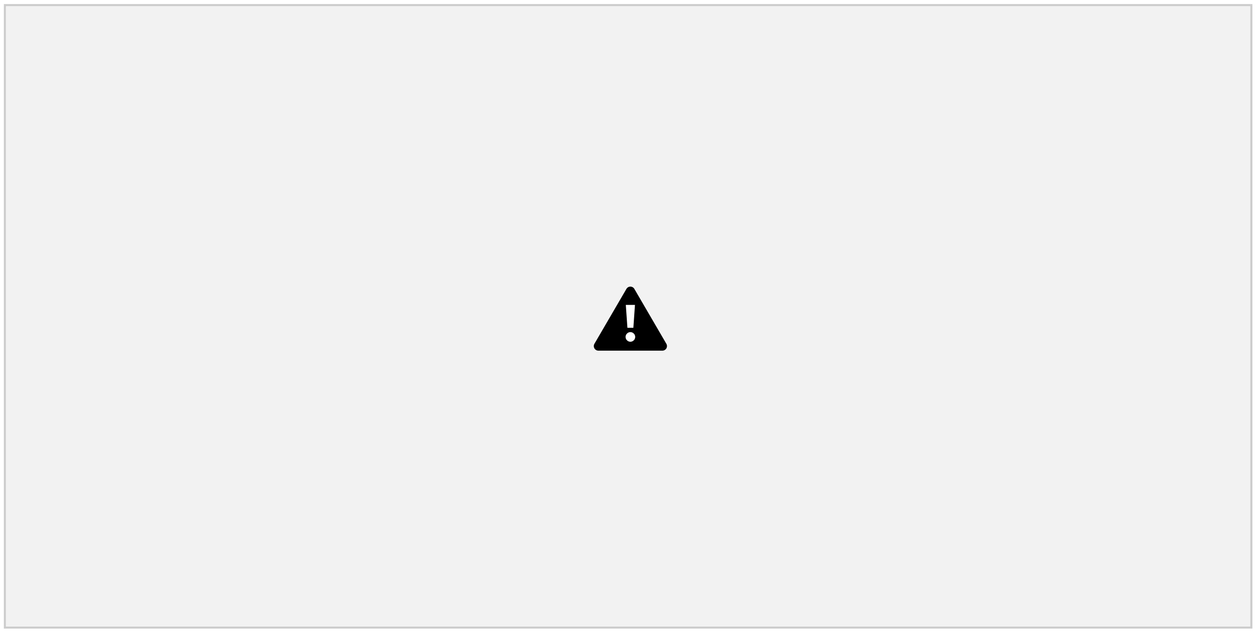
Pros:

- **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.

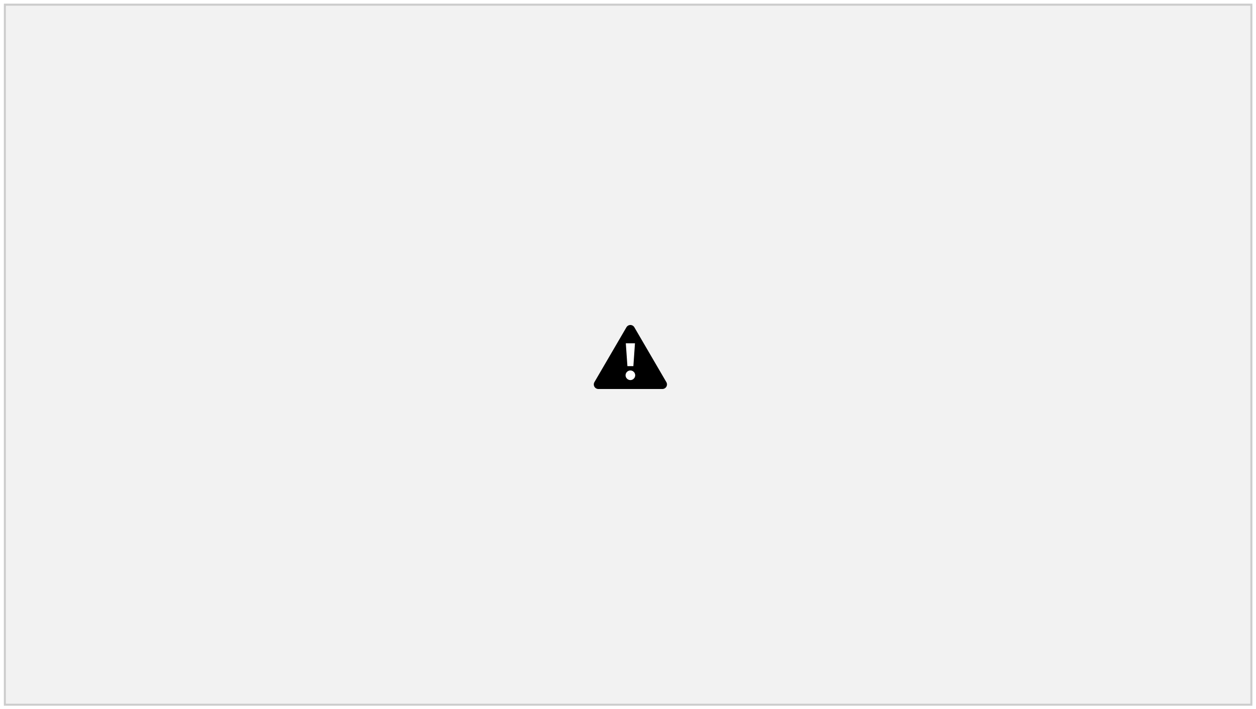


Preview! Much much easier than dynamic memory, take preview if you want to start on assignment 4.

CS162 Notes Spring 2021 **5/14/21 - Lecture 20 - Vectors & Template**

classes!

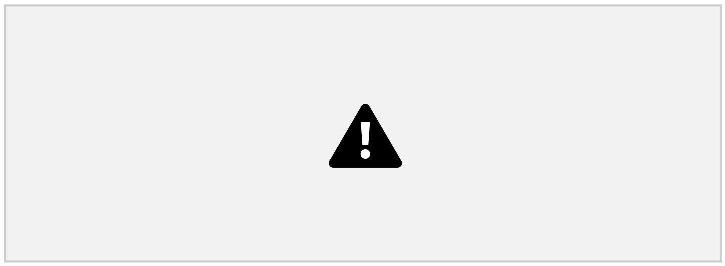
Function template **instances** -- these are what the specific examples of template functions that the compiler generates are called



```
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;
            }


};
```

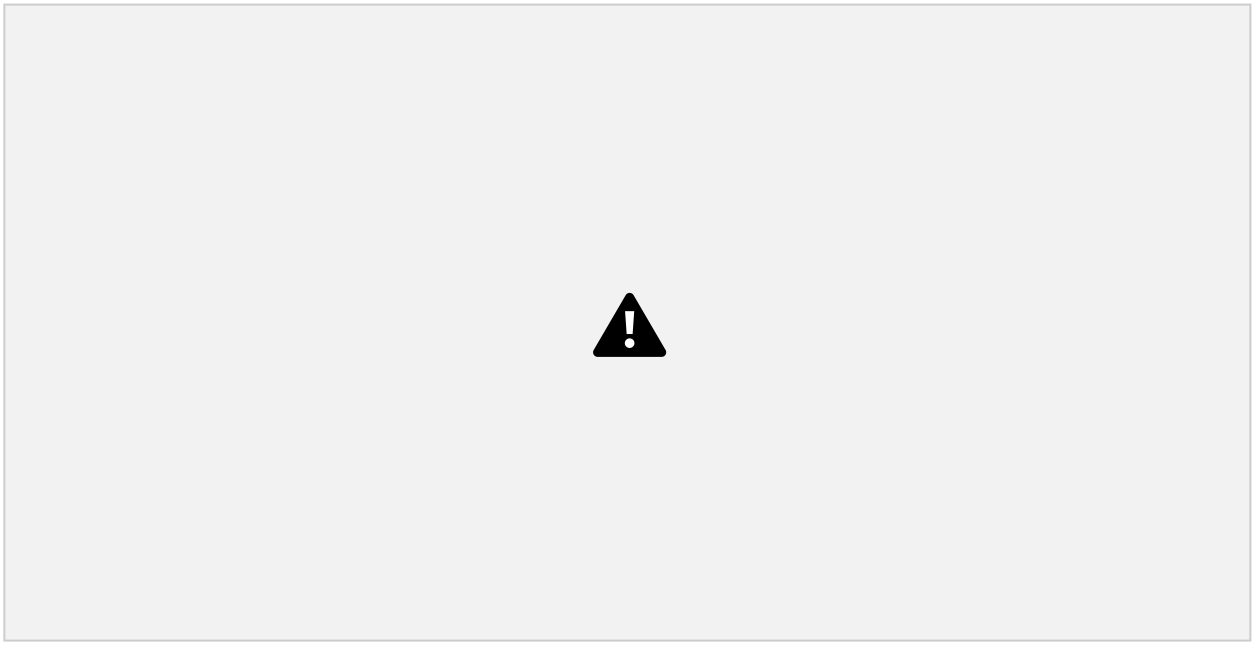


You can overload bracket operators -- it must accept an int, and then return a value at some index.

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++

CS162 Notes Spring 2021



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.