ENGR 151

Fall 2023

Lab 9: Object-Oriented Programming

Use this checklist to ensure that you've completed all activities in this lab.

student.cpp: should contain your modified version of the starter file. You should make sure your code generates the **provided output** without modifying anything except the TODOs.

buses.cpp: Should contain your modified version of the starter file. In particular, make sure your code **can produce the output specified** in the assignment below with the data files on Canvas. We will be looking to make sure that you've declared and defined your class functions separately, and that all TODOs are completed. Don't print anything additional and **comment out debugging statements before submission.**

Lab 9 FAQ Link:
https://docs.google.com/document/d/1TLK0qH12LKiReGX6L2XUD3-YdclTQ8k6LqUJpTfbS6o/edit?usp=sharing

Introduction

Object-oriented programming (OOP) is one of the most widely used programming *paradigms* today. OOP is based on the principles of *Encapsulation, Abstraction, Inheritance,* and *Polymorphism.* In this lab assignment, you'll mostly explore encapsulation and abstraction, but let's look at all of these principles briefly:

Encapsulation refers to the idea that objects should not be modifiable from everywhere or everyone. There should be a defined set of functions and variables of an object that the programmer should be able to call, modify or access, while everything else should only be accessible by the object itself. This concept allows classes to hide data and functionality to prevent misuse of the object. As an example, a vector keeps track of the capacity of itself in a private member variable - if we could modify this, we could wreak havoc on the vector.

Abstraction is hiding the implementation details of a class. Most programmers are interested only in **using** the member functions of a string, and don't necessarily care about how it's implemented under the hood. Indeed, if every line of code had to be analyzed in every project, there would never be enough time to validate it.

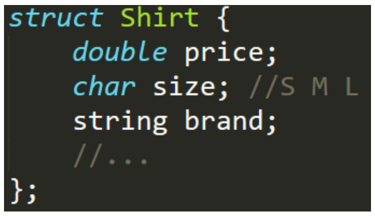
Inheritance - often, objects you'll use will have a lot in common. It is useful to create a *base* class where common details about an object are stored, and then define *derived* objects which share the same properties, but add more functionality or data.

Polymorphism has already been seen a few times in the course in the form of overloading. When combined with inheritance however, we have a powerful mechanism for using the same functions for similar objects when the relevant qualities of the objects are not different. For example, two classes named shirt and pants derived from a base class clothing would almost certainly want to implement a `getColor()` function, and they could even share the same definition. This allows the amount of code in a project to be significantly reduced.

Structs and Classes:

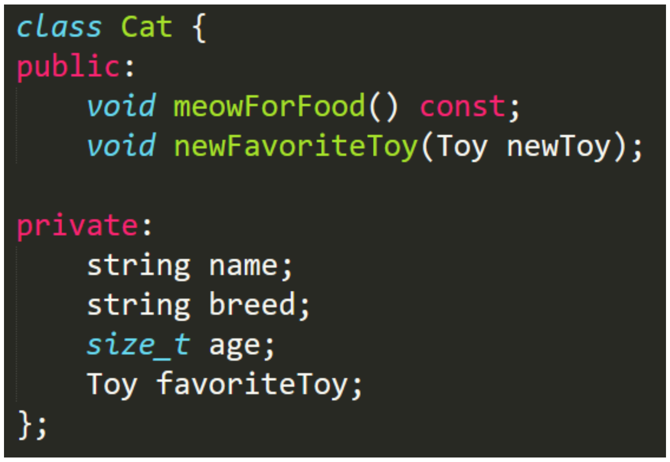
While C++ treats structs and classes almost exactly the same way (only the default access specifier is different), by **convention,** structs are used for storing only data, while classes are used for data **and** functions. Following this convention, let's look at structs first.

Structs: It would be quite cumbersome to keep track of lots of data points with individual variables. The solution we've employed so far is to use arrays or vectors, but what about when the data types we want to store are not necessarily the same, or there are several characteristics of the data? For example, if a clothing store is keeping track of its inventory, it might want to keep a record of the items in stock. A given shirt could have a price, season, color, size, brand, style, or any other number of characteristics the store would want to track. Clearly, these have different data types, and so we could use a struct to group them. The syntax for this is shown below.



```
struct Shirt {
    double price;
    char size; //S M L
    string brand;
    //...
};
```

Classes: Of course, it would be great if we could make this object more powerful - for example, changing the price when it goes on sale or clearance or throwing it away when it's out of season. We **could** write a function taking in the struct as an argument, but this would not be nearly as clean as using the class interface provided by C++. You've already dealt with several built-in classes of C++: vectors, strings, filestreams, etc. While incredibly useful, eventually, their usefulness is outlived as you create increasingly powerful and specialized code. At that point, you will rely heavily on the class mechanism to provide robust definitions for necessary objects



```
class Cat {
public:
    void meowForFood() const;
    void newFavoriteToy(Toy newToy);

private:
    string name;
    string breed;
    size_t age;
    Toy favoriteToy;
};
```

in your code.

Let us review the class syntax. You start a class definition with the class keyword and then give it a **name.** By convention, programmer-defined classes start with a capital letter. The definition of the class itself must be **within curly braces**. You can include data of built-in types or even other classes, as well as create functions specific to that class (a sample of this is shown on the left).

Notice the const keyword at the end of `void meowForFood() const;`. We would expect that this function should only make the cat meow - **not change any members of the class** - which is guaranteed by using `const`.

Exercise 1: Practice with structs

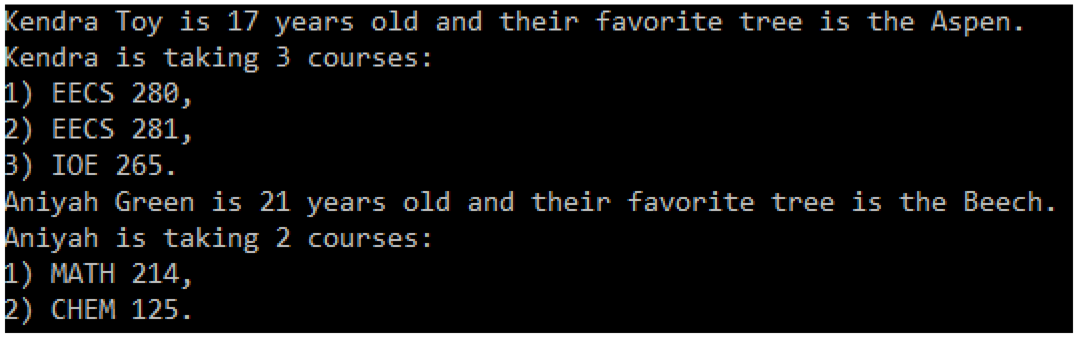
The idea behind this exercise is similar to the data_entry exercise, except now we're working in the other direction - given less human intelligible data, we would like to make it prettier for humans. This idea is known as *pretty-printing*. You're given functions that read in and parse data from a data file **"student_data.txt".** These functions take in a vector of type Student, which is a struct. Unfortunately, the struct definitions have gone missing and you have to write new ones to make the printer work again!

You will have to define two structs:

1) The `Course` struct defines a type that holds a single course as a department and a course number. For example, ENGR 151 is in department "ENGR" and has course number 151. Define this struct with two members department and courseNum as indicated in the starter file.
2) The Student struct is used to hold information about a single student. This includes their age, first and last name, courses they're enrolled in, and their favorite tree. You should define a struct that can hold all of these variables as indicated in the starter file.

To accomplish this task, you will find a starter file called "**structs_practice_starter.cpp**" on Canvas. You will finish writing this program by defining the `structs` described above. Look for the comments in the starter file that begin with "TODO" to finish implementing the program.

After running your completed program, the final 9 lines printed should be:



```
Kendra Toy is 17 years old and their favorite tree is the Aspen.
Kendra is taking 3 courses:
1) EECS 280,
2) EECS 281,
3) IOE 265.
Aniyah Green is 21 years old and their favorite tree is the Beech.
Aniyah is taking 2 courses:
1) MATH 214,
2) CHEM 125.
```

Submit your modified code in a file called "**student.cpp**" on Autograder.

Exercise 2: Magic Bus

The scenario in this exercise is simulating the daily operations of the Blue Buses. In particular, we're interested in making sure that everyone gets on a bus that will eventually take them to where they're going.

You've been given a file "**buses_starter.cpp**" on Canvas that contains starter code. Similar to the previous exercise, you must define a class to make the code work. In addition, you will have to implement some functions that demonstrate the ideas behind classes.

In particular, your program will first process bus data from a file called **"buses.txt"** that contains the route and stops associated with a bus - you should store this information to use throughout the program. You will then process a list of riders from **"riders.txt"** which contains the start and end points of several thousand students. Finally, you will simulate the daily operations of the bus by alternately moving buses forward by one stop and letting riders off an on as needed. The program finishes when all riders have at least gotten on a bus.

Much of the functionality is already provided for you as the purpose of this exercise is to learn how to implement classes and use them with your code. There will also be steps below to help guide you through the implementation process. Be sure to read them carefully. That said, spend extra time thinking about **why** specific choices are made regarding the design of the class. Think about ways you could improve it or use them in your own projects.

Follow the steps outlined here to complete the exercise and to get an idea of the power of object-oriented programming:

1) Let us start in `main()`. *Suppose* the class Bus and the struct Rider already existed. As with vectors and strings, we could find a list of the functions available to us and use those as an interface to use these objects without knowing how they're implemented. Similarly, we can abstract away the details of provided functions. So suppose you're told that the function `readBuses()` will store bus data read from a file into a list of Bus objects in the code. We can assume this is implemented correctly and simply use the just as we would `pow()` or `vector::size()`.
    a) Create the appropriate object to store bus data from `readBuses()` by checking what the function takes in.
    b) Create the appropriate object to store rider data from `loadRiders()` by checking what the function takes in.
2) A crucial exercise in designing an object-oriented program is to think about what your classes need to look like. **What functions do they need? What variables should they have? What should be accessible by whom?** Let us think about these questions in this step.

   a) What variables should there be in a bus class? We know that we're supposed to store the data from our bus. As discussed, this includes a **list of stops** as well as a **route name**. Now it would be reasonable to assume that a bus has a **capacity**. Let us assume this is 50. How will a bus keep track of where it is? Perhaps we should track an index in our route list indicating our **current stop.** Finally, we should have a way of deciding who gets off of the bus at each stop, we can do this by keeping track of a list of rider **destinations** for people currently on the bus.
   b) What functions should we have? We know that at each simulation step, each bus moves forward a stop and passengers get off. Let us perform this in a single function. In addition, we would like our bus to be able to check if someone can get on (we will explore this further below). Additionally, you may find it helpful to define *helper functions* which perform some small operation. Now how will we create a bus object - is there anything we would like to initialize it with? It is usually a good idea to define a variety of *constructors* for our classes.
   c) What should be **private** or **public**? A good rule of thumb is if a function or variable should be called or accessed by something outside of the class, it should be public, and everything else private. When you work with inheritance (not today), this is slightly modified, but the idea remains mostly the same.
   d) We will also define a simple Rider struct - note that in doing so, we reserve the ability to easily expand the functionality later if we so choose by simply adding to the struct. As long as we leave the original functionality available, it is backwards compatible.
3) Let us start slow with implementing the Rider struct. Find the relevant TODO in the starter file and implement Rider as described in the code comments.
4) Now let us begin implementing our class. Find the TODO in the starter file corresponding to this. We will start with variables. Following the guidance above and the starter file comments, define the class and add **member variables.** Should these be public or private (remember, we don't want an abuser of our class to be able to modify the functionality - should just anyone be able to change the route? Probably not.). Add the following member variables:
   a) `id` is an `int` that identifies the bus.
   b) `routeName` is a `string` (i.e. Bursley, Commuter, etc.)
   c) `stops` is a `vector` of `strings` (i.e. [CCTC, Llloyd, Michell, …])
   d) `riderDestinations` is a `vector` of `strings` storing where people are going.
   e) `capacity` is an `int` of the maximum bus capacity.
   f) `currentStop` is an `integer` index for stops to access where the bus currently is
5) Most of the class member functions are already mostly implemented, but they aren't declared anywhere. Let us go through the functions required and match up the definitions. First, we will declare (don't define yet!) functions within the class:
   a) `void drive();` //should be public - why?

This function will simulate the bus moving to the next stop and letting people off of the bus.

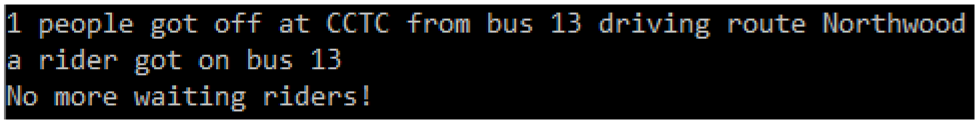
b) `bool getOn(Rider rider);` //should be public - why?
This checks to see if a rider can get on, and if so, lets them on and returns true. Otherwise, it returns false.

c) `bool stopsAt(string stop);` //should be private
This is a private helper function that simply checks whether the bus stops at the stop passed in. This **will be used by Bus::getOn(Rider)** to check whether the rider can get in the bus.

d) `bus(?, ?, ?)` //public - why?
This is our constructor for Bus. It will take in three arguments (in order) a routeName, a list of stops, and an id. Figure out what the types of these should be to finish declaring the constructor.

6) `Bus::drive()` is already fully implemented and you have declared it within `Bus`. Now, find where the definition is supposed to go and add the appropriate function header.
7) The constructor is mostly implemented and you have declared it within `Bus`. Find where the definition is supposed to go, add the appropriate function header, and finish initializing the values of a Bus object.
8) The `Bus::stopsAt(string)` function is already defined completely and you declared it - no TODO! Yay!
9) The `Bus::getOn(Rider)` function is not fully implemented. Start by finding the partial implementation and adding the appropriate header. You can try to finish implementing the function, but it will still not compile.
10) There is one more minor TODO in each of `loadRiders, moveBuses,` and `getOn` (note this `getOn` is different than `Bus::getOn`!) Complete these. If you have done the previous steps correctly, you should be able to compile.
11) Go back and fix the logic in `bus::getOn` to respect the comments in each condition

After completing the above, you can run the program and you will get a few seconds worth of output (we're processing thousands of riders!). The final three lines should be:



```
1 people got off at CCTC from bus 13 driving route Northwood
a rider got on bus 13
No more waiting riders!
```

Submit your modified code in a file named **buses.cpp** to autograder.io. Make sure to not print (cout) anything additional before submission.