Lab 2 - Classes, Constructors, and Operator Overloading

Bookkeeping

- Switch - 1 person from each group must move to the next group clockwise to them.
- Creating new groups, as well as your group number

Goals

- Convert a struct in C into a class in C++
- Implement both default and overloaded constructors
- Overload various arithmetic operators
- Implement getter/setter member functions

Task 1

- In the lab code, there is a file named **`Point2D.hpp`**. This is a C-style struct that needs to be converted into a C++ class with member functions, as well as standalone operator overloads to implement arithmetic with points.
- Your first task is to convert the **Point2D** struct into a C++ class, complete with:
    - A **default constructor** that sets member variables **x** and **y** to zero.
    - A **overloaded constructor** with parameters **(float, float)** to place the two numbers into member variables **x** and **y**, respectively.
    - Overload the four arithmetic operators:
        - **`Point2D operator+(Point2D lhs, Point2D rhs)`**
        - **`Point2D operator-(Point2D lhs, Point2D rhs)`**
        - **`Point2D operator*(Point2D lhs, Point2D rhs)`**
        - **`Point2D operator/(Point2D lhs, Point2D rhs)`**
    - Member functions for getting and setting the two components:
        - **`float get_x()`**
        - **`float get_y()`**
        - **`void set_x(float f)`**
        - **`void set_y(float f)`**
- Your Point2D class should be implemented in 2 files:
    - `Point2D.hpp` **(Modify this as necessary)**
    - `Point2D.cpp` **(Make sure to create this file)**
- To test your Point2D class, invoke **`` `make test-Point2D` ``**. The resulting program will run tests, and will crash only on failure.

Task 2

- In **`main.cpp`**, there is a program that tests use of a **Person** class. However, that **Person** class is unfinished.
- Your second task is to use your **Point2D** class to finish the design for a **Person** class that contains:
    - The name of the person as a **std::string (from <string>)**
    - The location of the person in a 2D coordinate space, represented as a **Point2D**

    - A default constructor that initializes the **Person** object with an empty string name and a location of (0, 0).
    - A overloaded constructor with signature **(std::string, float, float)** that will initialize the internal Point2D and std::string objects.
    - Getters and setters for the name and the location (the names for these functions can be seen in **main.cpp**)
- Your Person class should be implemented in two separate files:
    - `Person.hpp` **(Modify this file as necessary)**
    - `Person.cpp` **(Make sure to create this file)**
- To test your Person class, invoke **`make test-Person`**. The resulting program will run tests, and will crash only on failure.

Task 3

- Your third task is to adapt C code to figure out whether a 2D point lies within a polygon into C++ code that detects whether a **Person** objects lies within a polygon formed from the locations of an array of **Person** objects.
    - The C code comes directly from this Stack Overflow answer.
    - You do not need to modify anything in **main.cpp.**
- Your **pnpoly()** function should be implemented in two separate files:
    - `point_in_polygon.hpp` (Do not touch this file)
        - This file contains the correct type signature for how the function **pnpoly()** will be used in **main.cpp**. **Do not touch it!**
    - **`point_in_polygon.cpp` (Modify this as necessary)**

Notes:

- Unless a bug is found in the lab code directly, you do not need to touch any of the other files besides:
    - **point_in_polygon.cpp**
    - **Person.hpp**
    - **Person.cpp**
    - **Point2D.hpp**
    - **Point2D.cpp**
- Testing the code can be accomplished by invoking the **`make`** program in the **project/** directory using the command:
    - **`make test`** will invoke a preset set of commands within the **Makefile** that will test your program against our own solution for the desired program output/input.
    - **`make test-all`** will also run some additional programs from the **`tst/`** directory that test your two classes individually, as well as the commands from **`make test`**. **If all the tests pass, the program will not crash.**
        - **`make test-Person`** will invoke the test executable for the Person class
        - **`make test-Point2D`** will invoke the test executable for the Point2D class
    - If you wish, look at the Makefile and see exactly what commands are being run.

Submission

- In a file called **team.txt**, write your group number on the first line. On subsequent lines, write the last, then first names of your team members separated by a single comma, each on its own line. If your name contains whitespace (e.g. the single space in the last name, “da Silva”), you may keep the whitespace.
- Your final executable should have the filename **a.out**.
- **Comment your code! If your code does not work correctly, you may be able to get *partial credit* based on your thinking that we can infer from the comments!**
- Place your source code files into a directory called **project**, using the following directory structure:
    - `project/`
        - `Makefile`
        - `docs/`
            - *`team.txt`*
        - `src/`
            - `<put your source files here>`
        - `bin/`
            - `<your executables should appear here after invoking the Makefile>`
- Before turning in your .tar file, type **“`make clean`”** in the **project** directory to remove any executable files.
- Archive and compress your **project** directory using the command:
    - `tar czf name_of_archive_file.tar.gz project`
    - The name of the archive file should be in the following format:
        - `ee205-lab<lab-number>-group<group-number>.tar.gz`
- Submit the *tar.gz* file with one group member on Laulima for the assignment.
    - For your safety, test uncompressing and unpacking your archive in a clean folder, then compile your source files. Ensure that the executable will run correctly.

Supplementary Material

OOP in C++

- C++ has better support for “class-based object-oriented programming” compared to C.
- In C++, **structs** no longer “formally” exist as a core language construct -- instead, all **structs** are simply instances of **classes**.
- **Classes = member data + member functions**
- To access members, you use the **. (dot) operator** -- much like accessing member variables of a **struct** (as well as **-> (arrow) operator** for pointers to class objects’ members).
- To define new members of a class, just place their declarations/definitions within the curly braces for the class. See Figure 1.

```
1   struct Pair {
2       int x;
3       int y;
4   };
5   int Pair_get_x(Pair* p) { return p->x; }
6   int Pair_get_y(Pair* p) { return p->y; }
7
8   ////////////////////////////////////////
9
10  class Pair {
11  private:
12      int x;
13      int y;
14  public:
15      int get_x() { return this->x; }
16      int get_y() { return this->y; }
17  };
```

Figure 1 - In C, structs cannot be directly associated with functions.
In C++, classes can have member functions,which have an implicit argument called **this** which refers to the current object.

Ownership

- The idea that **ownership** decides which pieces of code own other data or functions.
- The time or space over which a thing is “owned” is called **scope.**
    - E.g. “Your local **int** variable remains in scope for only this function, when the curly braces denote the end of a scope.”
- In the OOP model, objects strive to establish strong ownership schemes or *semantics* over the data they own.
    - E.g. The memory for a length-varying **String** object should remain in scope only as long as it is used

    - E.g. Classes will own functions that only make sense when you're using them with the class (e.g. **`Car.honk_horn()`**).

OOP and Encapsulation

- Compared to C, C++ makes building abstractions more transparent and more fluid. Designing an object to be used by other programmers in code in C++ means that you have more options to hide "implementation complexity" from other programmers.
    - I.e. Using code doesn't mean you need to understand what's inside of it -- you just need to understand how it works.
    - "How it works" ⇒ **Implementation**
    - "How it's designed to be used" ⇒ **Interface**
    - **Encapsulation** is the basic idea that "you should not have to understand how something works in order to use it." In other words, one should hide the implementation and expose a clean interface.
- To use encapsulation, use the privacy access modifiers for class members.
    - **public**
    - **private**

Function and Operator Overloading

- In C++, you can have multiple functions with the same name but that have different type signatures.
    - For example:
        - **`int foo(int, int);`**
        - **`int foo(float, float, float);`**
- In C++, you are allowed to overload the functionality of the operator, which are just functions that look different, but are still functions.
- This is useful for repurposing the built-in operators to "make code look more natural."
- For example, when you see:
    - **`std::cout << "Hello world" << std::endl;`**
    - This is (essentially) converted into the following code:
        - `operator<<(operator<<(std::cout, "Hello world"), std::endl);`
    - Between the two, which would you like to type?
- Another example:
    - `3DPoint p1(1, 2, 3);`
    - `3DPoint p2(4, 5, 6);`
    - `3DPoint p3 = p1 + p2;`
    - Here, the code for **`p1 + p2`** is converted to **`operator+(p1, p2)`**,

Constructors

- In C++, constructors should take in data and "setup" an object for use.
    - E.g. a **Coordinate** class may take in N numbers and set up its internal data array to hold those N numbers.

- E.g. a **FileReader** class may take in a **std::string** and use it to open the correct file by name using the string and store the **FILE*** in its internal data.