C++ Lab 1: Jumping Right In

working with objects

C++ lets us program with *objects*. We describe objects in C++ by declaring and defining *classes*. We *declare* our class's structure in a header file, just like in C, and *define* it (that is, write the code that actually does the work) in a corresponding source code file.

Here is a sample header file `Point.hh` that describes an object that represents a point in two-dimensional Euclidean space:

```
// A 2D point class!
class Point {

private:
    // Coordinates of the point.
    double x_coord;
    double y_coord;

public:
    // Constructors
    Point();    // Default constructor
    Point(double x, double y);  // Two-argument constructor

    // Destructor
    ~Point();

    // Accessor methods
    double getX();
    double getY();

    // Mutator methods
    void setX(double val);
    void setY(double val);
};
```

We can *instantiate*, or create an instance of, our class anywhere in the rest of our code by calling any of the constructors we have defined:

```
Point myPoint;            // Calls Point::Point()
Point myOtherPoint(5, 3); // Calls two-argument constructor Point::Point(double, double)
```

Your Task

1. Copy the source code for the Point class (Point.hh and Point.cc) to a convenient working directory.
2. Change the `Point` class to represent points in three dimensions. Make sure to update the comments to match this change!
3. Add a new member function to `Point` called `distanceTo`. This member function should accept as an argument a `Point &` (a reference to a `Point`), and it should return a `double` that approximates the distance between the two points.

   You will probably find a square-root function useful for this! The C standard library has one, called `sqrt()`. The function takes a double and returns another double.

   If you were programming in C, you would `#include <math.h>`, but in C++ you say `#include <cmath>`. (This means, "Include the C Math header.") And then you are all set.
4. Create a new source file `lab1.cc` and implement two functions:
   - `double computeArea(Point &a, Point &b, Point &c)`

     which takes by reference three `Point` objects, computes the area within the triangle defined by these points (hint: use Heron's Formula), and returns it as a double-precision floating point number.
   - `int main()`

     which requests three points (nine coordinates) from the user, determines the area inside the triangle described by the three points using the aforementioned function, and displays the result.

     In C++, you don't have to declare all variables at the top of a block; you can intermingle variable declarations and statements of code. So, you should only need to use three `double` variables to input the coordinates, and then create each `Point` along the way.

     Don't worry about dealing with bad user input. For now, you can follow the oft-used program behavior: "Garbage in, garbage out!"
5. Compile these sources together like so:

   ```
   g++ -Wall lab1.cc Point.cc -o lab1
   ```

Remember:

- `g++` is the GNU C++ compiler.
- `-Wall` means "output ALL warnings." (If you see any warnings, be sure to fix them.)
- Next comes the list of source files. (You don't list header files here. They are included by the source files.) The only new part here is that we have *two* files: `lab1.cc` and `Point.cc`.
- `-o` specifies the output file. In this case, we are making a program called `lab1`.

6. Run the generated program:

```
./lab1
```

7. Go back and test `computeArea` with several boundary and/or extreme cases to make sure your implementation is robust for all triangles, including degenerate ones. Things you might try:

   - Triangles with no area - all three points on the same line
   - Triangles where two points are the same
   - Triangles where all three points are the same
   - Triangles with negative coordinate values
   - etc.

   In software-engineering parlance, these degenerate cases are called "edge cases" or "corner cases," because they test combinations of inputs that may be rare, but still possible. Usually, you want to start out by seeing if your program can handle simple cases that will expose obvious issues. Once your program can handle these, then you move on to the more subtle test scenarios.

Submission

That's it! Once you are finished, and have tested your code, submit your completed files via the csman website, for Lab 1 of the CS11 Intro C++ track! If you need a csman account, or if you aren't in the Intro C++ class, then please email Donnie to get things set up.
