2.2 - The Class Concept

Exercise 1: Point Class

Now you must use C++ syntax!!

In this exercise we start creating a *Point* class with and x- and y-coordinates. This class will be extended in further exercises.

In Visual Studio, create an empty "Win32 Console Application". If you don't check the "Empty Project" checkbox in the wizard, Visual Studio will generate code for you and will set the "use pre-compiled headers" on... Pre-compiled headers, which are a Visual Studio specific option, require special attention in your code and file settings so that is why an empty project is more appropriate.

First add a header file for the *Point* class with private members for the x- and y-coordinates. Do not forget to add the `#ifndef/#define/#endif` statements to avoid multiple inclusion.

Also make sure you make to following public functionality (see also Figure 1):

- Default constructor.
- Destructor.
- Getter functions for the x- and y-coordinates (*GetX()* and *GetY()* functions).
- Settter functions for the x- and y-coordinates (*SetX()* and *SetY()* functions).
- A *ToString()* that returns a string description of the point. Use the *std::string* class as return type.

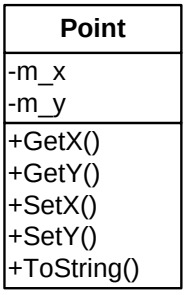

Figure 1: Point Class

Next create the source file that implements the *Point* class defined in the header file. The source file must include the header file.
Making the string in the *ToString()* function, requires conversion of the double coordinates to a string. Easiest is to use a *std::stringstream* object and the standard stream operators (as with *iostream*) to create the string. This requires the "sstream" header file. Use the *str()* function to retrieve the string from the string buffer. The output can be like: "Point(1.5, 3.9)"

Finally create a test program (separate source file with a *main()* function) for the *Point* class. It should do the following things:

- Include the point header file.

- Ask the user for the x- and y-coordinates using the *std::cin* object (needs the "iostream" header file).
- Then create a *Point* object using the default constructor.
- Set the coordinates entered by the user using the setter functions.
- Print the description of the point returned by the *ToString()* function.
- Print the point coordinates using the get functions.

Exercise 2: Distance Functions

In this exercise we are going to add distance functions to the *Point* class. The distance functions have the following signature:

```
double DistanceOrigin();   // Calculate the distance to the origin (0, 0).
double Distance(Point p);  // Calculate the distance between two points.
```

Add the definitions to the header file and implement the functions in the source file. Use the *std::sqrt()* function from the "cmath" header file to implement the Pythagoras algorithm.

Extend the main program to print the distance between the origin and another point and test it.

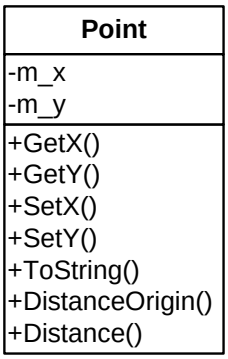

Figure 2: Point Class with Distance() functions

2.3 - Improving your Classes

Exercise 1: Extra Constructors

In this exercise we are going to add extra constructors. But first we do a little experiment. In the *Point* class constructor and destructor, add some code that displays some text. In the main program, make sure you use the *Distance()* function to calculate the distance between two points. Run the program and count how many times the constructor and destructor are called. Are they the same?

Now add a copy constructor to the *Point* class that also displays some text. Also add a constructor that accepts x- and y-coordinates so you can create a point with the right values without using the set functions. Use this constructor to create the point from the user input.

Run the program again and count the number of times the constructors and destructor are called. Is the copy constructor called and is the number of constructor calls now the same as the number of destructor calls?

We can derive two things from these results:

1. When calling the *Distance()* function, the input point is copied (call by value).
2. You will get the copy constructor 'for free' when you do not create one yourself.

Exercise 2: Pass by Reference

In the previous exercise, you saw that the point passed to the *Distance()* method was copied. Since creating a copy is unnecessary in this case, change this function so that it passes the input point “by reference” so that no copy is made. Pass it as “const reference” to make it impossible to change the input point from within the *Distance()* function.

Run the program again. It should call the copy constructor fewer times than before.
Also test if you can change the input point in the *Distance()* function. This should result in a compiler error.

Exercise 3: Function Overloading

Previously you saw that there could be more than one constructor as long as the input arguments are different. You can do the same for normal member functions. Thus you can rename the *DistanceOrigin()* function to *Distance()*. Also you can rename the *SetX()* and*GetX()* functions to just *X()*. The same is true for the setter and getter of the y-coordinate.

Exercise 4: Const Functions

In the test program create a const point and try to set the x-coordinate:

```
const Point cp(1.5, 3.9);
cp.X(0.3);
```

Compile the program. Did you get a compiler error? It should give a compiler error because you try to change a const object.

Now replace the line that changes the x-coordinate to code that reads the x-coordinate:

```
cout<<cp.X()<<endl;
```

Compile the program again. You will see that is still gives a compiler error even while retrieving the x-coordinate does not change the point object. This is because the compiler does not know that the function does not change anything. So we need to mark the x-coordinate getter as const by making it a const function. Do this also for the y-coordinate getter and the *Distance()* and *ToString()* functions because these don't change the point object as well.

Recompile the application. It should now work.

Exercise 5: Line Class

In the final exercise for this chapter we are going to create a *Line* class. The *Line* class has a start- and an end-point. So the *Line* class should have two *Point* objects as data members. This mechanism is called “composition”. See also Figure 3.

Give the *Line* class the following functionality:

- Default constructor (set the points to 0, 0).
- Constructor with a start- and end-point.
- Copy constructor.
- Destructor.
- Overloaded getters for the start- and end-point.
- Overloaded setters for the start- and end-point.
- A *ToString()* function that returns a description of the line.
- A *Length()* function that returns the length of the line. Note that you can use the distance function on the embedded*Point* objects to calculate the length. This mechanism is called “delegation”.

Use const arguments, const functions and pass objects by reference where applicable.

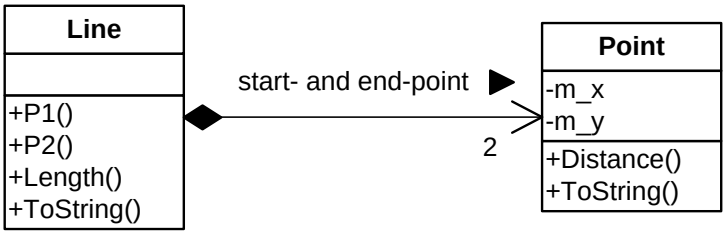

Figure 3: Line Class

Exercise 6: Circle Class

Create a *Circle* class. It has a center point and radius. Create the proper constructors, destructor, selector and modifier functions. Also add functions for getting the diameter, area and circumference. Don't forget a *ToString()* function.

In further exercises, all optimizations requested for *Line* should also be implemented for *Circle*. See also Figure 4.

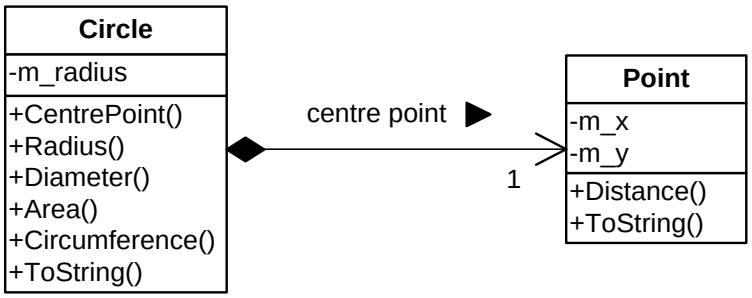

Figure 4: Circle Class

Note that instead of using your own PI value, Microsoft's version of the math library defines a constant for PI named M_PI. But because it is not standard you need to enable that define by setting the _USE_MATH_DEFINES symbol in the project settings or add before including "cmath":

```
#define _USE_MATH_DEFINES
```

https://www.quantnet.com/threads/level-3-exercise-2-3-6.9269/

Exercise 7: Inline Functions

Inline functions can speed up the execution of short functions because the code of such function will be copied in place instead of calling that function.

Make the setters and getters of the *Point* class inline functions. Use normal inline (outside the class declaration) for the getters and default inline (inside the class declaration) for the setters. *Note that the implementation of the normal inline functions must be in the header file*; else they will not be inlined.