MECHTRON 2MD3
Data Structures and Algorithms for Mechatronics
Winter 2022

Assignment 01

Published: January 22, 2022

Due at 11:59pm on February 2nd, 2022.

Please read the following carefully:

- Late submissions will be marked with a penalty of 20% per day.
- You have to submit one **PDF** file containing the answers to Question 1 and Question 2, and **a source code** file containing the answer to Question 3.
- The answer to Question 3 must be submitted as a source code file (a cpp or a zip file containing the source code). Do not submit the source code as PDF file, otherwise you will receive a mark of **zero**.
- Please ensure your answers are typed and clearly readable. Non-typed solutions (such as an image of the source code or handwritten solutions), will not be marked and receive a mark of **zero**.
- Include your name and student ID number in the PDF file and the source code file. You can add a comment on top of the source code indicating your name and student ID.
- Your work must be your own. Plagiarism and copying will not be tolerated.
- All files are to be submitted via Avenue to Learn.

This assignment consists of 3 questions, and is worth 50 marks.

1 Question 1 [7 marks]

Explain:

- two differences between the Procedure-Oriented design (using C structs) and Object-Oriented design (using C++ classes). [4 marks]
- when a copy constructor is strictly required for a class to function properly? [3 marks]

2 Question 2 [8 marks]

Identify **four** issues (errors or possibly poor design) with the following class definition, explain what is the issue and also explain how to correct each of them:

```
#include <iostream>
using namespace std;

class Counter {
    public:
        void Counter(int c = 10) : count(c) { }

        ~Counter(int c){ }

        int getIncrementedCount() const{
            return ++count;
        }

        int getCount() {
            cout << "Counter is " << count << endl;
        }
    private:
        int count;
};
```

[each issue found worth 1 mark for the explanation and 1 mark for the correction. The answer has 8 marks in total.]

3 Question 3 [35 marks]

Design a C++ Object-Oriented Class structure to model a simple Polygon and its relation with Points. In geometry, a **Polygon** is an object with a number of vertices (**Points**) that can be drawn by connecting consecutive vertices with line segments, and connecting the last Point to the first Point.

Figure 1, shows an example Polygon with 5 Points: $\langle(2,1),(3,3),(5,4),(7,2),(4,-2)\rangle$. Note that, consecutive Points are connected to each other with red line segments (sides of polygon) and the last Point, i.e., (4, -2) is connected to the first Point, i.e., (2, 1).

Important note: If the order of Points changes, the sides of Polygon may intersect each other. For example, if the order of Points changes to: $\langle(2,1),(7,2),(5,4),(3,3),(4,-2)\rangle$, you will get the Polygon in Figure 2.

You are **NOT** required to model such a Polygon. You have to **assume** that the user of the class takes care of entering Points in the correct order such that Polygon sides do not intersect with each other. So, your task is simple. Just

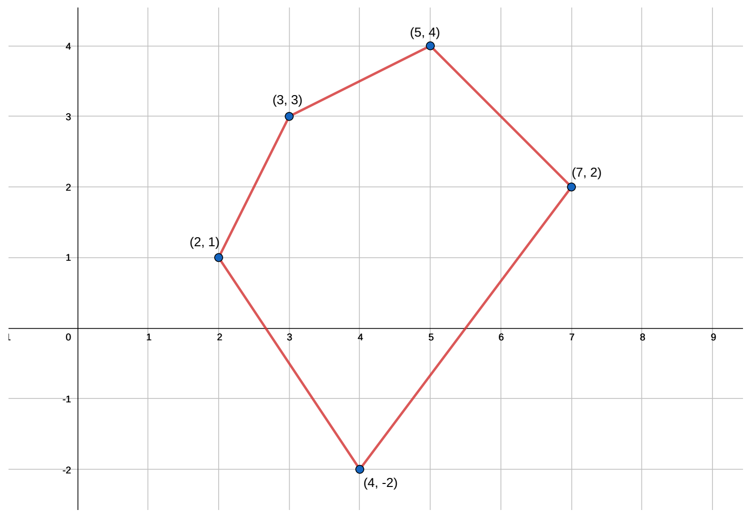


Figure 1: A Polygon with 5 Points.

don't worry about the order of points and always assume that the user entered them correctly.

In this question, you are asked to design a Point class with the following properties and behaviour:

- must have two double member variables x and y as coordinates of the Point. Users of the Point class must not have access to these members. [1 mark]
- must define a constructor with **no arguments**. This constructor can initialize member variables to zero, inside its body. Note that, this is different from having a constructor with default arguments. This constructor serves instantiations like this:

  ```
  Point p1;
  ```

  [2 marks]
- must define a constructor with arguments that can set member variables x and y. [2 marks]
- must have appropriate methods that allow getting the values of private members. [2 marks]

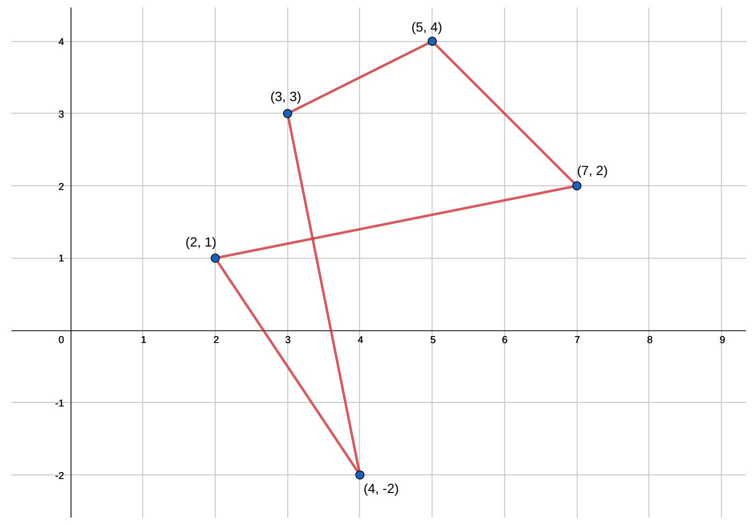


Figure 2: A **wrong** order of Points for the same Polygon in Figure 1. You don't care about this case!

- must have a member method **"distanceTo(otherP)"** that computes the object's Euclidean distance to "otherP". Note that, this method receives an instance of the Point class as parameter. Please refer to Section 4 (Appendix A) of this document for a note on the Euclidean distance. [3 marks]

- must overload >> operator so that a user can use such an statement to input the values of x and y for the point:

```
Point p1;
cin >> p1;
```

  [2 marks]

- must overload << operator so that a user can use such an statement to output the values of x and y in an appropriate manner:

```
Point p1;
cout << p1;
```

  For example, you can output a Point as "P(6, -2)" where x and y values are 6 and -2, respectively. [2 marks]

After defining the Point class, you can define the Polygon class. Remember that a Polygon is an object that has some Points in it. In our *Vect* class examples during lectures, we had a similar relationship. A *Vect* consists of some **int** values. You can think about this Polygon class in the same manner as the *Vect* class, with a difference that a Polygon has some Points.

You are asked to design a Polygon class with the following properties and behavior:

- must have a member variable to store the number of Points (vertices) and a dynamically allocated member variable to store Points. Users of the Polygon class must not have access to these members. [2 marks]
- must define a constructor with an argument that indicates the number of Points (default could be 10 like the *Vect* class). This constructor serves instantiations like this:

  ```
  Polygon poly1(5);
  ```

  [2 marks]
- must define a constructor with two arguments, one to receive the number of vertices and another to receive an array of pointers to Point objects. This constructor serves instantiations like this:

  ```
  Point* points_list = new Point[4];
  points_list[0] = Point(0, 0);
  points_list[1] = Point(0, 1);
  points_list[2] = Point(1, 1);
  points_list[3] = Point(1, 0);

  Polygon rectangle(4, points_list);
  ```

  This example creates the rectangle in Figure 3.

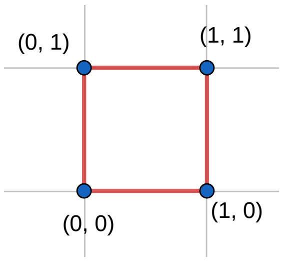
  

  Figure 3: Simple Rectangle using the Point and Polygon class

  [3 marks]

- must have a destructor. [2 marks]
- must have a member method **"perimeter()"** that computes the Polygon's perimeter. Note that, you have to compute the distance between consecutive Points of the Polygon **using the "distanceTo" member function of each Point**. For example, the perimeter of Polygon in Figure 3 can be computed by calculating the **sum** of the following distances:

  1. the distance from $(0, 0)$ to $(0, 1)$
  2. the distance from $(0, 1)$ to $(1, 1)$
  3. the distance from $(1, 1)$ to $(1, 0)$
  4. the distance from $(1, 0)$ to $(0, 0)$

  which evaluates to 4.0. So,

  ```
          rectangle.perimeter();
  ```

  will return 4.0. [4 marks]
- must overload >> operator so that a user can use such an statement to input consecutive Points:

  ```
          Polygon poly2(4);
          cin >> poly2;
  ```

  [2 marks]
- must overload << operator so that a user can use such an statement to output the values the consecutive Points of a Polygon in an appropriate manner. For example:

  ```
          cout << rectangle;
  ```

  may output the following line for the example of Figure 3:

  ```
          P(0, 0) P(0, 1) P(1, 1) P(1, 0)
  ```

  [2 marks]

Note that, if you appropriately design overloaded operators >> and << for the Point class, the overloaded operators >> and << for the Polygon class will automatically use those functionalities of the Point class.

Note that correct declaration of Point [2 marks] and Polygon [2 marks] classes is required.

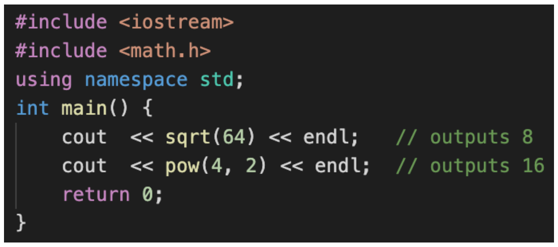
4 Appendix A

Given two 2D points (x_1, y_1) and (x_2, y_2), the Euclidean distance between them can be computed as:
$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

In C++, the mathematical function *sqrt(a)* calculates the square root of a. The function *pow(b, c)* finds the value of b raised to the power of c. Both of these functions are available, in the **math.h** library. So, you have to *include* this library:

```
#include <math.h>
```

This code fragment prints the value of 4^2 and $\sqrt{64}$ and may help you to implement the Euclidean distance:



```
#include <iostream>
#include <math.h>
using namespace std;
int main() {
    cout  << sqrt(64) << endl;   // outputs 8
    cout  << pow(4, 2) << endl;  // outputs 16
    return 0;
}
```

Figure 4: How to use *sqrt* and *pow* in C++