National University of Singapore
School of Continuing and Lifelong Education

TIC1002: Introduction to Computing and Programming II
Semester II, 2020/2021

Problem Set 4
Object Oriented Programming

Release date: 1st April 2021, 12:00 mn
Due: 18th April 2021, 11:59 pm

Introduction

This problem set is to introduce/revise object oriented programming concepts in a "gentle" way. For the first two parts, we will rewrite old code into OOP approach. This serves both as example of how OOP code looks like as well as a way to contrast the "procedural/function based" approach and the OOP approach. The last part we will apply the OOP concepts and write a new class from ground up.

Task 1: 2D Point (2 marks)

Refer to the question 1 in lecture 6. ADT training on Coursemology. In that training, we introduce a simple Point ADT that represents a location in a 2D Cartesian plane, e.g. $(1, 2)$ is a point where the X-coordinate is 1, Y-coordinate is 2. The code was written in the procedural style, i.e. structure + related functions.

In the given template file, the same ADT is now rewritten into a C++ class. Other than some minor name changes for the method (to make them sounds better and to improve readability), the general logic is the same. Use this opportunity to contrast the procedural and OOP approach. Pay attention to:

- How the "internal representation" of a Point is now hidden from user by using private declaration.
- Information (i.e. the structure) is now "inherent" in the object. User need not pass in the information of a point for the function to work in the OOP approach.
- Take note of the `distance_to()` method, which highlight a typical way to interact with other object of the same class.

Deliverable: `virtual bool equal(const Point& p)`

Your task is relatively simple. Implement the method `equal`, which checks whether the point object contains the same coordinates as another point object.

Task 2: Line connecting points (3 marks)

Similarly, refer to question 2 of the same Lecture 6 ADT training for the Line ADT. The template file gives the equivalent implementation in OOP approach. Pay attention to the interaction between Point ADT and Line ADT. Observe that in the OOP approach the Line ADT can no longer directly access the internal information of a Point ADT. The only way for an "external party" to access the internal representation of an object is to go through the public methods provided, e.g. we need to use `set_x()` to change the x-coordinate of a point now instead of directly changing the internal data.

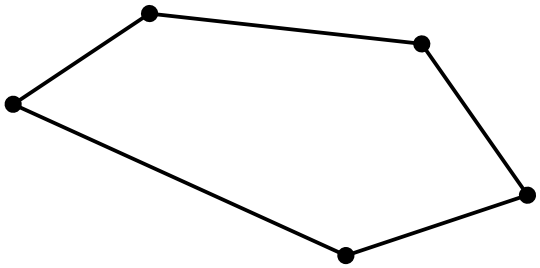
Deliverable: `bool equal(const Line & l)`

Your task is equally simple. Implement the `equal` method to check whether two lines are the same. Note that line "(1,1)–(2,2)" is the same as the line "(2,2)–(1,1)", i.e. as long as the two end points are the same, the lines are considered as the equivalent.

Task 3: Polygon (20 marks)

This is the "real meat" of this problem set. Part A and B should have given you enough understanding on how a class work, now it is time for you to implement a class from ground up.

The target class, polygon, represents 2D shapes composed by connected straight lines, e.g.



Deliverable: `Polygon(const Point & p1, const Point & p2, const Point & p3)`

Since the simplest Polygon has 3 points, we force the user to pass 3 points into the Polygon constructor. For simplicity, you can assume the 3 points will form a valid Polygon (i.e. there are no duplicate points, the points are not on the same line, etc).

You need to decide an appropriate internal representation of the Polygon before you can start, e.g. do you want to store the lines of the polygon, or just the points, or both? Should you use array, vector, linked list or something else? etc. Declare the appropriate information in the private section of the class.

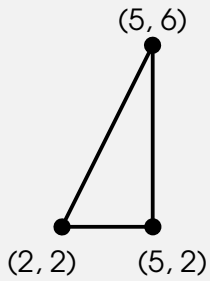
Also, you may want to read through the entire list of tasks before making your decision.

Example:

```
Point p1(2,2), p2(5,2), p3(5,6);
```

```
Polygon pg(p1, p2, p3);
```

The above represents a polygon `pg`:



With the constructor implemented, you may want to complete the following helper methods next.

Deliverable: `virtual void print( ostream & out )`

Similar to `Point` and `Line` class, write a print method to display the information of the polygon in the following format (␣represent a space):

(X_1,Y_1)␣->␣(X_2,Y_2)␣->␣(X_3,Y_3)␣->␣(X_1,Y_1)

Notes:

- There can be more than 3 points in a Polygon object.
- The first point is printed again at the end to indicate that the last point is connected to the first point.

Example: `pg.print();` produces the output `(2,2) -> (5,2) -> (5,6) -> (2,2)`

Deliverable: `virtual double perimeter()`

This method calculate and print the perimeter of the polygon, i.e. the sum of the length of all lines composing the polygon. Again, note that the polygon can have more than 3 sides.

Example: `cout << pg.perimeter();` produces the output `12`

With the helper methods in place, you can now implement the more challenging methods. We are going to expand the polygon by adding new points. Let's define two different versions as follows.

Deliverable: `virtual bool add(Point & p)`

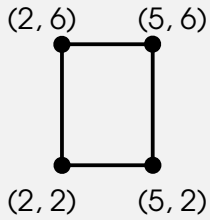
Add point p as **the last point** in the polygon. You need to ensure that p is not a duplicate of any of the existing points in the polygon. Return `true` if point p is added successfully, `false` otherwise.

Note: You can assume that point p (if it is not a duplicate of any existing points) will form a valid polygon, i.e. we will NOT pass in a point that cause any of the line to cross each other.

Example:

```
Point p1(2,2), p2(5,2), p3(5,6), p4(5,2), p5(2,6);
Polygon pg(p1, p2, p3);
pg.add( p4 );              // false is returned as p4 is a duplicate of p2
pg.add( p5 );              // this will add p5 to the polygon successfully.
```

After adding p5, the polygon looks like:



You should test the `add` method by using `print` and `perimeter` methods.

Deliverable: `virtual bool add(Line & exist, Point & p)`

This version of add removes the line `exist` from the polygon, and adds the point `p` *between* the original end points of the line. You need to ensure the line `exist` is indeed one of the existing lines in the polygon before adding point `p`. Return `true` if point `p` is added successfully, `false` otherwise.

Notes:

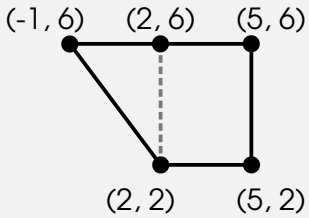
- The line exist can be any line that forms the current polygon.
- You can assume that point p (if it is not a duplicate of any existing points) will form a valid polygon, i.e. we will NOT pass in a point that cause any of the line to cross each other.
- This is an example of method overloading. You can see that both methods are named `add`, but this is acceptable as long as the parameters are different. The C++ compiler will decide which version you are calling based on the parameters you pass in.

Example (continue from the example before, the polygon is now a rectangle):

```
Point p1(2,2), p2(5,2), p3(5,6), p4(5,2), p5(2,6);
Point p6(-1,6), p7(-1, 2);
Line  l5(p1, p5);
```

```
Line  l6(p1, p3);     // this line does not exist in the current polygon
pg.add( l6, p7 );     // false is returned, as l6 is not an existing line.
pg.add( l5, p6 );     // this will add p6 to the polygon successfully.
```

After p6 is added, the polygon looks like following. Note that the removed line l5 is shown as dotted line to aid your understanding only, you do not need to keep track of removed line in any way.



Sample output for your reference

The given template duplicated some of the examples above with minor modification. Read through the tests so that you have a better understanding. If your implementation is correct, the completed program should output the following (includes output from all parts):

```
Distance from p1 to p2 is 3
Is p1 the same as p2 =  0
Is p2 the same as p4 =  1
Is l1 the same as l2 =  1
Is l1 the same as l3 =  0
(2,2) -> (5,2) -> (5,6) -> (2,2)
Perimeter = 12
Add p1 to polygon failed!
Add p5 to polygon ok!
(2,2) -> (5,2) -> (5,6) -> (2,6) -> (2,2)
Perimeter = 14
Remove l6 and add p7 to polygon failed!
Remove l5 and add p6 to polygon ok!
(2,2) -> (5,2)
```