Advanced Programming

Homework Assignment 2

Classes with Dynamic Fields

General guidelines:

a. Maximize readability and ensure indentation.
b. Do exactly as required in the questions.
c. Every class in every question must be submitted in two seperate files – a header file (.h) and an implementation file (.cpp).
d. Add functions and helper methods as necessary to ensure readability.
e. Submission must be done according to the submission guidelines – a document published in the course website – read them carefully!
f. Use relevant names.
g. Use comments to document your functions and complex code parts. **Add an output example at the end of your file!**
h. Individual work and submission – paired submission is not allowed.

Important note: Unless otherwise specified, ever homework has no more than one week for submission.
Open submission boxes past the deadline are not permission for late submission.

Question 1:

Define and implement a class called **Point** that represents a point in a 2D plane.

The class must include the following fields (**private**):

- x – a floating point number for the x coordinate
- y – a floating point number for the y coordinate

The class must also include the following methods (**public**):

- Constructors:
  - A default constructor that receives no arguments and initializes the fields x and y to the value 0.
  - An assignment constructor that receives two arguments and assigns them to x and y.
  - A copy constructor that assigns x and y the corresponding values in the fields of the argument Point.
- A getter and a setter for every field.
- A method **distance** that receives an argument Point and returns the distance between the two points.

Define and implement a class called **Polygon** that represents a polygon in a 2D plane.

The class must include the following fields (**private**):

- A pointer to the address of a Point array of unspecified size.
- The number of points in the polygon (the size of the array).

The class must also include the following methods (**public**):

- Constructors:
  - A default constructor that receives no arguments and initializes the number of points to 0 and the array to nullptr.

- An assignment constructor that receives the number of points as an argument and uses dynamic allocation to initialize the fields accordingly. The points in the polygon are to all be initialized to zeros using the automatic call to their default constructor.
- A copy constructor that receives a Polygon instance and deep-copies its points into the instance being constructed. This means a new dynamic allocation of the same size is required.

- A destructor that frees the dynamically allocated memory.
- A getter for every field. The getter for the array must return the address of a newly allocated, deep-copied, array. (Why no setters?)
- A method **setPoint** that receives a Point and an index. The method sets the Point at the given index in the array to the coordinates of the Point received. This occurs even if the Point at the given index is already set to other coordinates.
- A method **perimeter** that computes the perimeter of the Polygon. The method is to sum up the distances between every two consecutive Points, assuming that the Points in the array are ordered according to the edges of the polygon (don't forget the distance between the last and the first Points). The method is to return a floating point number.
- A method **isIdentical** that receives a Polygon and returns true if it is the same as the calling Polygon. Sameness is defined as having the same Points and the same edges. Notice that the order of the Points is not necessariy the same, for instance: (0,0) (1,1) (2,0) is the same as (1,1) (2,0) (0,0) even though the order is different.

So as to better understand the behavior of the constructors add in Polygon the following outputs:

- In the default constructor print: "in default constructor"
- In the assignment constructor print: "in one parameter constructor"
- In the copy constructor print: "in copy constructor"
- In the destructor print: "in destructor"

Implement a main program that inputs data pertaining to two polygons and prints their perimeters, **rounded to the closest integer** (you may make use of the function round in the library cmath), as follows:

a. If the polygons are identical the program must print "equal" and on the next line print the perimeter (shared by both).
b. If the polygons are not identical the program must print "not equal" and on the next lines the perimeters – each in a separate line.

The input of the polygons will consist of the output "enter number of sides:" followed by the input of the number of points. Then the output "enter the point values:" followed by the input of the coordinates.
The coordinate input will be of the format: (x1,y1) (x2,y2) … (xN,yN)
Where N is the number of points in the polygon and (xi,yi) are the point coordinates.
After the input of each Point it must be saved in its place in the polygon using the method setPoint.
For any invalid input the program will print "ERROR".

Output examples:

Input of two right triangles with the edges of length 3-4-5:	**Input of a square with sides of length 2 followed by a scalene triangle (no equal edges):**

```
enter number of sides:
3
in one parameter constructor
enter the point values:
(10,10) (10,14) (13,10)
enter number of sides:
3
in one parameter constructor
enter the point values:
(13,10) (10,10) (10,14)
in copy constructor
in destructor
equal
perimeter: 12
in destructor
in destructor
```

```
enter number of sides:
4
in one parameter constructor
enter the point values:
(0,0) (0,2) (2,2) (2,0)
enter number of sides:
3
in one parameter constructor
enter the point values:
(1,1) (2,0) (3,1)
in copy constructor
in destructor
not equal
perimeter: 8
perimeter: 5
in destructor
in destructor
```

Question 2:

A **Circular Array** is a fixed length array in which values can be stored consecutively in a circular fashion. **Insertion** is done at the "**tail end**" of the array after the last element which was inserted. **Removal** is done from the "**head end**" of the array. The first element inserted is the first element removed. So as to make use of the full size of the array, the memory will be handled in a circular fashion, meaning that the element inserted after the element that is in the last index in the array is put in the first index (0). This means that elements are not always put after preceding elements and not always removed from the first index. The circular behavior will be implemented using modulo (the % operator) as described below.

Terminology:

- **head** – The index where the next element to be removed is.
- **tail** –The index where the next element to be inserted is.
- **capacity** – The maximum number of elements in the array at any given time. The actual size of the array will be defined to be **capacity** + **1** so as to differentiate between a full array (capacity elements) and an empty one (0 elements).

An **empty array** has the head and the tail on the same index:

- Tested by: head == tail
- Achieved by: zeroing both head and tail (head = tail = 0).

A **full array** has one available place after the last element:

- Tested by: (tail + 1) % (capacity + 1) == head

Insertion:

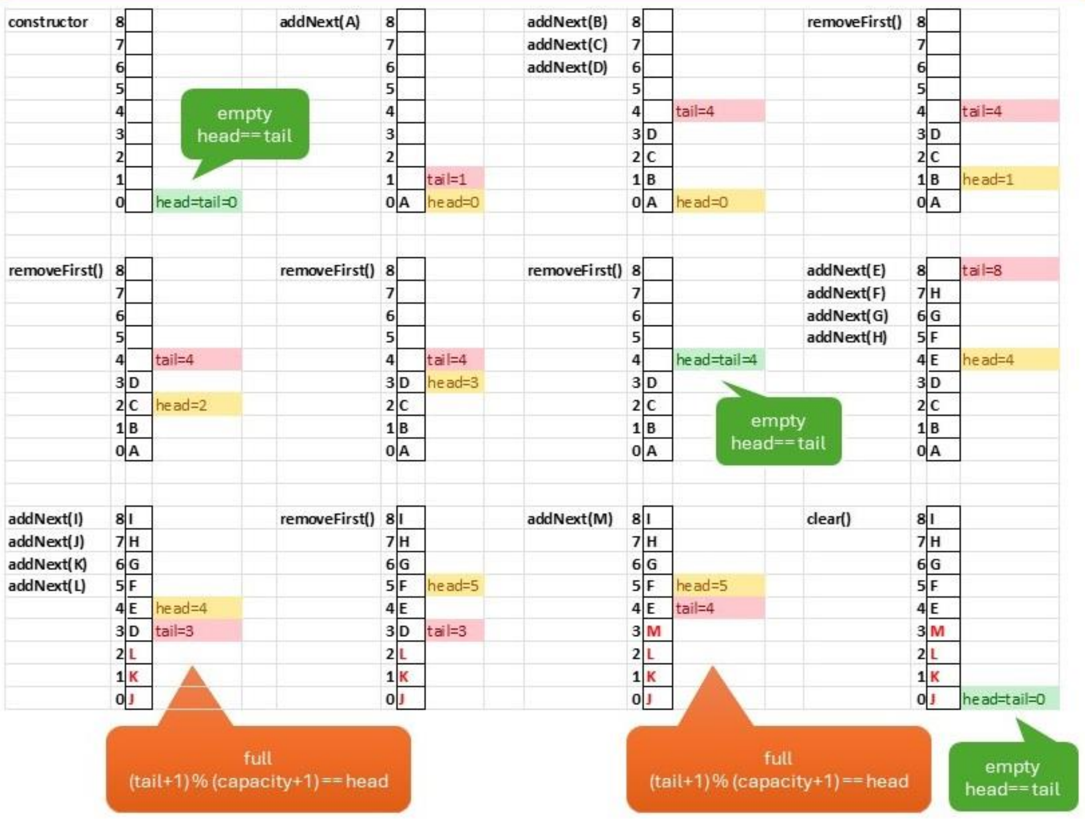
- Before: Test that the array is not full.
- Insert at tail and move tail to the next index.
- The tail must be moved in a circular fashion: tail = (tail + 1) % (capacity + 1)

Removal:

- Before: Test that the array is not empty.
- Move head to the next index. Do not remove value – leave it as garbage value.
- The head must be moved in a circular fashion: head = (head + 1) % (capacity + 1)

Demonstrated below is an example of execution of the different operations.
Read the examples from left to right line after line.
The requirements of question 2 follow the examples.

A circular array of type char with up to 8 elements.
The maximum number of elements: capacity = 8
The actual size of the array: capacity + 1 = 9



Define and implement a class called **RoundVector** that represents a circular array of integers.
You may base your implementation on the class **Vector** viewed in the lab. The class **RoundVector** will be similar but not identical to the class **Vector**: some of the fields and methods will be the same, some irrelevant and some completely new.

The class must include the following fields (**private**):

- **capacity** – the maximum number of elements that can be stored in the array. The actual number of elements will be less than or equal to capacity.

- **vec** – a pointer containing the address of an integer array size capacity+1. As explained above: the extra space is to determine if the array is full (with **capacity** elements) or empty (with no elements).
- **head** – the index of the next element to be removed from the array.
- **tail** – the index of the next available space in the array. The next element to be added to the array will be put here.

The class must also include the following construction/destruction methods (**public**):

- **constructor** – receives the maximum possible number of elements to be stored in the array. Initializes **capacity** and dynamically allocates **vec** to be **capacity**+1.
- **copy constructor** – receives an existing **RoundVector** (a cbr & const) and initializes the **RoundVector** being constructed with the same values:
  - The capacity must be the same and the allocation must be capacity+1.
  - The values are copied sequentially so that the first is put in index 0. The values of head and tail are to be set accordingly (not necessarily the same as the source).
- **move constructor** - receives an existing **RoundVector** (a cbr &&) and initializes the **RoundVector** being constructed with the same values: all values are to be identical including the address stored in vec. The source vec should be set to nullptr.
- **destructor** – frees all existing dynamic allocations within the class.

So as to better understand the behavior of the constructors add in RoundVector the following outputs:

- In the constructor print: "in constructor"
- In the copy constructor print: "in copy constructor"
- In the move constructor print: "in move constructor"
- In the destructor print: "in destructor"

The class must also include the following methods (**public**):

- **addNext** – an insertion method as described above. The method receives a number and inserts it at the tail of the array if the array is not full. If the array is full the following output will be printed: "Vector is full".
- **removeFirst** – a removal method that removes the head element as described above. The method is to return the removed element. **Note**: Assume the array is not empty. At this stage no test will be done to check if the array is empty – such a test will be added after we study exceptions.
- **firstValue** – returns the head element without removing it. **Note**: Assume the array is not empty. At this stage no test will be done to check if the array is empty – such a test will be added after we study exceptions.
- **isEmpty** – a boolean method the returns true if the array contains no elements. See description above.
- **clear** – a method that empties the array. It is not required to modify in any way the contents of the array – just zero the head and tail.
- **print** – a method that prints the elements contained in the array from the head to the tail. Each two elements are to be separated by a space and the printing is to be terminated by an end line.

In the same file where the main program is put, implement a **global function** called **input** that creates and returns a new instance of RoundVector after inputting values into it:

- The parameter for the function is the maximal number of elements to be stored in the array (capacity).
- The function is to print a prompt for entering capacity numbers in sequence. For example, if capacity=8, the function will print: "Enter 8 numbers:".
- The function will input capacity elements from the console and insert them ni the order of input into the RoundVector.
- The function will return the instance of RoundVector. Note that the instance is to be a local instance and the move constructor is meant to be called. If the move constructor is not called, you can manually call it using the **move** function, as such:

```
RoundVector rv(_capacity);
// ...
return move(rv);
```

Use the following main program to test your implementation:

```
#include<iostream>
using namespace std;
#include"RoundVector.h"

enum OPTIONS { STOP, TEST1, TEST2, TEST3 };

int main()
{
	int choice;
	do
	{
		cout << "Enter your choice 0-3:" << endl;
		cin >> choice;
		switch (choice)
		{

		case TEST1:
		{
			cout << "--- Test 1 --- constructor --" << endl;
			RoundVector rv1(4);
			rv1.addNext(10);
			rv1.addNext(11);
			rv1.addNext(12);
			rv1.addNext(13);
			rv1.print();
			rv1.addNext(14);
			rv1.print();
			cout << rv1.firstValue() << endl; //10
			break;

		}

		case TEST2:
		{
			cout << "--- Test 2 --- copy constructor --" << endl;
			RoundVector rv1(4);
			rv1.addNext(10);
			rv1.addNext(11);
			rv1.addNext(12);
			rv1.addNext(13);

			RoundVector rv2(rv1);
			rv2.print();
```

```
					cout << rv2.firstValue() << endl;
					cout << rv2.removeFirst() << endl;
					rv2.print();
					rv2.addNext(14);
					rv2.print();
					rv2.addNext(15);
					rv2.print();
					cout << rv2.removeFirst() << endl;
					cout << rv2.removeFirst() << endl;
					rv2.print();
					rv2.addNext(15);
					rv2.addNext(16);
					rv2.print();
					rv2.addNext(17);
					cout << rv2.firstValue() << endl;
					break;
				}

				case TEST3:
				{
					cout << "--- Test 3 --- move constructor --" << endl;
					RoundVector rv3 = input(6);
					rv3.print();
					rv3.addNext(7);
					cout << rv3.removeFirst() << endl;
					rv3.print();
					cout << rv3.removeFirst() << endl;
					rv3.print();
					cout << rv3.removeFirst() << endl;
					rv3.print();
					cout << rv3.removeFirst() << endl;
					rv3.print();
					cout << rv3.removeFirst() << endl;
					rv3.print();
					cout << rv3.removeFirst() << endl;
					if (!rv3.isEmpty())
						rv3.removeFirst();
					rv3.addNext(7);
					rv3.addNext(8);
					cout << rv3.firstValue() << endl;
					rv3.print();
					rv3.clear();
					rv3.addNext(9);
					rv3.addNext(10);
					rv3.addNext(11);
					cout << rv3.firstValue() << endl;
					rv3.print();
					break;
				}
				default:
					break;
				}
		} while (choice);
}
```

Output example:

```
Enter your choice 0-3:
1
--- Test 1 --- constructor --
in constructor
10 11 12 13
Vector is full
10 11 12 13
10
in destructor
Enter your choice 0-3:
2
--- Test 2 --- copy constructor --
in constructor
in copy constructor
10 11 12 13
10
10
11 12 13
11 12 13 14
Vector is full
11 12 13 14
11
12
13 14
13 14 15 16
Vector is full
13
in destructor
in destructor
Enter your choice 0-3:
3
--- Test 3 --- move constructor --
in constructor
Enter 6 numbers:
1 2 3 4 5 6
in move constructor
in destructor
1 2 3 4 5 6
Vector is full
1
2 3 4 5 6
2
3 4 5 6
3
4 5 6
4
5 6
5
6
6
7
7 8
9
9 10 11
in destructor
Enter your choice 0-3:
0
```

Good

Luck!