CMPT 135: Lab Work Week 3

1. Consider the following C++ class:

```
class A
{
private:
       int v;
public:
    A()
    {
       setValue(0);
    }
    A(int v)
    {
       setValue(v);
    }
    int getValue()
    {
       return v;
    }
    void setValue(int v)
    {
       v = v;
    }
    A subtract (const A &a)
    {
       return A(getValue() - a.getValue());
    }
};
```

a. Is this class definition valid or invalid?
b. If it is invalid, state the error clearly and correct the error so that the class definition is valid.
c. Once you correct the class definition, analyze the following testing main program and give the output as it would appear on the output screen.

```
int main()
{
      A a1, a2(8), a3(2);
      cout << a1.getValue() << ", " << a2.getValue() << ", " << a3.getValue() << endl;
      cout << a1.subtract(a2).getValue() << endl;
      cout << a3.subtract(a2).getValue() << endl;
      return 0;
}
```

2. Write a C++ program that creates a dynamic array of **RationalNumber** objects of user desired size, set the numerator and denominator of each element of the array to some random integer in the range [-5, 5] and finally compute and print the minimum and maximum elements of the array.

3. Write a C++ program that creates a dynamic array of **RationalNumber** objects of user desired size, set the numerator and denominator of each element of the array to some random integer in the range [-5, 5] and finally compute and print the minimum and maximum positive valued elements of the array. If there is no

any positive **RationalNumber** object in the array then your program must instead print the message "No minimum or maximum positive elements".

4. Write a non-member and non-friend function named **sortArray** that takes an array of **RationalNumber** objects and sorts the array using any of the sequential sorting algorithms (insertion sort, bubble sort or selection sort).

Write a main program to test your function. In your program ask the user for the size of an array, create a dynamic array of **RationalNumber** data type with the specified size, fill the array with some random rational numbers of your choice, print the elements of the array, sort the array by calling the **sortArray** function, and finally print the sorted array to see the correctness of your sorting function.

5. Consider the following sequence of **RationalNumber** objects:

r1 = 1/3, r2 = 1/2, r3 = 5/6, r4 = 4/3, r5 = 13/6, r6 = 7/2,...

That is the first element is 1/3, the second element is 1/2 and every element afterwards is the sum of the two elements preceding it. Write a non-member and a non-friend function named **elementAt** that takes an integer argument **n** and returns the **n^{th} RationalNumber** object in the sequence.

For example, the function call **elementAt(1)** must return 1/3, **elementAt(2)** must return 1/2, **elementAt(5)** must return 13/6 etc. Write a test main program to test your function.

6. Consider the sequence given in Question #5 above. Write a non-member and a non-friend function named **elementIndex** that takes a **RationalNumber** object and returns its index in the sequence if the **RationalNumber** object is found in the sequence; otherwise returns -1. For example, the function call **elementIndex(1/3)** must return 1, **elementIndex(1/2)** must return 2, **elementIndex(5/6)** must return 3, **elementIndex(7/2)** must return 6, **elementIndex(3/2)** must return -1, etc.

Write a test main program to test your function.

7. Consider a complex number object as in **3+5i** which you have learned in mathematics. A complex number is represented by two double data type numbers representing the real part and the imaginary part of the complex number. Design a class that represents a complex number. Add all necessary constructors, getters, setters, additional member functions and operators as you see fit. Then write a test main program to see the functionality of your class.

8. In the imperial system of measuring of weight, a **Weight** is represented by two integer values representing pounds and ounces where one pound is equal to 16 ounces. Write a C++ class named **Weight** that represents weight in the imperial system. Have proper constructors, getters, setters, any additional functions and operators (binary arithmetic operators, binary relational operators, unary arithmetic operators and streaming operators). In your unary operators, let the increment and decrement operators increment/decrement the ounce value by 1. Remember for any Weight object at any time, you must keep the value of the pound greater or equal to zero and the value of the ounces between 0 and 15. Then write a test main program to see the functionality of your class.

9. In Linear algebra, a vector in 2D an object with a magnitude and direction and it is represented by a directed straight line from the origin to a point in the 2D plane. Essentially, a vector is described by a Point in a 2D plane. Therefore one way to design a class that represents vectors in 2D is to design the class with only one member variable of type Point.

Write a C++ class named Vector2D that represents vectors in 2D space. Have proper constructors, getters, setters, any additional functions and operators (binary arithmetic operators, binary relational operators, unary arithmetic operators and streaming operators).

Remark:

- For binary arithmetic operators implement only vector addition and subtraction
- For binary relational operators use the length of the vector for comparison. Therefore v1 > v2 is true if and only if v1 is longer than v2; and so on so forth.
- For the unary pre/post increment operators, your overloaded function must increment the magnitude of the calling object by 1 but not alter the direction of the object. For example, if v1 has magnitude 3.2 then ++v1 and v1++ should modify v1 so that its magnitude becomes 4.2. Similarly, for the unary pre/post decrement operators.
- For the streaming operators, print the vector object in a nice format of your choice.

Then write a test main program to see the functionality of your class.

10. Extend the **RationalNumber** class discussed in the lecture by adding the following overloaded operators:

- Binary **subtraction** operator that performs r1 - r2
- Binary **subtraction** operator that performs r1 - **integer**
- Binary **subtraction** operator that performs **integer** - r2
- Binary **multiplication** operator that performs r1 * r2
- Binary **multiplication** operator that performs r1 * **integer**
- Binary **multiplication** operator that performs **integer** * r2
- Binary **division** operator that performs r1 / r2
- Binary **division** operator that performs r1 / **integer**
- Binary **division** operator that performs **integer** / r2
- Binary **compound addition** operator that performs r1 += r2
- Binary **compound addition** operator that performs r1 += **integer**
- Binary **compound subtraction** operator that performs r1 -= r2
- Binary **compound subtraction** operator that performs r1 -= **integer**
- Binary **compound multiplication** operator that performs r1 *= r2
- Binary **compound multiplication** operator that performs r1 *= **integer**
- Binary **compound division** operator that performs r1 /= r2
- Binary **compound division** operator that performs r1 /= **integer**
- Binary **equal** operator that performs r1 == r2
- Binary **equal** operator that performs r1 == **integer**
- Binary **equal** operator that performs **integer** == r2

- Binary **not equal** operator that performs r1 != r2
- Binary **not equal** operator that performs r1 != **integer**
- Binary **not equal** operator that performs **integer** != r2
- Binary **greater than** operator that performs r1 > r2
- Binary **greater than** operator that performs r1 > **integer**
- Binary **greater than** operator that performs **integer** > r2
- Binary **less than** operator that performs r1 < r2
- Binary **less than** operator that performs r1 < **integer**
- Binary **less than** operator that performs **integer** < r2
- Binary **greater or equal** operator that performs r1 >= r2
- Binary **greater or equal** operator that performs r1 >= **integer**
- Binary **greater or equal** operator that performs **integer** >= r2
- Binary **less or equal** operator that performs r1 <= r2
- Binary **less or equal** operator that performs r1 <= **integer**
- Binary **less or equal** operator that performs **integer** <= r2
- Unary operator **pre decrement - -** as in **--r**
- Unary operator **post decrement --** as in **r--**

Write a test main program to test all your overloaded operators.

11. Consider the **RationalNumber** class discussed in the lecture. Add a constructor member function that takes a double data type argument and constructs a **RationalNumber** object whose value is equal to the argument. The declaration of the constructor is given below.

```
RationalNumber(const double& d);
```

Hint:- In this constructor you need to compute the numerator and denominator values from the double data type argument. If you like, you may use the string stream library to do this very easily.

Now, test your class with the following test main program.

```
int main()
{
    RationalNumber r1(0.5), r2(-1.74), r3(5);
    cout << "r1 is " << r1 << endl; //output must be 1/2
    cout << "r2 is " << r2 << endl; //output must be -87/50
    cout << "r3 is " << r3 << endl; //output must be 5/1
    system("Pause");
    return 0;
}
```

Question:- How does the construction of r3 object work after all we don't have a constructor that takes one integer argument?

Answer:- Automatic type casting. It will use the constructor that takes a double data type argument.

12. Now that the **RationalNumber** class has a constructor that takes a double data type argument, consider the following test program and answer the questions that follow:

```
int main()
{
    RationalNumber r1(0.5), r2, r3, r4, r5;
    r2 = r1 + 0.4;
    r3 = 0.4 + r1;
    r4 = r1 + 4;
    r5 = 4 + r1;
    cout << "r1 is " << r1 << endl; //output must be 1/2
    cout << "r2 is " << r2 << endl; //output must be 9/10
    cout << "r3 is " << r3 << endl; //output must be 9/10
    cout << "r4 is " << r4 << endl; //output must be 9/2
    cout << "r5 is " << r5 << endl; //output must be 9/2
    system("Pause");
    return 0;
}
```

Question:- Does this program have any syntax error after all we don't have

`RationalNumber + double`
`double + RationalNumber`

operators overloaded?

Answer:- No. There is no syntax error thanks to automatic casting.

Question:- Does this program have any semantic error?

Answer:- Yes. Because **r2 = r1+0.4;** will be casted to **r2 = r1+0;** because there exists a function that performs **`RationalNumber + int`**. Similarly the statement **r3 = 0.4 + r1;** will be casted to **r3 = 0 + r1;** Hence the outputs will not be as expected mathematically.

So what should we do then? The simple fix will be to change our overloaded operator functions that perform

`RationalNumber + int`
`int + RationalNumber`

to

`RationalNumber + double`
`double + RationalNumber`

making sure to also adjust their implementations to take into account the fact that the parameter(s) is/are now double data type(s). Once we do these adjustments then our class will work flawlessly to add not only `RationaNumber` with `double` but also `RationalNumber` with `int` thanks to automatic type casting.