North South University

Department of Electrical and Computer Engineering

CSE 225L.13 (Data Structures and Algorithms Lab)

Lab 4: Template Class and Operator Overloading

Instructor: Syed Shahir Ahmed Rakin, Arfana Rahman

Objective:

- Learn to work with template classes.
- Recall how operator overloading works

What is a template class:

Template classes are used for creating a single class where variable data types could be used and make progress easier.

A class template starts with the keyword ***template*** followed by template ***parameter(s) inside <>*** which is followed by the class declaration.

For example, the statement below is used for declaring a class template or a function template. Any variables could be used afterward.

```
template <class Temp>
```

After that, you can directly declare template variables in this manner -> `Temp x`

For the methods, write in this manner -> `Class<Temp> :: Class(insert parameters)`

What is operator overloading:

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading. The syntax is given as follows:

```
return_type operator op(parameters) {
    // Implement whatever you want
}
```

Where the return_type is what you want to return, op is which operator you want to overload, and parameters to the operator function.

The following operators cannot be overloaded -> '.' , '::', '.*'

Tasks:

1. (Template Class) This time, modify the header file and the source file given below so that they now work as template classes (the array elements in the dynamically allocated memory can be any type as the user defines)

```
dynarr.h
#ifndef DYNARR_H_INCLUDED
#define DYNARR_H_INCLUDED
class dynArr
{
private:
    int *data;
    int size;
    public:
    dynArr(int);
    ~dynArr();
    void setValue(int, int);
    int getValue(int);
};
#endif // DYNARR_H_INCLUDED
```

```
dynarr.cpp
#include "dynarr.h"
#include <iostream>
using namespace std;
dynArr::dynArr(int s)
{
    data = new int[s];
    size = s;
}
dynArr::~dynArr()
{
    delete [] data;
}
int dynArr::getValue(int index)
{
    return data[index];
}
void   dynArr::setValue(int   index,   int
value)
{
    data[index] = value;
}
```

2. Modify the following class and overload the *(multiplication) and the != (not equal) operators for the class given below.

```
complex.h
#ifndef COMPLEX_H_INCLUDED
#define COMPLEX_H_INCLUDED
class Complex
{
public:
    Complex();
    Complex(double, double);
    Complex operator+(Complex);
    void Print();
private:
    double Real, Imaginary;
};
#endif // COMPLEX_H_INCLUDED
```

```
complex.cpp
#include "complex.h"
#include <iostream>
using namespace std;
Complex::Complex()
{
    Real = 0; Imaginary = 0;
}
Complex::Complex(double r, double i)
{
    Real = r; Imaginary = i;
}
Complex Complex::operator+(Complex a)
{
    Complex t;
    t.Real = Real + a.Real;
    t.Imaginary = Imaginary + a.Imaginary;
    return t;
}
void Complex::Print()
{
    cout << Real << endl;
    cout << Imaginary << endl;
}
```