# Lab No.4 Operator Overloading

## 4.1 Objectives of the lab:

Introducing the concepts of operator overloading and overloading different operators such as

16 Arithmetic operators
17 Relational operators
18 Logical operators
19 Unary operators

## 4.2 Pre-Lab

### 4.2.1 Operator overloading

- Allows to use operators for ADTs
- Most appropriate for math classes e.g. matrix, vector, etc.
- Gives operators class-specific functionality
- Analogous to function overloading -- *operator@* is used as the function name
- Operator functions are not usually called directly
- They are automatically invoked to evaluate the operations they implement
- Assume that + has been overloaded
  actual C++ code becomes
  c1+ c2 + c3 +c4
  The resultant code is very easy to read, write, and maintain

### 4.2.2 Syntax of operator function for binary operators

```
TYPE   operator   OP    (TYPE     rhs)
{
        body of function………
}
```

### 4.2.3 Example code for overloaded operator functions

```
#include <iostream.h>
class complex
{
private:
    double re, im;
public:
    complex():re(0), im(0)
```

```
        {}

        complex(double r, double i):re(r), im(i)
        {}
        void show()
        {
                cout<<"complex number: "<<re<<"+"<<im<<"i"<<endl;
        }

        complex operator + (complex rhs)
        {
                complex temp;
                temp.re=re + rhs.re;
                temp.im=im + rhs.im;
                return temp;
        }
    };

    void main()
    {
        complex     c1(3, 4.3), c2(2, 4);
        c1.show();
        c2.show();
        complex c3;
        c3=c1+ c2;          //Invocation of "+" Operator -- direct
    //or
    //c3=c1.operator+ (c2);   //Invocation of "+" Operator -- Function
        c3.show();
    }
```

### 4.2.4 Syntax of operator function for unary operators

```
        TYPE   operator    OP ()
        {
                body of function………
        }
```

### 4.2.5 Example code for overloaded operator functions

```
class complex
{
private:
        float re, im;
public:

        complex():re(0), im(0)
        {}
        complex(double r, double i):re(r), im(i)
        {}

        void show()
        {
                cout<<"complex number: "<<re<<"+"<<im<<"i"<<endl;
        }

        void operator ++()
        {
                ++re;
                ++im;
        }
};
void main()
{

        complex         c1(3, 4.3), c2;
        c2.set_val(4, 2.3);
        c1.pre_inc();
        c1.show();
        ++c2;
        c2.show();
}
```

#### 4.2.6 How can the above program be modified to allow for a statement like c2=++c1;?

## 4.3 In-Lab

### 4.3.1 Activity

Create a class **RationalNumber** that stores a fraction in its original form (i.e. without finding the equivalent floating pointing result). This class models a fraction by using two data members: an integer for numerator and an integer for denominator. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the numerator and denominator of a fraction to some fixed values.

b) A **two-argument constructor** that initializes the numerator and denominator to the values sent from calling function. This constructor should prevent a 0 denominator in a fraction, reduce or simplify fractions that are not in reduced form, and avoid negative denominators.

c) A **display** function to display a fraction in the format a/b.

d) An overloaded **operator +** for addition of two rational numbers.

Two fractions a/b and c/d are added together as:

$$\frac{a}{b}+\frac{c}{d}=\frac{(a*d)+(b*c)}{b*d}$$

e) An overloaded **operator -** for subtraction of two rational numbers.

Two fractions a/b and c/d are subtracted from each other as:

$$\frac{a}{b}-\frac{c}{d}=\frac{(a*d)-(b*c)}{b*d}$$

f) An overloaded **operator *** for subtraction of two rational numbers.

Two fractions a/b and c/d are multiplied together as:

$$\frac{a}{b}*\frac{c}{d}=\frac{a*c}{b*d}$$

g) An overloaded **operator /** for division of two rational numbers.

If fraction a/b is divided by the fraction c/d, the result is

$$\left.\frac{a}{b}\middle/\frac{c}{d}\right.=\frac{a*d}{b*c}$$

h) Overloaded relational operators

   a. **operator >**: should return a variable of type **bool** to indicate whether 1st fraction is greater than 2nd or not.

   b. **operator <**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than 2nd or not.

   c. **operator >=**: should return a variable of type **bool** to indicate whether 1st fraction is greater than or equal to 2nd or not.

   d. **operator <=**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than or equal to 2nd or not.

i) Overloaded equality operators for **RationalNumber** class

   e. **operator==**: should return a variable of type **bool** to indicate whether 1st fraction is equal to the 2nd fraction or not.

f. **Operator!=:** should a **true** value if both the fractions are not equal and return a **false** if both are equal.

### 4.3.2 Activity

Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:

a) A **no-argument constructor** to initialize hour, minutes, and seconds to 0.

b) A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.

c) A member function **show** to display time in 11:59:59 format.

d) An overloaded **operator+** for addition of two Time objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

e) Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **seconds** unit of time. Keep track that **seconds** should not exceed 60.

f) Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **seconds** unit of time. If number of seconds goes below 0, take appropriate actions to make this value valid.

A **main()** programs should create two initialized **Time** objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third **Time** variable. Finally it should display the value of this third variable. Check the functionalities of ++ and -- operators of this program for both pre- and post-incrementation.

## 4.4 Home-Lab

### 4.3.3 Activity

Create a class called **Distance** containing two members feet and inches. This class represents distance measured in feets and inches. For this class, provide the following functions:

f) A **no-argument constructor** that initializes the data members to some fixed values.

g) A **2-argument constructor** to initialize the values of feet and inches to the values sent from the calling function at the time of creation of an object of type Distance.

h) Overloaded arithmetic operators

a. **operator+** to add two distances: Feet and inches of both objects should add in their corresponding members. 12 inches constitute one feet. Make sure that the result of addition doesn't violate this rule.

b. **operator+=** for addition of two distances.

i) overloaded relational operators

a. **operator >**: should return a variable of type **bool** to indicate whether 1st distance is greater than 2nd or not.

b. **operator <**: should return a variable of type **bool** to indicate whether 1st distance is smaller than 2nd or not.

c. **operator >**=: should return a variable of type **bool** to indicate whether 1st distance is greater than or equal to 2nd or not.

d. **operator <=**: should return a variable of type **bool** to indicate whether 1st distance is smaller than or equal to 2nd or not.

j) Overloaded equality operators

a. **operator==:** should return a variable of type **bool** to indicate whether 1st Distance is equal to the 2nd distance or not.

b. **Operator!=**: should a **true** value if both the distances are not equal and return a **false** if both are equal.

### 4.3.4 Activity

Let us model digital storage. Digital data is stored in the form of bits. 8 bits constitute one byte. Create a class **Storage** that specifies the size of a file. This class has two integer data members: bits and bytes.

g) Provide a **no-argument constructor** to initialize the size of some file to 0 bits and 0 bytes.

h) Provide a **two-argument constructor** to initialize the size of a file to values specified at the time of creation of an object.

i) Provide an overloaded **operator +** that is used to merge two files together in a third file.

j) Provide an overloaded **operator +=** that is used to concatenate one file at the end of the other.

k) Provide overloaded **post-increment and pre-increment** operators to increment the size of a file by one bit. (You must write the functions to accommodate statements like f2=++f1; and f2=f1++; where f1 and f2 are instances of the class Storage)

l) Provide an overloaded **operator >** to determine whether one file is larger in size than the other.

Write a test program to test the functionality of this class.

## 4.5 References:

9 **Class notes**

*10* **Object-Oriented Programming in C++ by** ***Robert Lafore*****(Chapter 8)**

*11* **How to Program C++ by** ***Deitel & Deitel*** **(Chapter 8)**