Lab Sheet 5
Understanding the Concept of Friend Function/Class and Operator Overloading

Friend Function and Class

In some cases you need to access the private data members of a class from non member functions. In such situations you must declare the function as friend function of the class. This friend function seems to violate the data hiding feature of OOP concept. However, the function that accesses private data must be declared as friend function within the class. With friend functions data integrity is still maintained.

Sometimes you may need to make, one or all the member functions of a class friend to other class. For that we declare class or member function as the friend to the other class so that one or all the member functions of the declared class will be friend to the class within which it is declared.

Example:

```
class breadth;

class length

{

      private:

            ......

      public:

            ......

            friend int add(length, breadth);    //friend function
      declaration

};

class breadth

{

      private:

            ......

      public:

            ......

            friend int add(length, breadth);    //friend function
      declaration
```

```
};

int add( length l, breadth b) { }
```

Operator Overloading

We have already studied that we can make user defined data types behave in much the same way as the built-in types. C++ also permits us to use operators with user defined types in the same way as they are applied to the basic types. We need to represent different data items by objects and operations on those objects by operators. Operator can be overloaded for those different operations. Most programmers implicitly use overloaded operators regularly. For example, the addition operator (**+**) operates quite differently on integers, float and doubles and other built-in types because operator (**+**) has been overloaded in the C++ language itself. Operators are overloaded by writing a function definition as you normally would, except that the function name now becomes the keyword **operator** followed by the symbol for the operator being overloaded. Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators for your class.

Syntax

```
<return type> operator <operator_symbol> ( <parameters> )
{
      <statements>;
}

example:

      complex operator + (complex c1,complex c2)
      {
          return complex(c1.real+c2.real,c1.imag+c2.imag);
      }
```

Exercises

1. Write a class for instantiating the objects that represent the two-dimensional Cartesian coordinate system.

   A. make a particular member function of one class to friend function in another class for addition

   B. make other three functions to work as a bridge between the classes for multiplication, division and subtraction.

   C. Also write a small program to demonstrate that all the member functions of one class are the friend functions of another class if the former class is made friend to the latter.

   Make least possible classes to demonstrate all above in single program without conflict.

2. Write a class to store x, y, and z coordinates of a point in three-dimensional space. Using operator overloading, write friend functions to add, and subtract the vectors.

3. Compare the two object that contains integer values that demonstrate the overloading of equality (==), less than (<), greater than (>), not equal (!=),greater than or equal to (>=) and less than or equal to(<=) operators.

4. Write a class **Date** that uses pre increment and post increment operators to add 1 to the day in the Date object, while causing appropriate increments to the month and year (use the appropriate condition for leap year). The pre and post increment operators in your Date class should behave exactly as the built in increment operators.