Creating Header Files Staring to End

```
#ifndef COMPLEX_H
#define COMPLEX_H
class Complex 7
{
 public:
         Complex( double = 0.0, double = 0.0 ); // constructor
         Complex operator+( const Complex & ) const; // addition
 Complex operator-( const Complex & ) const; // subtraction
 void print() const; // output
 private:
 double real; // real part
 double imaginary; // imaginary part
}; // end class Complex
#endif
```

Save it as a Complex.h

For better understanding of Operator Overloading

Operator overloading is a concept of overloading of existing operators, so that they can be used in customized ways. The C++ language uses the keyword **"operator"** for overloading of operators.

Operator overloading is normally applied on "class" data types, as these are user defined types and operators normally do not work with them. By applying overloading of operators, we can make them (operators) work for user defined types.

```
#include<iostream>
using namespace std;

class Rational
{
        private:
                double num, den;
        public:
                void get();
                void show();
                Rational add(Rational);
                Rational operator+(Rational);
};

void Rational::get()
{
        cout << endl << "Enter numberator : ";
        cin >> num;
        cout << endl << "Enter Denominator :";
        den=0;
        while(den == 0)
        {
                cin >> den;
```

```
                if(den==0)
                {
                  cout << endl << "Re-enter (Denominator can not be zero";
                }
        }
}
void Rational::show()
{
        cout << num << "/" << den;
}

Rational Rational::add(Rational arg)
{
        Rational temp;
        temp.num = num * arg.den + den * arg.num;
        temp.den = den * arg.den;
        return temp;
}

Rational Rational::operator+(Rational arg)
{
        Rational temp;
        temp.num = num * arg.den + den * arg.num;
        temp.den = den * arg.den;
        return temp;
}

int main()
{
        Rational obj1, obj2, obj3;
        obj1.get(); obj2.get();
        obj3=obj1 + obj2;
        obj3.show();
}
```

Note: the use of the operator+ in the last section. (The function main will follow below, without it the program will not compile. So add it first).

The example above shows the use and advantage of operator overloading. Suppose we have 3 objects of the class Rational and 2 of them have some values accepted in them, then the addition process of them can be performed as follows:

```
  obj3 = obj1.add(obj2);
```

or

```
obj3 = obj1+obj2;
```

For better understanding of friend Functions/Classes

Occasionally you'll want to allow a function that is not a member of a given class to access the private fields/methods of that class. (This is particularly common in operator overloading.)

We can specify that a given external function gets full access rights by placing the signature of the function inside the class, preceded by the word friend.So, a non-member function can access

the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend.The same happend with classes, a friend class is a class whose members have access to the private or protected members of another class:

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {
         return (width * height);
       }
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a):side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();


  return 0;
}
```