Friend Function
&
Friend Class

Why we need Friend Function?

- The *private* data of a class can't be accessed from the outside of the class.
- But consider the following situation,

  There are two classes *manager* and *scientist*. We would like to use *income_tax()* function to operate on the objects of these classes.
- In such situation C++ allows the common function to be made friendly with both the classes.

Friend function declaration

```
class ABC{
        .........
        .........
        public:
        .........
        .........
        friend void xyz();  //declaration
};
```

- ✓ The function declaration should be preceded by the keyword *friend*.
- ✓ The function is defined elsewhere in the program.
- ✓ The function definition does not use either the keyword *friend* or the scope resolution operator ::.

Program to demonstrate friend function

```
class Alpha{
    int a;
    int b;
public:
    void set();
    friend void add(Alpha ob2);        //add is declared as friend to class Alpha
};
void Alpha :: set(){   //member function set() is defined outside the class
    a=10;
    b=20;
}

void add(Alpha ob2){              //add() is a normal C++ function
    int sum= ob2.a + ob2.b;
    cout<<“Sum=“<<sum;
}

int main(){
    Alpha ob1;
    ob1.set();
    add(ob1);
    return 0;
}
```

Friend Function properties

- It is not in the scope of the class to which it has been declared as friend.
- It can’t be called using the object of that class. Thus has to be invoked like a normal C++ function.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually it takes objects as arguments.

Using member function of one class as friend of other class

```
Class Alpha{
    .....
    int fun1();          // member function of class Alpha
    .....
};
Class Beta{
    .....
    friend int Alpha :: fun1();          // fun1() of Alpha is friend of Beta
    .....
};
```

Here function *fun1()* is a member of *class Alpha* and a **friend** of *class Beta.*

```
class X {
        int a;
public:
        X(int a1) { a = a1; }
        friend void Y :: add(X);

};
```

```
class Y {
        int b;
public:
        Y(int b1) { b = b1; }

        void add (X p) {

        cout<< ( b + p.a)

        }
};
```

```
int main() {

        X x1(5);

        Y y1(10);

        y1.add(x1);
        return 0;

}
```

Member function add() of Y is accessing the private data of X

So X must declare add() of Y as its friend

Friend Class

All the member functions of one class are **friend** functions of another class.

```
class Alpha{
        .....
        int fun1();
        float fun2();                    // member function of class Alpha
        .....
};
```

```
class Beta{
        .....
        friend int Alpha :: fun1();      // fun1() of Alpha is friend of Beta
        friend int Alpha :: fun2();      // fun2() of Alpha is friend of Beta
        .....
};
```

```
class Beta{
        .....
        friend class Alpha; // All member functions of Alpha are friends to Beta
        .....
};
```

Here the *class Alpha* is the **friend class** of *class Beta i.e.* all the member functions of *class Alpha* will be **friend** of *class Beta*.

```
class X {
        int a;
public:
        X(int a1) { a = a1; }
        friend class Y;
};
```

```
class Y {
        int b;
public:
        Y(int b1) { b = b1; }
        void add (X p) {
            cout<< ( b + p.a)
        }
        void sub (X p) {
            cout << ( b - p.a)
        }
};
```

```
int main() {
        X x1(5);
        Y y1(10);
        y1.add(x1);
        y1.sub(x1);
        return 0;
}
```

Member functions add() and sub() of Y is accessing the private data of X

So X must declare add() and sub() of Y as its friends otherwise it has to declare the whole Y class as its friend.

Operator Overloading

Introduction

- ✓It is a mechanism of adding some extra features to the existing operators so that they can act upon objects to treat the class as built-in data type.
- ✓The mechanism of giving special meanings to an existing operator is called *operator overloading*.

```
class Alpha{
            .........
            .........
};

int main(){
Alpha ob1,ob2,ob3;                  // int a,b,c;
ob3 = ob1 + ob2 ;                   // c = a + b;
}
```

- ✓The *semantics* of an operator can be extended but we can't change its *syntax*.

Introduction contd..

- ✓Operator overloading is done by using a special function called *operator function*.
- ✓The general form of operator function is:

```
returntype classname :: operatorop(arglist) {
        Function body
}
```

- ✓return type is the type of value returned by the specified operation.
- ✓op is the operator being overloaded.
- ✓op is preceded by the keyword operator.

- Operator function must either be a member function or a friend function.

	Unary	Binary
Member function	No arg	1 arg
Friend function	1 arg	2 args

Overloading unary minus

Unary minus operation on built-in types:

```
int a=5;
a = -a;
cout << a;          // Displays –5
```

Unary minus operation on objects:

```
class X {
    public:
    int a;
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
};
```

```
int main( ) {
        X x1(5);
        x1.disp();          // 5

        x1.a=-x1.a;

        x1.disp();          // -5
        return 0;
}
```

Using member function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }

    void operator-() {
            a = -a;
    }
};
```

```
int main( ) {
            X x1(5);
            x1.disp();          // 5

            -x1;
            x1.disp();          // -5
            return 0;
}
```

The compiler interprets –x1 as

x1.operator-();

Using member function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    X operator-();
};
X  X :: operator-() {
            X temp;
            temp.a = -a;
            return (temp);
}
```

```
int main( ) {
            X x1(5);
            X x2;
            x1.disp();          // 5

            x2 = -x1;
            x2.disp();          // -5
            return 0;
}
```

The compiler interprets x2=–x1 as

x2 = x1.operator-();

Using friend function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    friend X operator-( X ob);
};
X  operator-( X ob) {
            X temp;
            temp.a = - ob.a;
            return (temp);
}
```

```
int main( ) {
            X x1(5);
            X x2;
            x1.disp();          // 5

            x2 = -x1;
            x2.disp();          // -5
            return 0;
}
```

The compiler interprets x2=–x1 as

x2 = operator-(x1);

Using friend function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    friend void operator-( X ob);
};
void operator-( X ob) {
    ob.a =  - ob.a;
}
void operator-( X &ob) {
    ob.a =  - ob.a;
}
```

```
int main( ) {
        X x1(5);
        x1.disp();          // 5

        -x1;
        x1.disp();          // -5
        return 0;
}
```

Wrong Output 5

Correct Output -5

Overloading binary plus

Binary + operation on built-in types:

```
int a=5,b=10,c;
c = a + b;
cout << c;          // Displays 15
```

Binary + operation on objects:

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
};
```

```
int main( ) {
        X x1(5), x2(10), x3;

        x3.a = x1.a + x2.a;

        x3.disp();          // 15
        return 0;
}
```

Using member function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }

    X operator+( X p) {
            X t;
            t.a = a + p.a;
            return (t);
    }
};
```

```
int main( ) {
            X x1(5),x2(10),x3;

            x3 = x1 + x2;
            x3.disp();          // 15
            return 0;
}
```

- The compiler interprets x3 = x1 + x2 as
  x3 = x1.operator+(x2);
- The first operand always calls the operator function and the second operand is passed as argument

Using friend function

```
class X {
    int a;
    public:
    X() { }
    X(int b) {  a=b;  }
    void disp()  {  cout<<a;  }
    friend X operator+( X p,X q);
};
X operator+( X p,X q) {
    X t;
    t.a = p.a + q.a;
    return (t);
    }
```

```
int main( ) {
        X x1(5),x2(10),x3;

        x3 = x1 + x2;
        x3.disp();          // 15
        return 0;
}
```

- The compiler interprets x3 = x1 + x2 as
  x3 = operator+(x1,x2);

Manipulation of String using operator overloading

- Strings can be defined as class objects which can be manipulated by overloading the operators like =,>=,< etc.

  eg. string3 = string1 + string2;

  if (string1 >= string2) maxstring = string1;

```
class string {
    char *p;
    int len;
public:
    string() { len = 0; p = null; }          // create null string
    string(char *s);                         // create string from arrays
    string(string &s);                       // copy constructor
    ~string() { delete p; }                  // destructor
    void disp() { cout << p; }               // display the string
    friend int operator<=(string &s, string &t);  //overloading <= operator
};
string :: string(char *s) {                  // constructor definition
    len = strlen(s);
    p = new char[len+1];
    strcpy(p,s);
}
string :: string(string &s) {                // constructor definition
    len = s.len;
    p = new char[len+1];
    strcpy(p,s.p);
}
```

```
int operator<=(string &s, string &t){
    int m = strlen(s.p);
    int n = strlen(t.p);

    if (m <= n) return(1);
    else return(0);
}

int main() {
    string s1 = “NIT”;
    string s2 = “ROURKELA”;

    if ( s1 <= s2) cout<<“Second string is longer than First”;
    else cout<<“First string is longer than Second”;
    return 0;
}
```

Some Restrictions

- The following operators can't be overloaded in C++.
  - Class Member Access Operator (. , .*)
  - Scope Resolution Operator (::)
  - Size Operator (sizeof)
  - Conditional Operator (?:)
- Friend function can't be used to overload following operators.
  - Assignment Operator (=)
  - Function Call Operator (())
  - Subscription Operator ([])
  - Class Member Access Operator (->)

Assignments

- WAP to create a Date class, increment the day, month, year of the Date class using operator overloading mechanism .
- WAP to create a class COMPARE having int type data member, compare the objects of COMPARE of class .
- WAP to create a class COMPLEX, perform addition and multiplication operation on COMPLEX class objects,where class consists of real int and imag int as data members and desired memberfunctions.
- WAP to create a class MAT of size m x n and define all possible matrix operations for MAT type objects (operator overloading using friend function)