overloaded ++, -- using friend functions

```
class vector {
      int    x1, x2, x3 ;
      public :
      vector( )   { x1 = 10 ; x2 = 20 ; x3 = 30 ; }
      friend void operator ++(vector &v, int j) ;
      friend void operator ++(vector &v) ;
      void disp( ) { cout << "\n\nx1 ::"<<x1<<" x2::"<<x2<<" x3::"<<x3 ;  }
} ;
void operator ++(vector &v, int j) {
      for(int cntr = 0; cntr < 2; cntr++) {  v.x1++ ; v.x2++ ; v.x3++ ;     }
}
void operator ++(vector &v) {
      v.x1++ ;  v.x2++ ;  v.x3++ ; }

main( ) {
      vector      v1 ;  v1.disp() ;
      v1++ ; v1.disp() ;
      ++v1 ; v1.disp() ;
}
```

overloaded ++, -- using member functions

```
class vector {
      int    x1, x2, x3 ;
      public :
      vector( ) { x1 = 10 ; x2 = 20 ; x3 = 30 ; }
      //postfix form
      void operator ++(int i) {
            for(int cntr = 0; cntr < 2; cntr++) {
                  x1++ ; x2++ ; x3++ ; } }
      //prefix form
      void operator ++( ) { x1++ ;  x2++ ; x3++ ; }

      void disp( ) {
      cout << "\n\nx1 ::"<<x1<<" x2::"<<x2<<" x3::"<<x3 ;}
} ;

main( ) {
      vector      v1 ; v1.disp() ;
      v1++ ; v1.disp() ;
      ++v1 ; v1.disp() ;
}
```

THE CLASS MEMBER ACCESS

- Class member access using -> is considered a unary operator.
- The expression *x -> y* is interpreted as *(x.operator->())(y)* where *x* is a class object.

OVERLOADING THE SUBSCRIPT OPERATOR

- Subscripting is considered a binary operator.
- The expression *x [y]* is interpreted as *x.operator[ ] (y)* where *x* is a class object.

```
class vector {
int   arr[10] ;
public : vector( )    {     for (int i = 0; i < 10; i++)
                            arr[i] = i+1 ; }
friend int operator +(vector v, int x) ;
int operator [ ](int x){    int temp = 1 ;
                    for (int i = 0; i < x; i++)
                            temp = temp * arr[i] ;
                    return temp ;}} ;
int operator +(vector v, int x){
int temp = 1 ;
for (int i = 0; i < x; i++)
        temp = temp * v.arr[i] ;
return temp ;}
main( ) {
        vector      v1 ;
        cout << "\nsubscript result is : " << v1[4] ;
        cout << "\naddition result is : " << v1+3 ;}
```

```
class complex { private : int real, img ;
        public :    complex() {}
                complex(int r, int i) {
                        real = r; img = i ; }
        friend ostream & operator << (ostream & s, complex & c) ;
        friend istream & operator >> (istream & s, complex & c) ; } ;

ostream & operator << (ostream & s, complex & c) {
        s<<"("<<c.real<<","<<c.img<<")"<<endl ;
        return s ; }

istream & operator >> (istream & s, complex & c) {
        s>>c.real>>c.img ;
        return s ; }

void main( ) {
        complex c1, c2(1, 2) ;
        cout<<endl<<"c2 = "<<c2 ;
        cout<<"enter a complex number\n" ;
        cin>>c1 ;
        cout << "c1 = "<<c1 ;
}
```

WHY THE NEED FOR THE FRIEND FUNCTIONS FOR OPERATOR OVERLOADING

- Situations arise where a member function usage for operator overloading does not work.

- Eg :

  $A = 2 + B$ // both A and B are objects.

  This can not be achieved through a member function because the left hand operand is responsible for invoking the member function.

  In this case the left hand operand is not an object.

  In case of friend function both the operands are passed as arguments. Hence it is not compulsory that the first argument has to be object.

OVERLOADING UNARY OPERATORS :

Using Member Functions :

```
class vector {
    int x1, x2, x3 ;
    public :
        vector( ) { x1 = 10 ; x2 = 20 ; x3 = 30 ; }
        void operator –( ) {
        x1 = -x2 ; x2 = -x3 ; x3 = -x1 ;
        }
        void disp( ) {
        cout << “x1 ::”<<x1<<“ x2::”<<x2<<“ x3::”<<x3 ;
        }
} ;
main( ) {
    vector v1 ;
    -v1 ;   // -------> v1.operator -()
    v1.disp( ) ;
}
```

Only the semantics of an operator can be modified, not its syntax.

That is, the rules like the number of operands and precedence rules can not be modified.

Operator overloading is achieved through the use of *operator functions*.

Operator functions are either :

- non-static member functions or
- friend functions

Operator Overloading

This feature provides with the flexibility of creating new definition for most of the C++ operators.

Following operators can not be overloaded :

- class member access operators (. and .*)
- scope resolution operator (::)
- size operator (sizeof)
- conditional operator (?:)

Declaring a class to be a friend also implies that private and protected names from the class granting friendship can be used in the class receiving it. Example :

```
class X  {
     struct sx {} ;
     void f() {cout<<"in class X\n" ;}
     friend class Y ;  } ;
class Y {
public :
     void f_Y() {
          X::sx    sx_obj ;
          X   x_obj ;
          x_obj.f() ;}  } ;
void main() {
     Y   y_obj ;
     y_obj.f_Y() ;
}
```

Declaring a class to be a friend also implies that private and protected names from the class granting friendship can be used in the class receiving it. Example :

```
class X  {
     struct sx {} ;
     void f() {cout<<"in class X\n" ;}
     friend class Y ;  } ;
class Y {
public :
     void f_Y() {
          X::sx    sx_obj ;
          X   x_obj ;
          x_obj.f() ;}  } ;
void main() {
     Y   y_obj ;
     y_obj.f_Y() ;
}
```

Following code shows *friend* declaration referring to an overloaded name :

```
class test {
    int i ;
    friend void f(test k) ;
    friend void f(test k, char c) ;
public :
    test(){i = 10 ;}
} ;

void f(test i, char c) {cout << "in 1st f :: "<<i.i<<"\n" ;}
void f(test i) {cout << "in 2nd f :: "<<i.i<<"\n" ;}

void main() {
    test obj ;
    f(obj) ;
}
```

‘this’ Pointer

‘this’ is a pointer to that very object on which the function is invoked.

Example :

```
class C {
    int     i ;
public :
    C(int i)     { this->i = i ; }
    void disp( ) { cout<<i<<"\n" ; }
} ;

main( ) {
    C   obj(10) ;
    obj.disp() ;
}
```

Pointers

- Pointers to objects can be declared and initialized in just the same way as pointers to ordinary variables.
- They can also be initialized by using the *new* operator.

Example :

```
class  C {
int   i ;
public :
C(int  x) : i(x) { }   C() : i(9) {}
void f( ) { }
} ;
main( ) {
C  *p1 = new C(9) ;
C    o ;
C    *p2 = &o ;
}
```

Following are the advantages of *new* over *malloc* :

- It automatically computes the size of the data object, the use of the *sizeof* operator is not required.

- It automatically returns the correct pointer type, no type casting is required.

- It is possible to initialize the object while creating the memory space.

- As the other operators, *new* and *delete* can also be overloaded.

NEW RETURNS NULL POINTER IN CASE OF FAILURE.

Other Examples :

The following creates an integer variable with initial value 12 ;

```
int      * p1 = new int (12) ;
```

The following creates a memory space for an array of 10 integers :

```
int      *p2 = new int [10] ;
```

Code :

```
class temp {
   public :
    temp( ) {
        cout<<"CONS\n" ; }
    ~ temp( ) {
        cout<<"DESTR\n"; }
} ;
main( ) {
   temp    *p1 = new temp[5] ;
   delete[ ]  p1 ; }
```

Output :

```
CONS
CONS
CONS
CONS
CONS
DESTR
DESTR
DESTR
DESTR
DESTR
```

‘New’ and ‘Delete’ Operators

These are used for dynamic allocation (*new*) and de-allocation (*delete*) of variables and objects.

An object created by *new* is destroyed only when it is *deleted*.

Example :

```
class temp {
    public :
        temp( ) {
        cout << “CONS\n” ;
        }

        ~temp( ) {
        cout<<“DESTR\n” ;
        }
} ;
```

```
main( )
{
    temp   *ptr ;
    ptr = new temp ;
}
```

CONS

```
main( )
{
    temp   *ptr ;
    ptr = new temp ;
    delete(ptr) ;
}
```

CONS
DEST
R

- Can be explicitly called as shown in the following code:

```
class X {
    int x ; char ch ;
public :
    X(){cout<<"constr\n" ;}
    ~X(){cout<<"destr\n" ;}
} ;

main() {
    X   obj ;
    X   *ptr = new X ;

    cout<<"explicit destr invocation\n" ;
    ptr->~X() ;    //delete ptr ;  instead of ptr->~X() will cause further error
    memcpy(ptr, &obj, sizeof(X)) ;
    delete ptr ;
}
```

Constructors and Destructors

Constructors :

- are special functions
- have the same name as the class name
- are invoked automatically every time an object of an associated class is created.
- can not be invoked explicitly
- do not have a return type
- should not be declared in the private section of a class, else objects of that class can not be created
  [there should be at least one public constructor]

- If the conversion is possible to have multiple matches, then the compiler generates an error message.

Example :

Declarations	Function call
long sq(long n)	*sq(10) ;*
double sq(double x)	

This generates an error because int can be converted to both long as well as double.

Declarations	Function call
void f(int x, int y, int z = 80)	*f(2, 20)*
void f(int x, int y)	*f(3, 33, 333)*

The first call generates an error while the second call executes the first function.

Method Overloading

- It allows us to use the same name for different functions.
- It is generally used for functions with similar functionalities.
- It results in **static polymorphism** also known as **early binding**.
- The functions with the same name are identified on the basis of their unique input argument list.

Example :

Let there exist the following function declarations :

int add(int a , int b , int c) ;

int add(int a , int b) ;

Let there be the following function calls :

add(3 , 8) ;// invokes the second version

add(1 , 2 , 3) ; // invokes the first version

Default Arguments

- This mechanism allows invocation of a function without specifying all the arguments.

- This is possible if the function has been declared with default arguments.

  Example :

  float amount (float p, int time, float r = 0.15) ;

  Such a function can be invoked in either of the following ways :

  float val = amount(5000, 7) ;

  float val = amount(3000, 3, 0.18) ;

  In the first case the third argument takes the default value of 0.15

- It is used for providing *outside the class definition* for member functions.

Example :

```
class item {
        int  x , y ;
  public :
        void getdata(int a , int b) ;
        void dispdata( ) ;
} ;

void  item :: getdata(int a , int b)   {
// code for getdata
}

void item :: dispdata( )  {
// code for dispdata
}
```

Scope Resolution Operator

- It can be used to access a global variable from within an inner block even if the inner block tries to hide the global variable

Example :

```
int        m = 10 ;
main( )
{
  int m = 20 ;
  {
       int m = 30 ;
       cout<<"in inner loop\nm="<<m<<"\n" ;
       cout<<"::m="<<::m<<"\n" ;
  }
  cout<<"in outer loop\nm="<<m<<"\n" ;
  cout<<"::m="<<::m<<"\n" ;
}
```

OUTPUT :

```
in inner loop
m=30
::m=10
in outer loop
m=20
::m=10
```

Copy Constructor :

- It takes a reference of an object of the same class as itself as an argument.

- It creates a new object by initializing each member of the new object with the value of the corresponding member of the object passed as argument.

- When no copy constructor is defined within the class, the compiler supplies its own copy constructor.

  Example :

```
class   temp {
public : temp( ) {  cout << "constr\n" ; }
temp (temp &var) { cout << "copy constr\n" ; }} ;
main( ) {
temp t1 , t4 ;
temp      t2(t1) ;   // copy constructor invoked
temp      t3 = t1 ; // copy constructor invoked
t4 = t1 ; }      // copy constructor not invoked
```

The following code displays how a 'pass-by-value' invokes the copy constructor :

```
class temp {
public :        temp() {cout << "constr\n" ;}
    temp(temp &) {cout << "copy constr\n" ;}
    ~temp() {cout << "destr\n" ;} } ;

void f(temp t) {  }
void g(temp &t) {  }


main( ) {
    temp obj ;
     cout<<"invoking f ----\n" ;
    f(obj) ;
     cout<<"invoking g ----\n" ;
    g(obj) ;
}
```

The following code displays how a ‘pass-by-value’ invokes the copy constructor :

```
class temp {
public :        temp() {cout << "constr\n" ;}
    temp(temp &) {cout << "copy constr\n" ;}
    ~temp() {cout << "destr\n" ;} } ;

void f(temp t) {  }
void g(temp &t) {  }


main( ) {
    temp obj ;
     cout<<”invoking f ----\n” ;
    f(obj) ;
     cout<<”invoking g ----\n” ;
    g(obj) ;
}
```

Destructors :

- are special functions like constructors
- have the same name as the class name preceded by a tilde
- can be invoked explicitly
- are invoked implicitly by the compiler upon exit from the program (or block or function); basically it is invoked when the object is no longer accessible
- can not return anything and has no return type
- can not take any input arguments – overloaded destructors are not possible
- should not be declared in the private section of a class

- Can be explicitly called as shown in the following code:

```
class X {
   int x ; char ch ;
public :
   X(){cout<<"constr\n" ;}
   ~X(){cout<<"destr\n" ;}
} ;

main() {
   X  obj ;
   X  *ptr = new X ;

   cout<<"explicit destr invocation\n" ;
   ptr->~X() ;    //delete ptr ;  instead of ptr->~X() will cause further error
   memcpy(ptr, &obj, sizeof(X)) ;
   delete ptr ;
}
```

Friend Functions and Classes

Friend functions are those functions which have access to the private and protected members of a class although they are themselves not members of the class.

Example :

```
class  temp {
      int        i ;
      public :
      temp( ) {i = 10 ;}
      friend  void disp( temp  t) ;
} ;

void  disp( temp  t) {
      cout <<t.i ;
}
```

Some Examples :

Following code shows how an outside function is friendly to a class :

```
class  temp {
      int       i ;
      public :
      temp( ) {i = 10 ;}
      friend  void disp( temp  t) ;
} ;

void  disp( temp  t) {
      cout <<t.i ;
}

main( ) {
      temp    t ;
      disp(t) ;
}
```

Following code shows how a function of one class be friendly to a function of another class :

```
class  X {
    public :
    int   f( ) { Y o ; o.y=9 ;}
} ;

class  Y {int  y ;
    friend  int  X::f( ) ;
} ;
```

If a class (X) declares another class (Y) as a friend class then all the members of the friend class (Y) can access all the members of the class X.

Following code shows how a class can be declared as friendly :

```
class  X {
    friend class Y ;
} ;
```

Following code shows friendship is neither transitive nor inherited:

```
class A {
    int  a ;
    friend class B ;
} ;
class B {
    friend class C ;
} ;
class C {              // NON TRANSITIVE
    public : void f(A  *p) {
        p -> a++ ; }        //error
} ;
class D : public B {        // NON INHERITABLE
    public : void f(A  *p) {
        p -> a++ ; }        //error


} ;
```

OPERATORS THAT *CANNOT* BE OVERLOADED USING FRIENDS

- = Assignment operator
- ( ) Function call operator
- [ ] Subscripting operator
- -> Class member access operator

This is to ensure that their first operands are lvalues.
{Friend functions can take variables, i.e., non-objects as first argument also. This should not happen for these overloaded operators.}

[
An lvalue is an expression referring to an object or function. Originally the term was used to mean 'something that can be on the left hand side of an assignment'
]

OPERATORS THAT *CANNOT* BE OVERLOADED USING FRIENDS

- = Assignment operator
- ( ) Function call operator
- [ ] Subscripting operator
- -> Class member access operator

This is to ensure that their first operands are lvalues.
{Friend functions can take variables, i.e., non-objects as first argument also. This should not happen for these overloaded operators.}

[
An lvalue is an expression referring to an object or function. Originally the term was used to mean 'something that can be on the left hand side of an assignment'
]

Rules for overloading operators :

- Only existing operators can be overloaded.
- The overloaded operator must have at least one user defined type operand.
- The syntax rules of the original operators can not be changed.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- Binary arithmetic operators like '+', '-', '*', '/' must explicitly return a value.

Using Friend Function :

```
class vector {
    int  x1, x2, x3 ;
    public :
        vector( int a, int b, int c) { x1 = a ; x2 = b ; x3 = c ; }
        friend vector operator +( const vector&  v, const vector&
u) ;          void disp( ) {
        cout << "x1 ::"<<x1<<" x2::"<<x2<<" x3::"<<x3 ;
        }
} ;
vector operator +( const vector&  v, const vector& u) {
    return  vector (u.x1-v.x1, u.x2-v.x2, u.x3-v.x3) ;
}
main( ) {
    vector  v1(10, 20, 30) , v(1, 2, 3) ;
    vector v2 = v1+v ;  //  operator+(v1, v)
    v2.disp( ) ;
}
```

OVERLOADING BINARY OPERATORS :

Using Member Functions :

left hand operand :- invokes the operator func

right hand operand :- passed as an argument

```
class vector {
    int  x1, x2, x3 ;
    public :
        vector( int a, int b, int c) { x1 = a ; x2 = b ; x3 = c ; }
        vector operator +( const vector & v) {
        return  vector (2*x1 + v.x1, 2*x2 + v.x2, 2*x3 + v.x3) ;
        }
        void disp( ) {
        cout << "x1 ::"<<x1<<" x2::"<<x2<<" x3::"<<x3 ;
        }
} ;
main( ) {
    vector  v1(10, 20, 30) , v(1, 2, 3) ;
    vector v2 = v1+v ;   //v1.operator+(v)
    v2.disp( ) ;
}
```

overloaded ++, -- using friend functions

```
class vector {
      int    x1, x2, x3 :
      public :
      vector( )   { x1 = 10 ; x2 = 20 ; x3 = 30 ; }
      friend void operator ++(vector &v, int j) ;
      friend void operator ++(vector &v) ;
      void disp( ) { cout << "\n\nx1 ::"<<x1<<" x2::"<<x2<<" x3::"<<x3 ;  }
} ;
void operator ++(vector &v, int j) {
      for(int cntr = 0; cntr < 2; cntr++) {  v.x1++ ; v.x2++ ; v.x3++ ;     }
}
void operator ++(vector &v) {
      v.x1++ ;  v.x2++ ;  v.x3++ ; }

main( ) {
      vector      v1 ;  v1.disp() ;
      v1++ ; v1.disp() ;
      ++v1 ; v1.disp() ;
}
```