Operator Overloading

Operator Overloading

- Operator Overloading is a process of defining additional task to existing operators so that a programmer can use operators with user-defined types as well.
- General syntax :

```
return_type operator op(arglist)
{
        //function body
}
```

Op is the operator being overloaded like +,-,<< etc.
Op is preceded by the keyword **operator**.

- This function is commonly known as **operator function**.

Operator Overloading (Cont...)

- The **operator function** can be either a member function or a friend function (non-member function).
- Differences between using friend function and member function to overload an operator:
  - Friend function will have only one argument for unary operator and two arguments for binary operators.
  - Member function will have no argument for unary operator and only one argument for binary operator.
  - The object used to invoke the member function is passed implicitly to the function and this is not the case with friend function, hence one less argument is required in case of member

Operator Overloading (Cont…)

- Overloaded operator can be invoked like:
- **Unary Operator:**

  Syntax: Op obj ; this would be interpreted as:

  - obj.operator op () in case of member function and
  - operator op(obj) in case of friend function

  Example: -obj ; this would be interpreted as:

  - obj.operator - () in case of member function and
  - operator -(obj) in case of friend function
- **Binary Operator:**

  Syntax: ob1 op ob2; this would be interpreted as:

  - ob1.operator op (ob2) in case of member function and
  - operator op(ob1,ob2) in case of friend function

  Example: obj1 + obj2 ; this would be interpreted as:

  - obj1.operator + (obj2) in case of member function

Overloading unary operator using member function

- A minus operator when used as a unary, takes just one operand.

```
#include<iostream>
using namespace std;
class space
{
   int x;
   int y;
   int z;
   public:
     void set_data(int a, int b, int c);
     void display();
     void operator-(); //overloading unary minus
   };
```

Defining all member functions

```
void space::set_data(int a, int b, int c)
{
  x=a; y=b; z=c;
}
void space::display()
{
        cout<<x<<“ ”<<y<<“ ”<<z<<“\n”;
 }
void space:: operator-()
{     x=-x;
  y=-y;
  z=-z;
}
```

Defining main() function

```
int main()
{
   space s;
   s.set_data(10,-20,30);
   cout<<“S:=”;
   s.display();
   -s;  //compiler see this statement like
   s.operator-();
   cout<<“-S:=“;
   s.display();
   return 0;
}
```

```
S: = 10  -20  30
-S: = -10  20  -30
```

Overloading unary operator using friend function

```
#include<iostream>
using namespace std;
class space
{
   int x;
   int y;
   int z;
   public:
     void get_data(int a, int b,int c);
     void display(void);
     friend void operator-(space &s);//overloading
   unary minus
   };
```

Defining all member functions

```
void space::get_data(int a,int b,int c)
{
  x=a; y=b; z=c;
}
void space::display(void)
{cout<<x<<“ ”<<y<<“ ”<<z<<“\n”;}

void operator-(space &s)
{      s.x=-s.x;
   s.y=-s.y;
  s.z=-s.z;
}
```

Defining main() function

```
int main()
{
   space s;
   s.get_data(10,-20,30);
   cout<<“S:=”;
   s.display();
   -s;  //compiler see this statement like
   operator-(s);
   cout<<“-S:=“;
   s.display();
   return 0;
}
```

Overloading a binary operator

- Binary operator can be overloaded in similar fashion as the unary operator.

```
#include<iostream>
using namespace std;
class complex
{
   float x;
   float y;
 public:
   complex(){}                    //Constructor 1
   complex(float real, float imag)      // Constructor 2
   { x=real; y= imag;}
   complex operator+(complex);     //Overloading + operator
   void display();
};
```

Overloading binary operators(cont...)

```
complex complex::operator +(complex c)
{
    complex temp;
   temp.x=x+c.x;
   temp.y=y+c.y;
    return (temp);
}
void complex:: display()
{
  cout<<x<<“+j”<<y<<“\n”;
}
```

Main() function

```
int main()
{
  complex c1(2.5,3.5); //invokes constructor 2
  complex c2(1.6,2.7); //invokes constructor 2
  complex c3;          //invokes constructor 1
  c3=c1+c2;  //complier sees this statement like this
   c3=c1.operator+(c2);
  cout<<"c1 ="; c1.display();
  cout<<"c2="; c2.display();
  cout<<"c3=";c3.display();
  return 0;
}
```

O/P
c1=2.5+j3.5
c2=1.6+j2.7
c2=4.1+j6.2

Nameless Temporary Object

Re-look this defintion

```
complex complex::operator +(complex c)
{
    complex temp;
   temp.x=x+c.x;
   temp.y=y+c.y;
    return (temp);
}
```

- Here temp object is created , whose sole purpose was to provide a return value for the + operator. This required four statements.
- Same definition with one statement

```
complex complex::operator +(complex c)
{
   return (complex(x+c.x, y+c.y));// here constructor2 will create
   temporary object                          without name
   }
```

Overloading binary a operator using friend function

```
#include<iostream>
using namespace std;
class complex
{
   float x;
   float y;
 public:
   complex(){}              //Constructor 1
   complex(float real, float imag) // Constructor 2
   { x=real; y= imag;}
   friend complex operator+(complex , complex);
   void display(void);
};
```

Overloading binary operators using friend function (Cont...)

```
complex operator +(complex c1, complex c2)
{
    complex temp;
   temp.x=c1.x+c2.x;
   temp.y=c1.y+c2.y;
    return (temp);
}
void complex:: display(void)
{
  cout<<x<<“+j”<<y<<“\n”;
}
```

```
  int main()
{
  complex c1(2.5,3.5); //invokes constructor 2
  complex c2(1.6,2.7); //invokes constructor 2
  complex c3;          //invokes constructor 1
  c3=c1+c2;  //complier sees like c3=operator+(c1,c2);
  cout<<"c1 ="; c1.display();
  cout<<"c2="; c2.display();
  cout<<"c3=";c3.display();
  return 0;
}
```

O/P
c1=2.5+j3.5
c2=1.6+j2.7
c2=4.1+j6.2

- Q.If we get same result in operator overloading by the use of either a friend function or a member function. Why then alternative is made available?

Ans:

Consider a situation where we need one buit-in type operand and one object.

- A=B+2; will work for both cases friend function and member function
- A=2+B;will generate syntax error for member function but works correctly for friend function (Why?)

*

Rules for overloading operators

- Only existing operator can be overloaded, new operators cannot be created.
- Overloaded operator should have at least one operand of user defined data type.
- Precedence and associativity of the operators cannot be changed.
- Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

Problem:Overload ++

- How to differentiate overloading of pre-increment and post-increment operators?
- To make both versions of the increment operator work, we define two overloaded ++ operators,
- For overloading pre-increment ++, usual way of unary operator will be followed.
- Eg. Returnt_type operator++()
- For overloading post-increment C++ developer used concept of **dummy parameter**
- Eg. Return_type operator++(int)
- Note: The only difference is the int in the parentheses. This int isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler to create the postfix version of the operator

*

Cont...

```
//  overloaded ++ operator in both pre-increment and post-increment
#include <iostream>
using namespace std;
class Counter
{  private:
     int count; //count
    public:
    Counter() : count(0)       //constructor no args
    { }
     Counter(int c) : count(c)      //constructor, one arg
     { }
```


*

Cont...

```
int get_count() const //return count
{ return count; }
Counter operator ++ () //increment count (pre-increment)
{
return Counter(++count);    //an unnamed temporary object
}
Counter operator ++ (int)     //increment count (post-increment)
{
return Counter(count++); / /return an unnamed temporary object initialized to
 }                  this count, then increment count
};
```


*

Cont...

```
int main()
{
        Counter c1, c2; //c1=0, c2=0
        cout << "\nc1=" << c1.get_count();    //display
        cout << "\nc2=" << c2.get_count();
        ++c1;       //c1=1
        c2 = ++c1; //c1=2, c2=2 (pre-increment)
        cout << "\nc1=" << c1.get_count();   //display
        cout << "\nc2=" << c2.get_count();
        c2 = c1++;     //c1=3, c2=2 (post-increment)
        cout << "\nc1=" << c1.get_count(); //display again
        cout << "\nc2=" << c2.get_count() << endl;
        return 0;
}
```

Output:
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2

*

Operators that cannot be overloaded

Sizeof	size of operator
.	Membership operator
.*	pointer to member operator
::	scope resolution operator
?:	conditional operator