LECTURE-14

Operator Overloading

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload all the C++ operators except the following:

- Class members access operator (. , .*)
- Scope resolution operator (: :)
- Size operator(sizeof)
- Condition operator (? :)

Although the semantics of an operator can be extended, we can't change its syntax, the grammatical rules that govern its use such as the no of operands precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. When an operator is overloaded, its original meaning is not lost. For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

DEFINING OPERATOR OVERLOADING:

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function called operator function, which describes the task. **Syntax:**

```
return-type class-name:: operator op( arg-list)
                {
                    function body
                }
```

Where return type is the type of value returned by the specified operation and op are the operator being overloaded. The op is preceded by the keyword operator, operator op is the function name. operator functions must be either member function, or friend function. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, this is because the object used to invoke the member function is passed implicitly and therefore is available for the member functions. Arguments may be either by value or by reference. operator functions are declared in. the class using prototypes as follows:

- vector operator + (vector); /./ vector addition
- vector operator-( ); //unary minus
- friend vector operator + (vector, vector); // vector add friend vector
- operator -(vector); // unary minus vector operator - (vector &a); // subtraction
- int operator = =(vector); //comparison
- friend int operator = =(vector ,vector); // comparison

vector is a data type of class and may represent both magnitude and direction or a series of points called elements. The process of overloading involves the following steps:

- Create a class that defines the data type that is used in the overloading operation.
- Declare the operator function operator op() in the public part of the class
- It may be either a member function or friend function.
- Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as **op x or x op**;

for unary operators **x op y**

for binary opearators. **operator op(x);**

for unary operator using friend function

operator op(x,y); for binary operator usinf friend function.

An operator is a symbol that operates on a value to perform specific mathematical or logical computations. They form the foundation of any programming language. In C++, we have built-in operators to provide the required functionality. There are six operators that can be overloaded in C++.

Operator		Types
Unary	++, --	Unary operator
Binary	+,-,*,/,%	Arithmetic
	<,<=,>,=>,==,!=	Relational
	&&, ||, !	Logical
	&, |, <<, >>, ~, ^	Bitwise
	=,+=,-=,*=,/=,%=	Assignment

Example: Using unary operator overloading (member function).

```
#include<iostream>
using namespace std;
class abc
{
        int m,n;
        public:
                abc()
                {
                        m=8;
                        n=9;
                }
                void show()
                {
                        cout<<"m= "<<m<<endl;
                        cout<<"n= "<<n<<endl;
                }
                operator --()
                {
                        --m;
                        --n;
                }
};
main()
{
        abc x;
        x.show();
        --x;
        x.show();}
```

Example: Unary -- operator overloading(using friend function)

```
#include<iostream>
using namespace std;
class abc{
        int m,n;
        public:
                abc()
                {
                        m=8;
                        n=9;
                }
                void show()
                {
                        cout<<"m= "<<m<<endl;
                        cout<<"n= "<<n<<endl;
                }
                friend operator --(abc &p);
};
operator -- (abc &p){
        --p.m;
        --p.n;}
main(){
        abc x;
        x.show();
        operator--(x);
        x.show();}
```

Example: Unary operator + for adding two complex numbers (using member function)

```
#include<iostream>
using namespace std;
class complex
{
	float real,img;
	public:
		complex()
		{
			real=0;
			img=0;
		}
		complex(float r,float i)
		{
			real=r;
			img=i;
		}
		void show()
		{
			cout<<real<<"+i"<<img<<endl;
		}
		complex operator+(complex &p)
		{
			complex w;
			w.real=real+p.real;
			w.img=img+p.img;
			return w;
		}
};
main()
{
	complex s(3,4);
	complex t(4,5);
	complex m;
	m=s+t;
	s.show();
	t.show();
	m.show();
}
```

Example: Unary operator + for adding two complex numbers (using friend function)

```cpp
#include<iostream>
using namespace std;
class complex
{
	float real,img;
	public:
		complex()
		{
			real=0;
			img=0;
		}
		complex(float r,float i)
		{
			real=r;
			img=i;
		}
		void show()
		{
			cout<<real<<"+i"<<img<<endl;
		}
		friend complex operator+(complex &p,complex &q);
};
complex operator+(complex &p,complex &q)
{
	complex w;
	w.real=p.real+q.real;
	w.img=p.img+q.img;
	return w;
}

main()
{
	complex s(3,4);
	complex t(4,5);
	complex m;
	m=s+t;
	s.show();
	t.show();
	m.show();
}
```

Overloading an operator does not change its basic meaning. For example, assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```
class integer
{
intx, y;
        public:
        int  operator + ( ) ;
}
int integer:  : operator + ( )
{
        return   (x-y) ;
}
```

Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But those overloaded by means of a friend function take one reference argument (the object of the relevant class). Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Operator to Overload	Arguments passed to the Member Function	Arguments passed to the Friend Function
Unary Operator	No	1
Binary Operator	1	2

Example:

```
#include<iostream>
using namespace std;
class A
{
	int count;
	public:
		A(int x): count(x){}
		void print()
		{
			cout<<"Count ="<<count<<endl;
		}

		A operator ++()
		{
			++count;
			return A(count);
		}
};
main()
{
	A ob(5);
	++ob;
	ob.print();
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
	int count;
	public:
		A(int x): count(x){}
		void print()
		{
			cout<<"Count ="<<count<<endl;
		}
		A operator ++()
```

```
                {
                        count++;
                        return count;
                }
};

main()
{
        A ob(5);
        ob++;
        ob.print();
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
        int count;
        public:
                A(int x): count(x){}
                void print()
                {
                        cout<<"Count ="<<count<<endl;
                }

                A operator ++()
                {
                        ++count;
                        return A(count);
                }
                A operator ++(int)
                {
                        count++;
                        return A(count);
                }
};
main()
{
        A ob(5);
        ++ob;
        ob++;
        A bb = ++ob;
        ob.print();
        bb.print();
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
        int count;
        public:
                A(int x): count(x){}
                void print()
                {
                        cout<<"Count ="<<count<<endl;
                }

                A operator --()
                {
                        --count;
                        return A(count);
                }
                A operator --(int)
                {
                        count--;
                        return count;
                }
};
main()
{
        A ob(5);
        --ob;
        ob--;
        ob.print();
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
	int val;
	public:
		A(int xx): val(xx){}
		void print()
		{
			cout<<"Value ="<<val<<endl;
		}
		A operator +(A obj)
		{
			int x = val+obj.val;
			return A(x);
		}
};
main()
{
	A ob1(4);
	A ob2(6);
	A ob3 = ob1+ob2;
	ob3.print();
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
	int val_1;
	int val_2;
	public:
		A(int x, int y)
		{
			val_1 = x;
			val_2 = y;
		}
		void print()
```

```
                {
                        cout<<"Value_1="<<val_1<<endl;
                        cout<<"Value_2="<<val_2<<endl;
                }
                A operator +(A obj)
                {
                        int xx = val_1+obj.val_1;
                        int yy = val_2+obj.val_2;
                        return A(xx,yy);
                }
};
main()
{
        A ob1(4,5);
        A ob2(6,5);
        A ob3 = ob1+ob2;
        ob3.print();
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
        public:
                int val_1;
                A(int x): val_1(x) {}
};
void operator<<(ostream& COUT,A& p)
{
        COUT<<p.val_1;
}
main()
{
        A ob(5);
        cout<<ob;
}
```

Example:

```
#include<iostream>
using namespace std;
class A
{
       public:
              int val_1;
              A(int x): val_1(x) {}
};
ostream& operator<<(ostream& COUT,A& p)
{
       COUT<<p.val_1;
}
main()
{
       A ob1(5);
       A ob2(10);

       cout<<ob1<<ob2;
}
```