



# **Programming and Computer Applications-2**

## **Operator Overloading**

**Instructor : PhD, Associate Professor Leyla Muradkhanli**

# Outline

- **Fundamentals of Operator Overloading**
- **Unary Operators Overloading**
- **Binary Operators Overloading**
- **Relational Operators Overloading**
- **Input/Output Operators Overloading**
- **++ and -- Operators Overloading**
- **Assignment Operators Overloading**

# Fundamentals of Operator Overloading

- Operators such as **=**, **+**, and others can be redefined when used with objects of a class
- The name of the function for the overloaded operator is **operator** followed by the operator symbol, *e.g.*,  
**operator+** to overload the **+** operator, and  
**operator=** to overload the **=** operator
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it
- Overloaded operator function definition goes with other member functions

# Fundamentals of Operator Overloading

- Most of C++'s operators can be overloaded.

Operators that can be overloaded							
+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

# Unary Operators Overloading

The unary operators operate on a single operand and following are the examples of Unary operators :

The increment (++) and decrement (--) operators.

The unary minus (-) operator.

The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded.

```
#include <iostream>
using namespace std;
class Numbers {
private:
int x,y,z;
public:
void setnum() {
cout<<"Enter three numbers ";
cout<<"\n x= ";
cin>>x;
cout<<"\n y= ";
cin>>y;
cout<<"\n z= ";
cin>>z;
}
void display() {
cout<<x<<"\t"<<y<<"\t"<<z;
}
```

```
void operator-() {
x=-x;
y=-y;
z=-z;
}
};
int main() {
Numbers num;
num.setnum();
cout<<"\n Numbers are : \n";
num.display();
-num;
cout<<"\n Negative numbers are : ";
num.display();
cout<<endl;
return 0;
}
```

```
Enter three numbers
x= 5
y= 10
z= 17

Numbers are :
5      10      17
Negative numbers are :
-5      -10      -17
Press any key to continue . . .
```

# Binary Operators Overloading

The binary operators take two arguments and following are the examples of Binary operators :

addition (+) operator,  
subtraction (-) operator and  
division (/) operator.

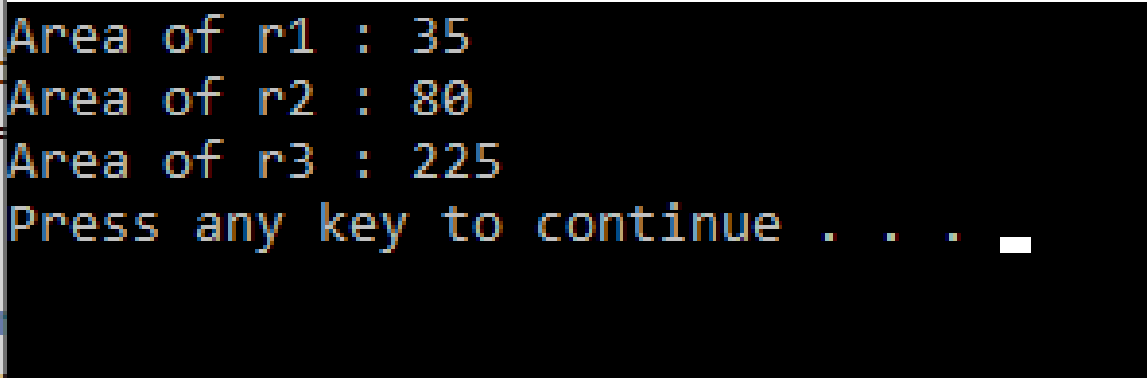
Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.



```
#include <iostream>
using namespace std;
class Rectangle {
private:
int length;
int width;
public:
int Area() {
return length*width;
}
void setlength(int len) {
length=len;
}
void setwidth(int w) {
width=w;
}
```

```
Rectangle operator+(const Rectangle& r) {  
    Rectangle rec;  
    rec.length=this->length+r.length;  
    rec.width=this->width+r.width;  
    return rec;  
}  
};  
  
int main() {  
    Rectangle r1;  
    Rectangle r2;  
    Rectangle r3;  
    r1.setlength(5);  
    r1.setwidth(7);
```

```
r2.setlength(10);  
r2.setwidth(8);  
cout<<"Area of r1 : "<<r1.Area()<<endl;  
cout<<"Area of r2 : "<<r2.Area()<<endl;  
r3=r1+r2;  
cout<<"Area of r3 : "<<r3.Area()<<endl;  
return 0;  
}
```



```
Area of r1 : 35  
Area of r2 : 80  
Area of r3 : 225  
Press any key to continue . . .
```

# Relational Operators Overloading in C++

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

```
#include <iostream>
using namespace std;
class Distance {
    private:
        int kilometer;
        int meter;
    public:
        Distance() {
            kilometer = 0;
            meter = 0;
        }
        Distance(int km, int m) {
            kilometer = km;
            meter = m;
        }
    void displayDistance() {
        cout << " KM: " << kilometer << " M:" << meter << endl;
    }
}
```

```
// overloaded < operator
bool operator <(const Distance& d) {
    if(kilometer < d.kilometer) {
        return true;
    }
    if(kilometer == d.kilometer && meter < d.meter) {
        return true;
    }
    return false;
}

};
```

```
int main() {  
    Distance D1(15, 100), D2(5, 200), D3(15,300);  
    if( D1 < D2 ) {  
        cout << "D1 is less than D2 " << endl;  
    }  
    else {  
        cout << "D2 is less than D1 " << endl;  
    }  
    if(D3<D1) {  
        cout<< "D3 is less than D1 "<<endl;  
    }  
    else {  
        cout<< "D1 is less than D3 "<<endl;  
    }  
    return 0;  
}
```

### Output

D2 is less than D1  
D1 is less than D3

# Input/Output Operators Overloading

C++ is able to input and output the built-in data types using the stream extraction operator `>>` and the stream insertion operator `<<`. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

Following example explains how extraction operator `>>` and insertion operator `<<`.



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

class Distance {
    private:
        int kilometer;
        int meter;

    public:

        Distance() {
            kilometer = 0;
            meter = 0;
        }
}
```

```
Distance(int km, int m) {
    kilometer = km;
    meter = m;
}
friend ostream &operator<<( ostream &output, const Distance &D )
{
    output << " KM : " << D.kilometer << " M : " << D.meter;
    return output;
}

friend istream &operator>>( istream &input, Distance &D ) {
    input >> D.kilometer >> D.meter;
    return input;
}

};
```

```
int main() {  
    Distance D1(50, 900), D2(70, 700), D3;  
  
    cout << "Enter the value of object : " << endl;  
    cin >> D3;  
    cout << "First Distance : " << D1 << endl;  
    cout << "Second Distance : " << D2 << endl;  
    cout << "Third Distance : " << D3 << endl;  
  
    return 0;  
}
```

```
Enter the value of object  
80  
600  
First Distance : KM : 50 M : 900  
Second Distance : KM : 70 M : 700  
Third Distance : KM : 80 M : 600  
Press any key to continue . . .
```

# Overloading Increment ++ and Decrement --

The increment (++) and decrement (--) operators are two important unary operators available in C++.

Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--).

```
#include <iostream>
using namespace std;
class Test
{
    private:
        int i;
    public:
        Test(): i(0) { }
        void operator ++()
        { ++i; }
        void Display()
        { cout << "i=" << i << endl; }
};
int main()
{
    Test obj;
    obj.Display();
    ++obj;
    obj.Display();
    return 0;
}
```

**Output**

i=0

i=1

```
#include <iostream>
using namespace std;
class Test
{
    private:
        int i;
    public:
        Test(): i(5) { }
        // Return type is Test
        Test operator ++()
        {
            Test temp;
            ++i;
            temp.i = i;
            return temp;
        }
        void Display()
        { cout << "i = " << i << endl; }
};
```

```
int main()
{
    Test obj1, obj2;
    obj1.Display();
    obj2.Display();

    obj2 = ++obj1;
    obj1.Display();
    obj2.Display();

    return 0;
}
```

**Output**

i=5

i=5

i=6

i=6

```
#include<iostream>
using namespace std;
class Test {
private :
int i;
public:
Test(): i(5) {}
// overloaded prefix ++ operator
Test operator++() {
Test temp;
++i;
temp.i=i;
return temp;
}
// overloaded postfix ++ operator
Test operator++(int) {
Test temp;
temp.i=i++;
return temp;
}
```



```
void Display() {  
    cout<<"i= " <<i<<endl;}  
};  
int main() {  
    Test obj1, obj2;  
    obj1.Display();  
    obj2.Display();
```

```
    obj2=++obj1;  
    obj1.Display();  
    obj2.Display();
```

```
    obj2=obj1++;  
    obj1.Display();  
    obj2.Display();  
    return 0;  
}
```

## Output

i=5

i=5

i=6

i=6

i=7

i=6

# Assignment Operators Overloading

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

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

class Distance {
    private:
        int kilometer;
        int meter;

    public:
        // Constructors
        Distance() {
            kilometer = 0;
            meter = 0;
        }
}
```

```
Distance(int km, int m) {  
    kilometer = km;  
    meter = m;  
}  
void operator = (const Distance &D ) {  
    kilometer = D.kilometer;  
    meter = D.meter;  
}  
void Display() {  
    cout << " KM: " << kilometer << " M:" << meter << endl;  
}  
};
```

```
int main() {  
    Distance D1(20, 500), D2(50, 800);  
  
    cout << "First Distance : ";  
    D1.Display();  
    cout << "Second Distance :";  
    D2.Display();  
  
    // use assignment operator  
    D1 = D2;  
    cout << "First Distance :";  
    D1.Display();  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

First Distance : KM: 20 M:500

Second Distance :KM: 50 M:800

First Distance :KM: 50 M:800