C++

Operator-Overloading

1. Introduction - I

- **The utility of operators such as +, =, *, /, >, <, and so on is predefined in any programming language.**
- **Programmers can use them directly on built-in data types to write their programs.**
- **However, these operators do not work for user-defined types such as objects.**
- **Therefore, C++ allows programmers to redefine the meaning of operators when they operate on class objects.**
- **This feature is called operator overloading.**

1. Introduction - II

- **Like function overloading, operator overloading is also a form of compile time polymorphism.**

- **Operator overloading is, therefore, less commonly known as operator ad hoc polymorphism .**

- **Since different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.**

1. Introduction - III

- **Operator overloading enables programmers to use notation closer to the target domain.**

- **For example, to add two matrices, we can simply write M1 + M2 , rather than writing M1.add(M2).**

- **With operator overloading, a similar level of syntactic support is provided to user-defined types as provided to the built-in types.**

1. Introduction - IV

- **In scientific computing where computational representation of mathematical objects is required, operator overloading provides great ease to understand the concept.**

- **Operator overloading makes the program clear.**

- **For example, the statement div(mul(M1, M2), add(M1,M2)); can be better written as M1 * M2 / M1+M2**

1. Operator Overloading - Syntax

```
class class_name
{--------
    public:
        return_type operator op(arguments if any)
        {   ---------
            //Function Body
            ----------
        }
        ----------
};
```

```
Complex operator +(Complex &c2)
{       Complex Temp;
        Temp.real = real + c2.real;
        Temp.imag = imag + c2.imag;
        return Temp;
}
```

1. Operator Overloading - Exception I

- **The exceptional operators that cannot be overloaded are as follows:**
- **Scope resolution operator (::)**
- **Member selection operator (.)**
- **Member selection through a pointer to a function (.*)**
- **Ternary operator (?:)**

1. Operator Overloading - Exception II

- **Operator overloading cannot change the operation performed by an operator.**
- **The two operators—the assignment operator (=) and the address operator (&) —need not be overloaded.**
- **Operator overloading cannot alter the precedence and the associativity of operators.**
- **New operators such as like **, <>, |&, and so on cannot be created.**
- **When operators such as &&, ||, and, are overloaded, they lose their special properties of short-circuit evaluation and sequencing.**
- **Overloaded operators cannot have default arguments.**

1. Operator Overloading - Exception III

- **All overloaded operators except the assignment operator are inherited by derived classes.**

- **Number of operands cannot be changed.**

- **Overloading an operator that is not associated with the scope of a class is not permissible.**

1. Operator Overloading - Implementation

Member function	Friend function
• Number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand. • Unary operators take no explicit parameters. • Binary operators take one explicit parameter. • Left-hand operand has to be the calling object. • `obj2 = obj1 + 10;` is permissible but `obj2 = 10 + obj1;` is not permissible.	• Number of explicit parameters is more. • Unary operators take only one parameter. • Binary operators take two parameters. • Left-hand operand need not be an object of the class. • `obj2 = obj1 + 10;` as well as `obj2 = 10 + obj1;` is permissible.

1. Operator Overloading - Implementation

```
return_type operator op ()
```

```
#include<iostream.h>
class Number
{   private:
        int x;
    public:
        Number()
        {   x = 0;      };
        Number(int n)  //parameterized constructor
        {   x = n;      }
        void operator  () // operator overloaded function
        {   x =  x;}
        void show data()
        {   cout<<"\n x - "<<x;       }
};
main()
{   Number N(7);   // create object
    N.show_data();
    -N;            // invoke operator overloaded function
    N.show_data();
}
```

OUTPUT

```
x = 7
x = -7
```

1. Operator Overloading - Implementation

```
#include<iostream.h>
class Number
{   private:
        int x;
    public:
        Number()
        {   x = 0;     };
        Number(int n)
        {   x = n;     }
        Number operator -()       // operator overloading function returns an object
        {       Number temp;
            temp.x = -x;
            return temp;       //object returned
        }
        void show_data()
        {   cout<<"\n x = "<<x;       }
};
main()
{       Number N1(-10), N2;
        N2 = -N1;     // return value assigned to another object
        N2.show_data();
}
```

1. Operator Overloading - Using Friend Function

The function will take one operand as an argument.

This operand will be an object of the class.

The function will use the private members of the class only with the object name.

The function may take the object by using value or by reference.

The function may or may not return any value.

The friend function does not have access to the this pointer.

1. Operator Overloading - Using Friend Function

```
#include<iostream.h>
class Number
{   private:
        int x;
    public:
        Number()
        {     x = 0;     };
        Number(int n)
        {     x = n;     }
        void show_data()
        {     cout<<"\n x = "<<x;     }
        friend Number & operator-(Number &);     //friend function declared
};
Number & operator -(Number &N)
{   N.x = -N.x;          //use objectname with data member
    return N;           //return object
}
main()
{   Number N1(100), N2;
    N2 = -N1;     // overloaded operator function called
    N2.show_data();
}
```

OUTPUT

```
x = -100
```