# Operator Overloading in C++

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.  
Example:

       int a;  
      float b,sum;  
      sum=a+b;

Here, variables “a” and “b” are of types “int” and “float”, which are built-in data types. Hence the addition operator ‘+’ can easily add the contents of “a” and “b”. This is because the addition operator “+” is predefined to add variables of built-in data type only.

Now, consider another example

class A  
{

};

int main()  
{  
      A   a1,a2,a3;

      a3= a1 + a2;

      return 0;  
}

In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

## Operators that can be overloaded

We can overload

* Unary operators
* Binary operators
* Special operators ( [ ], () etc)

But, among them, there are some operators that cannot be overloaded. They are

* Scope operator (::)
* Sizeof
* member selector(.)
* member pointer selector(\*)
* ternary operator(?:)

Operators that can be overloaded

1. Binary Arithmetic     ->     +, -, \*, /, %
2. Unary Arithmetic     ->     +, -, ++, —
3. Assignment     ->     =, +=,\*=, /=,-=, %=
4. Bit- wise      ->     & , | , << , >> , ~ , ^
5. De-referencing     ->     (->)
6. Dynamic memory allocation and De-allocation     ->     New, delete
7. Subscript     ->     [ ]
8. Function call     ->     ()
9. Logical      ->     &,  | |, !
10. Relational     ->     >, < , = =, <=, >=

## **Syntax of Operator Overloading**

1. return\_type class\_name  : : operator op(argument\_list)
2. {
3. // body of the function.
4. }

Where the **return type** is the type of value returned by the function.

**class\_name** is the name of the class.

**operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

## **Rules for Operator Overloading**

* Existing operators can only be overloaded, but the new operators cannot be overloaded.
* The overloaded operator contains atleast one operand of the user-defined data type.
* We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
* Operator cannot be used to overload when declaring that function as friend function **=** **()** **[]** **->**.
* When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
* When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

#include<iostream>

using namespace std;

class Complex {

private:

int real, imag;

public:

Complex(int r = 0, int i = 0) {real = r;   imag = i;}

// This is automatically called when '+' is used with

// between two Complex objects

Complex operator + (Complex obj) {

Complex res;

res.real = real + obj.real;

res.imag = imag + obj.imag;

return res;

}

void print() { cout << real << " + i" << imag << '\n'; }

};

int main()

{

Complex c1(10, 5), c2(2, 4);

Complex c3 = c1 + c2;

c3.print();

}

#include<iostream>

class Numbers

{

int x, y, z;

public:

void accept()

{

cout<<"\n Enter Three Numbers";

cout<<"\n --------------------------";

cout<<"\n First Number : ";

cin>>x;

cout<<"\n Second Number : ";

cin>>y;

cout<<"\n Three Number : ";

cin>>z;

cout<<"\n --------------------------";

}

void display()

{

cout<<" ";

cout<<x<<"\t"<<y<<"\t"<<z;

}

void operator-()

{

x=-x;

y=-y;

z=-z;

}

};

int main()

{

Numbers num;

num.accept();

cout<<"\n Numbers are :\n\n";

num.display();

-num; //Overloaded Unary (-) Operator

cout<<"\n\n Negated Numbers are :\n\n";

num.display();

return 0;

}

## *Overloading binary minus operator - using friend function*

* as you know, As you know, the minus operator **-** when applied to any built-in type variable such as int, float, double, long will change its value from positive to negative. We can even change the sign of values of an object by using the unary **-** operator using **member function**. But let's see how to achieve the same using a *non-member* **friend function**.

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**private:**

**int a;**

**public:**

**void set\_a();**

**void get\_a();**

**friend A operator -(A); *// Friend function which takes an object of A and return an object of A type.***

**};**

***//Definition of set\_a() function***

**void A :: set\_a()**

**{**

**a = 10;**

**}**

***//Definition of get\_a() function***

**void A :: get\_a()**

**{**

**cout<< a <<"\n";**

**}**

***//Definition of overloaded unary minus operator - friend function***

**A operator -(A ob)**

**{**

**ob.a = -(ob.a);**

**return ob;**

**}**

**int main()**

**{**

**A ob;**

**ob.set\_a();**

**cout<<"The value of a is : ";**

**ob.get\_a();**

***//Calling operator overloaded function - to negate the value***

**ob = -ob; *//ob object is passed as an argument to the friend function and its negative version is returned.***

**cout<<"The value of a after calling operator overloading friend function - is : ";**

**ob.get\_a();**

**}**