Operator Overloading in C

GeeksforGeeks

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.

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. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

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.

Example:

C++

C++

```
class A {

    statements;

};

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.

Now, if the user wants to make the operator “+” add two class objects, the user has to redefine the meaning of the “+” operator such that it adds two class objects. This is done by using the concept of “Operator overloading”. So the main idea behind “Operator overloading” is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

Example:

C++

C++

```
#include <iostream>

using namespace std;

class Complex {

private:

    int real, imag;

public:

    Complex(int r = 0, int i = 0)

    {

        real = r;

        imag = i;

    }

    Complex operator+(Complex const& 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();
}
```

What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.
Example:

C++

C++

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
    void print() { cout << real << " + i" << imag << endl; }
```

```
    friend Complex operator+(Complex const& c1,
                             Complex const& c2);
};
Complex operator+(Complex const& c1, Complex const& c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3
        = c1
          + c2;
    c3.print();
    return 0;
}
```

Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member operator))
Ternary or conditional (?:)

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 resolution operator (: 🙂

Member selection operator

**Member selection through *

Pointer to a member variable

Conditional operator (? 🙂

Sizeof operator sizeof()

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, —
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[ ]
Function call	()
Logical	&, | |, !
Relational	>, < , = =, <=, >=

Why can't the above-stated operators be overloaded?

1. sizeof – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

2. typeid: This provides a CPP program with the ability to recover the actually derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type 'look' like another type, polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

3. Scope resolution (::): This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this

operator.

4. Class member access operators (.(dot), .* (pointer to member operator)): The importance and implicit use of class member access operators can be understood through the following example:

Example:

C++

C++

```
#include <iostream>

using namespace std;

class ComplexNumber {

private:

    int real;

    int imaginary;

public:

    ComplexNumber(int real, int imaginary)

    {

        this->real = real;

        this->imaginary = imaginary;

    }

    void print() { cout << real << " + i" << imaginary; }

    ComplexNumber operator+(ComplexNumber c2)

    {

        ComplexNumber c3(0, 0);

        c3.real = this->real + c2.real;

        c3.imaginary = this->imaginary + c2.imaginary;

        return c3;

    }

};

int main()
```

```
{
    ComplexNumber c1(3, 5);
    ComplexNumber c2(2, 4);
    ComplexNumber c3 = c1 + c2;
    c3.print();
    return 0;
}
```

Explanation:

The statement ComplexNumber c3 = c1 + c2; is internally translated as ComplexNumber c3 = c1.operator+ (c2); in order to invoke the operator function. The argument c1 is implicitly passed using the **'.'** operator. The next statement also makes use of the dot operator to access the member function print and pass c3 as an argument.

Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

5. Ternary or conditional (?:): The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

conditional statement ? expression1 (if statement is TRUE) : expression2 (else)

A function overloading the ternary operator for a class say ABC using the definition

ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);

would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.

Important points about operator overloading

1) For operator overloading to work, at least one of the operands must be a user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Example:

C++

C++

```
#include <iostream>

using namespace std;

class Fraction {

private:

    int num, den;

public:

    Fraction(int n, int d)

    {

        num = n;

        den = d;

    }

    operator float() const

    {

        return float(num) / float(den);

    }

};

int main()

{

    Fraction f(2, 5);

    float val = f;

    cout << val << '\n';

    return 0;

}
```

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

Example:

C++

C++

```
#include <iostream>

using namespace std;

class Point {

private:

    int x, y;

public:

    Point(int i = 0, int j = 0)

    {

        x = i;

        y = j;

    }

    void print()

    {

        cout << "x = " << x << ", y = " << y << '\n';

    }

};

int main()

{

    Point t(20, 20);

    t.print();

    t = 30;

    t.print();

    return 0;

}
```

Output

x = 20, y = 20

x = 30, y = 0

Quiz on Operator Overloading