C++ Operator Overloading

programiz.com/cpp-programming/operator-overloading

Join our newsletter for the latest updates.

In this tutorial, we will learn about operator overloading with the help of examples.

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,

Suppose we have created three objects *c1*, *c2* and *result* from a class named `Complex` that represents complex numbers.

Since operator overloading allows us to change how operators work, we can redefine how the `+` operator works and use it to add the complex numbers of *c1* and *c2* by writing the following code:

```
result = c1 + c2;
```

instead of something like

```
result = c1.addNumbers(c2);
```

This makes our code intuitive and easy to understand.

Note: We cannot use operator overloading for fundamental data types like `int`, `float`, `char` and so on.

Syntax for C++ Operator Overloading

To overload an operator, we use a special `operator` function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Here,

- `returnType` is the return type of the function.
- `operator` is a keyword.
- `symbol` is the operator we want to overload. Like: `+`, `<`, `-`, `++`, etc.
- `arguments` is the arguments passed to the function.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator `++` and decrement operator `--` are examples of unary operators.

Example1: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output

```
Count: 6
```

Here, when we use `++count1;` , the `void operator ++ ()` is called. This increases the *value* attribute for the object *count1* by 1.

Note: When we overload operators, we can use it to work in any way we like. For example, we could have used `++` to increase *value* by 100.

However, this makes our code confusing and difficult to understand. It's our job as a programmer to use operator overloading properly and in a consistent and intuitive way.

The above example works only when `++` is used as a prefix. To make `++` work as a postfix we use this syntax.

```
void operator ++ (int) {
    // code
}
```

Notice the `int` inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

Example 2: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix and postfix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}


    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }


    // Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output

```
Count: 6
Count: 7
```

The **Example 2** works when `++` is used as both prefix and postfix. However, it doesn't work if we try to do something like this:

```
Count count1, result;

// Error
result = ++count1;
```

This is because the return type of our operator function is `void`. We can solve this problem by making `Count` as the return type of the operator function.

```
// return Count when ++ used as prefix

Count operator ++ () {
    // code
}

// return Count when ++ used as postfix

Count operator ++ (int) {
   // code
}
```

Example 3: Return Value from Operator Function (++ Operator)

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

class Count {
   private:
    int value;

   public
       :
    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    Count operator ++ () {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = ++value;

        return temp;
    }

    // Overload ++ when used as postfix
    Count operator ++ (int) {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = value++;

        return temp;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1, result;

    // Call the "Count operator ++ ()" function
    result = ++count1;
    result.display();

    // Call the "Count operator ++ (int)" function
    result = count1++;
    result.display();

    return 0;
}
```

Output

```
Count: 6
Count: 6
```

Here, we have used the following code for prefix operator overloading:

```
// Overload ++ when used as prefix
Count operator ++ () {
    Count temp;

    // Here, value is the value attribute of the calling object
    temp.value = ++value;

    return temp;
}
```

The code for the postfix operator overloading is also similar. Notice that we have created an object *temp* and returned its value to the operator function.

Also, notice the code

```
temp.value = ++value;
```

The variable *value* belongs to the *count1* object in `main()` because *count1* is calling the function, while *temp.value* belongs to the *temp* object.

Operator Overloading in Binary Operators

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, `+` is a binary operator that works on the operands *num* and `9`.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the *obj1* object and *obj2* is passed as an argument to the function.

Example 4: C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +
// This program adds two complex numbers

#include <iostream>
using namespace std;

class Complex {
   private:
    float real;
    float imag;

   public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
    }

    // Overload the + operator
    Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }

    void output() {
        if (imag < 0)
            cout << "Output Complex number: " << real << imag << "i";
        else
            cout << "Output Complex number: " << real << "+" << imag << "i";
    }
};

int main() {
    Complex complex1, complex2, result;

    cout << "Enter first complex number:\n";
    complex1.input();

    cout << "Enter second complex number:\n";
    complex2.input();

   // complex1 calls the operator function
   // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();

    return 0;
}
```

Output

```
Enter first complex number:
Enter real and imaginary parts respectively: 9 5
Enter second complex number:
Enter real and imaginary parts respectively: 7 6
Output Complex number: 16+11i
```

In this program, the operator function is:

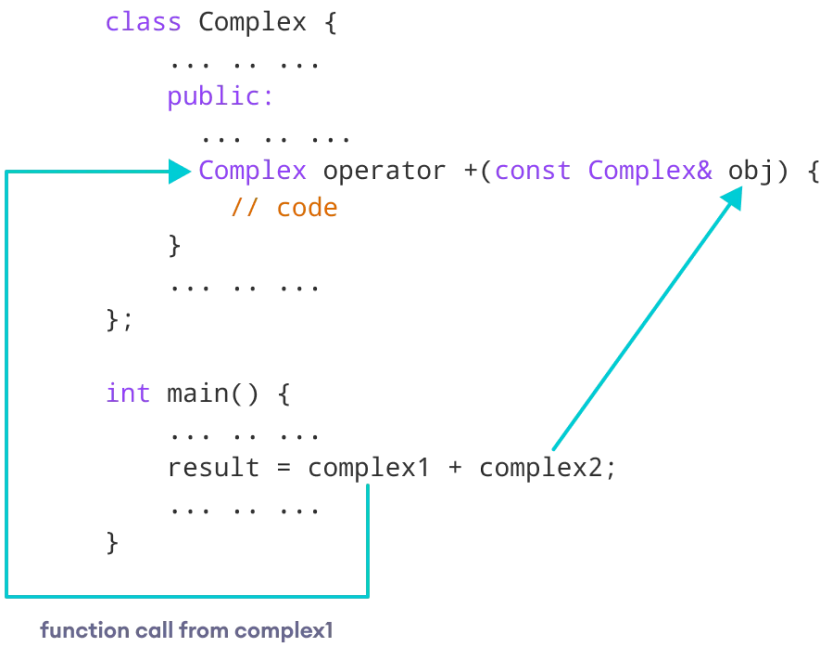
```
Complex operator + (const Complex& obj) {
    // code
}
```

Instead of this, we also could have written this function like:

```
Complex operator + (Complex obj) {
    // code
}
```

However,

- using `&` makes our code efficient by referencing the *complex2* object instead of making a duplicate object inside the operator function.
- using `const` is considered a good practice because it prevents the operator function from modifying *complex2*.



Overloading binary operators in C++

Things to Remember in C++ Operator Overloading

1. Two operators `=` and `&` are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the `=` operator. We do not need to create an operator function.

2. Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.
3. There are 4 operators that cannot be overloaded in C++. They are:
    a. `::` (scope resolution)
    b. `.` (member selection)
    c. `.*` (member selection through pointer to function)
    d. `?:` (ternary operator)

Visit these pages to learn more on: