Operator Overloading in C++

Operator overloading allows C++ operators to be redefined and used in different ways, depending on their arguments. It enables you to define the meaning of operators for user-defined types (like classes).

Why Use Operator Overloading?

- To provide intuitive syntax for user-defined types
- To enhance code readability and maintainability
- To mimic built-in types behavior with custom types

Syntax

The syntax for operator overloading involves defining a function with the operator keyword followed by the operator symbol. This function can be a member function of a class or a non-member function (often a friend function).

```
// Member function syntax
returnType ClassName::operator symbol (parameters) {
  // Implementation
}

// Non-member function syntax
returnType operator symbol (parameters) {
  // Implementation
}
```

- **returnType:** Specifies the type of value returned by the overloaded operator.
- **ClassName:** The class for which the operator is being overloaded (for member functions).
- **symbol:** The operator to be overloaded (e.g., +, -, *, /, ==, !=, <, >, etc.).
- **parameters:** The operands of the operator. The number and type of parameters depend on whether the operator is unary or binary and whether it is a member or non-member function.

Semantics

The semantics of operator overloading involve defining the specific behavior of the operator for the given class. It's crucial to maintain the expected logical behavior of the operator to avoid confusion and ensure code clarity.

- **Unary Operators:** When overloading unary operators (e.g., ++, --, -), member functions take no arguments, while non-member functions take one argument (the object on which the operator is applied).
- **Binary Operators:** When overloading binary operators (e.g., +, -, *, /), member functions take one argument (the right-hand operand), while non-member functions take two arguments (both operands).
- **Assignment Operator (=):** Should be overloaded as a member function and handle self-assignment to prevent errors.
- **Function Call Operator (()):** Must be a member function.
- **Subscript Operator ([]):** Typically overloaded for container-like classes.
- **Increment and Decrement Operators:** Have prefix and postfix versions that can be distinguished using a dummy int argument in the postfix version.

Rules of Operator Overloading

- Not all operators can be overloaded (e.g., ::, ., .*, ?:).
- The precedence and associativity of operators cannot be changed.
- At least one operand in an overloaded operator must be of user-defined type.
- Overloading cannot be done for built-in types only.
- Operators =, [], (), and -> must be overloaded as member functions.
- It is recommended to overload operators in a way that aligns with their conventional meaning to maintain code readability and avoid unexpected behavior.

Types of Operators that Can Be Overloaded

- Arithmetic Operators: +, -, *, /, %
- Comparison Operators: ==, !=, <, >, <=, >=
- Logical Operators: &&, ||, !
- Bitwise Operators: &, |, ^, ~, <<, >>
- Assignment Operators: =, +=, -=, etc.
- Increment and Decrement: ++, --
- Function Call Operator: ()
- Subscript Operator: []
- Pointer Operator: ->

Example 1: Overloading + Operator

This example shows how to overload the '+' operator for a class representing a complex number.

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

class Complex {
  float real, imag;
public:
  Complex(float r = 0, float i = 0) : real(r), imag(i) {}

  // Overloading '+' operator
  Complex operator + (const Complex& obj) {
    return Complex(real + obj.real, imag + obj.imag);
  }

  void display() {
    cout << real << " + " << imag << "i" << endl;
  }
};

int main() {
  Complex c1(3.5, 2.5), c2(1.5, 4.5);
  Complex c3 = c1 + c2;
  c3.display();
  return 0;
}
```

NOTE

In C++, operator overloading is usually done within a class, but you *can* overload operators **outside** the class as well — typically as **non-member functions** (often friend functions if they need access to private members).

Your code currently shows overloading of the + operator **inside** the Complex class. If you want to overload the + operator **without doing it inside the class**, here's how to do it:

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

class Complex {
    float real, imag;

public:
    Complex(float r = 0, float i = 0) : real(r), imag(i) {}

    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }

    // Give access to private members for non-member function
    friend Complex operator + (const Complex& c1, const Complex& c2);
};

// Overload '+' outside the class
Complex operator + (const Complex& c1, const Complex& c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main() {
    Complex a(2.5, 3.5), b(1.5, 2.5);
    Complex result = a + b;

    result.display(); // Output: 4 + 6i

    return 0;
}
```

Overloaded operator Calling syntax:

- Operator+(a,b) //outside class function
- a.operator+(b) //class method invoked by object
- a+b

```
#include <iostream>

using namespace std;

class Complex {

public:

   float real, imag;

   Complex(float r = 0, float i = 0) : real(r), imag(i) {}

   void display() const {

      cout << real << " + " << imag << "i" << endl;

   }

   // Give access to private members for non-member function

   //friend Complex operator + (const Complex& c1, const Complex& c2);

};

// Overload '+' outside the class

Complex operator + (const Complex& c1, const Complex& c2) {

   return Complex(c1.real + c2.real, c1.imag + c2.imag);

}

int main() {

   Complex a(2.5, 3.5), b(1.5, 2.5);

   Complex result = a + b;

   result.display(); // Output: 4 + 6i

   return 0;

}
```