



Himalaya College of Engineering

Advanced C++ Programming Lab Report

Lab 5: OPERATOR OVERLOADING

Prepared By: Ashutosh Thapa(HCE081BEI010)

Subject: Object-Oriented Programming (OOP)

Program: Bachelor of Electronics Engineering

Institution: Himalaya College of Engineering

Date: June,2025

Lab 5: Operator Overloading

Objective:

- To understand the concept of operator overloading in C++.
- To learn how to overload operators to work with user-defined types.
- To demonstrate the syntax and rules of operator overloading.
- To apply both member and friend functions in operator overloading.

Theory:

1. Introduction to Operator Overloading :

Operator overloading in C++ allows you to define custom behavior for operators (like +, -, etc.) when used with user-defined types. This makes the code more intuitive and easier to read. It is useful for operations like arithmetic, comparisons, and stream input/output.

Syntax ::

```
returnType operator op (parameters) {  
    // function body  
}
```

2. Overloading Operators Using Member Functions:

You can overload an operator inside the class. For binary operators, one operand is the current object.

Eg:

```
class Complex {  
private:  
    int real, imag;  
public:  
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}  
    Complex operator + (const Complex& obj) {  
        return Complex(real + obj.real, imag + obj.imag);  
    }  
};
```

3. Overloading Operators Using Friend Functions:

When using a friend function, it is defined outside the class but can still access private members. This is useful when operands are of different types.

Eg:

```
class Complex {  
private:  
    int real, imag;  
public:  
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}  
    friend Complex operator + (const Complex& obj1, const Complex& obj2);  
};  
Complex operator + (const Complex& obj1, const Complex& obj2) {  
    return Complex(obj1.real + obj2.real, obj1.imag + obj2.imag);  
}
```

4. Operators That Can Be Overloaded

- Arithmetic operators: +, -, *, /, %
- Relational operators: ==, !=, >, <, >=, <=
- Assignment operators: =, +=, -=, etc.
- Unary operators: ++, --, - (unary), !
- Stream operators: <<, >> (must be friend functions)

5. Operators That Cannot Be Overloaded

- Scope resolution (::)
- Member access (.)
- Member pointer selector (.*)
- Sizeof Ternary (?:)
- Type casting (dynamic_cast, static_cast)

6. Unary Operator Overloading

Overloading ++ and -- requires defining both prefix and postfix versions. Unary operators work on a single operand. These can be overloaded as member or friend functions.

Eg:

```
class Counter {  
private:  
    int value;
```

```

public:
    Counter(int v = 0) : value(v) {}
    Counter operator++() {
        ++value;
        return *this;
    }
};

```

7. Binary Operator Overloading

Binary operators work with two operands. For member functions, one operand is the current object.

Eg:

```

class Complex {

private:
    int real, imag;

public:
    Complex operator + (const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

};

```

8. Stream Insertion and Extraction Operators

You can overload the stream I/O operators (<<, >>) to handle user-defined types, making it easy to input and output objects.

Eg:

```

class Complex {
private:
    int real, imag;
public:
    friend std::ostream& operator << (std::ostream& out, const Complex& obj) {
        out << obj.real << " + " << obj.imag << "i";
        return out;
    }

    friend std::istream& operator >> (std::istream& in, Complex& obj) {
        in >> obj.real >> obj.imag;
    }
};

```

```
        return in;
    }
};
```

LAB QUESTIONS:

Ques 1: Write a program to add two complex numbers using + operator with operator overloading .

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

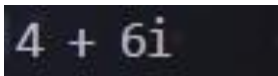
class Complex {
    float real, imag;
public:
    Complex() : real(0), imag(0) {}
    Complex(float r, float i) : real(r), imag(i) {}

    Complex operator+(const Complex& c) {
        return Complex(real + c.real, imag + c.imag);
    }

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

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

OUTPUT:



4 + 6i

Ques 2: Write a program to overload both postfix and prefix operator ++ using operator overloading .

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

class Count {
    int value;
public:
    Count(int v = 0) : value(v) {}

    // Prefix ++
    Count& operator++() {
        ++value;
        return *this;
    }

    // Postfix ++
    Count operator++(int) {
        Count temp = *this;
        value++;
        return temp;
    }

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

int main() {
    Count c(5);

    ++c;
    c.display();

    c++;
    c.display();

    return 0;
}
```

OUTPUT:

```
Value: 6
Value: 7
```

DISCUSSION:

Operator overloading in C++ allows developers to redefine the behavior of standard operators (like +, ++, etc.) for user-defined types such as classes. It makes object manipulation more intuitive and readable by enabling operators to work with class objects similarly to built-in types. For example, overloading the + operator for a Complex class allows direct addition of two complex numbers. Both unary (e.g., ++) and binary (e.g., +) operators can be overloaded to enhance the usability and functionality of objects.

CONCLUSION:

Operator overloading simplifies the interaction with class objects by allowing standard operators to be used meaningfully with them. It improves code readability and provides a natural syntax for object operations. Through this lab, we learned how to implement both unary and binary operator overloading, demonstrating how powerful and flexible C++ is for building user-friendly abstractions.