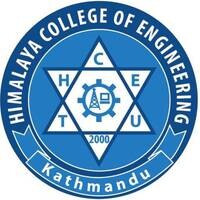


**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**



**HIMALAYA COLLEGE OF ENGINEERING**

**CHYASAL, LALITPUR**

**Lab Report No: - 5**

**Title: - Operator overloading**

**Submitted by: - Submitted To: -**

**Name: - Atul Bhattarai Department Of Computer and Electronics**

**Roll NO: - HCOE 081 BEI 011 Checked by: -**

**Date of submission: -**

## **Objective**

* To understand the concept of operator overloading in C++.
* To implement overloading of arithmetic, relational, and unary operators using member functions and friend functions.

## **Theory**

Operator overloading in C++ allows programmers to redefine the way operators work for user-defined data types (classes and objects). This enhances code readability and usability by enabling operations like +, -, ==, or ++ to work on objects just like they do on primitive data types.

C++ supports overloading of most operators (excluding a few like ::, ., .\*, and sizeof() by either using **member functions** or **friend functions**.

### **Types of Operators Commonly Overloaded:**

* **Arithmetic operators:** +, -, \*, /
* **Relational operators:** ==, !=, <, >
* **Unary operators:** ++, --
* **Assignment operators:** =

**Lab assignments:**

**Q1. Write a program to add two complex numbers using 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& obj) {

        return Complex(real + obj.real, imag + obj.imag);

    }

    void display() {

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

    }

};

int main() {

    Complex c1(3.4, 2.1), c2(1.6, 5.9), c3;

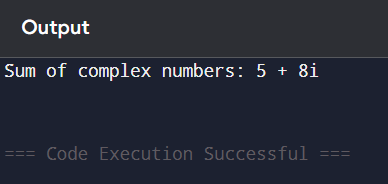
    c3 = c1 + c2;

    cout << "Sum of complex numbers: ";

    c3.display();

    return 0;

}



**Q2. Write a program to overload ++ increment operator.**

#include <iostream>

using namespace std;

class Counter {

    int value;

public:

    // Constructor

    Counter(int v = 0) {

        value = v;

    }

    // Overload prefix ++ operator

    Counter operator++() {

        ++value;

        return \*this;

    }

    // Overload postfix ++ operator

    Counter operator++(int) {

        Counter temp = \*this;

        value++;

        return temp;

    }

    void display() const {

        cout << "Value: " << value << endl;

    }

};

int main() {

    Counter c1(5);

    cout << "Initial: ";

    c1.display();

    ++c1;  // Calls prefix

    cout << "After prefix ++: ";

    c1.display();

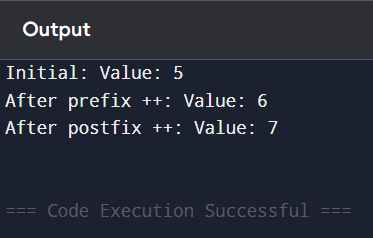
    c1++;  // Calls postfix

    cout << "After postfix ++: ";

    c1.display();

    return 0;

}

****

## **Discussion**

Operator overloading simplifies the interaction between objects by making the code intuitive and easy to read. In this lab, we observed how to redefine the + operator for the Complex class. The program was implemented using a member function that takes another object as an argument and returns a new object containing the result. This concept can be extended to overload other operators as well, like ==, -, or ++, allowing user-defined types to behave like built-in data types.

## **Conclusion**

The lab helped in understanding and implementing operator overloading in C++. Overloaded operators make code involving objects more natural and expressive. Mastery of operator overloading is essential for building reusable and modular code in object-oriented programming.