OBJECTIVE

- To comprehend the idea of C++ operator overloading.
- To apply operator overloading so that user-defined data types can be operated on.
- To distinguish between unary and binary operator overloading.

BACKGROUND THEORY

One of the most crucial aspects of C++'s object-oriented programming is operator overloading. It enables us to rethink how operators function for data types that are user-defined (such as classes and structures). Addition, subtraction, comparison, and other operations can be carried out on objects as though they were built-in data types by overloading operators.

Writing special functions called operator functions in C++ allows you to overload operators like +, -, *, ==, and many more. The operator symbol and the keyword operator are used to define these functions. Although it follows a particular format, the syntax is comparable to that of a regular function.

For example:

```
class Complex {
public:
   int real, imag;
   Complex operator + (Complex c);
};
```

In this example, the + operator is overloaded to add two Complex number objects.

There are two main types of operator overloading:

1. **Unary Operator Overloading**: Operates on a single operand. Examples include ++, --, -, etc.
2. **Binary Operator Overloading**: Operates on two operands. Examples include +, -, *, /, etc.

The following things should be considered while using operator overloading:

- It is not possible to overload every operator. Operators like :: (scope resolution),. (member access),.* (pointer-to-member), and sizeof are examples of non-overloadable operators.
- In order to prevent confusing code, operator overloading must preserve the operator's original meaning as much as feasible.

• Logical expectations should not be broken by the overloaded operator (for example, overloading == to behave like!= would make code difficult to understand).

1. **A member function inside a class:**

Syntax:

```
class ClassName {
public:
// Constructor and data members
ClassName(data_type var) : variable(var) {}
// Overload operator as member function
return_type operator<symbol>(const ClassName& other) {
// Define operator behavior
return result;
}
private:
data_type variable;
};
```

2. **Or a non-member (often friend) function:**

Syntax:

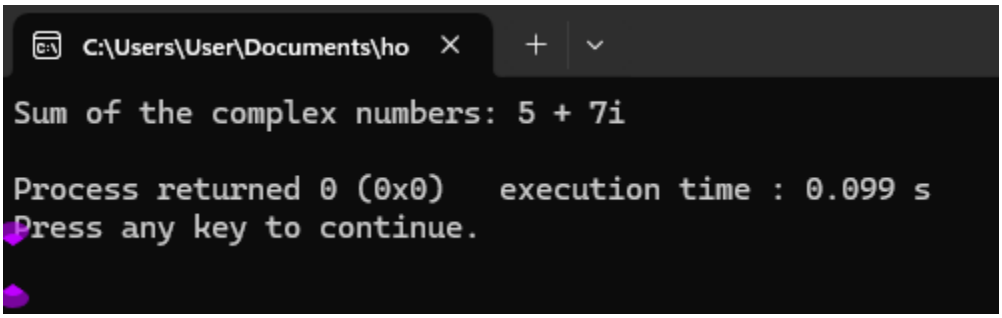
```
class ClassName {
public:
// Constructor and data members
ClassName(data_type var) : variable(var) {}
// Declare friend function for operator overloading
friend return_type operator<symbol>(const ClassName& obj1, const ClassName& obj2);
private:
data_type variable;
};
```

1. **Create a class Complex in C++ that represents Complex Number. Implement operator overloading for the plus operator to add two Complex Number objects and display the result.**

```
#include <iostream>
using namespace std;
class Complex {
private:
    float real;
    float imag;
public:
    // Constructor
    Complex(float r = 0, float i = 0) {
        real = r;
        imag = i;
    }
    Complex operator + (const Complex& obj) {
        Complex result;
        result.real = real + obj.real;
        result.imag = imag + obj.imag;
        return result;
    }
    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};
int main() {
    Complex c1(3.5, 2.5);
    Complex c2(1.5, 4.5)
    Complex c3 = c1 + c2;
```

```
    cout << "Sum of the complex numbers: ";
    c3.display();
    return 0;
}
```

Output



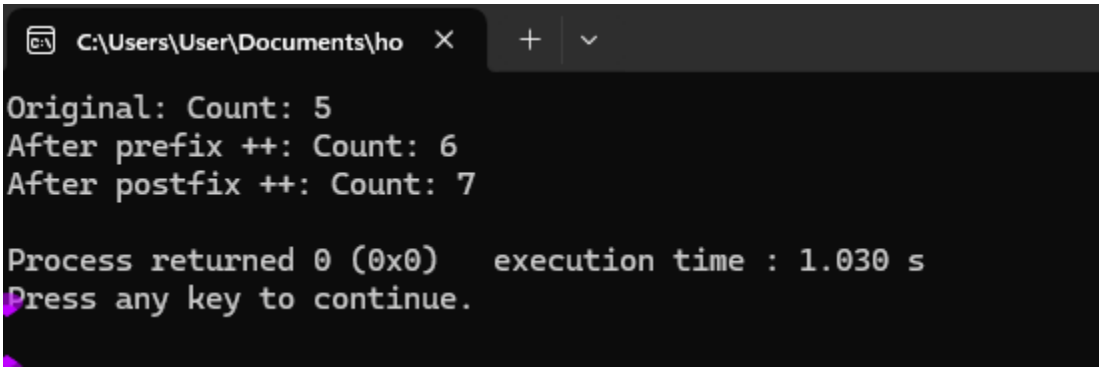
2. **Write a C++ program to overload both the prefix and postfix increment operators++ for a class.**

```
#include <iostream>
using namespace std;
class Counter {
private:
    int count;
public:
    Counter(int c = 0) : count(c) {}
  void display() {
        cout << "Count: " << count << endl;
    }
    Counter& operator++() {
```

```
        ++count;
        return *this;
    }
    Counter operator++(int) {
        Counter temp = *this;
        count++;
        return temp;
    }
};
int main() {
    Counter c1(5);
    cout << "Original: ";
    c1.display();

    ++c1;
    cout << "After prefix ++: ";
    c1.display();
    c1++;
    cout << "After postfix ++: ";
    c1.display();
    return 0;
}
```

Output:



Discussion

In this lab, we looked at C++'s operator overloading feature, which is a strong tool that lets us change how operators behave for user-defined types. We saw how custom implementations improve the intuitiveness of object interactions by overloading the prefix and postfix ++ operators for a counter class and the + operation for complex integers.

Conclusion

The lab effectively illustrated how C++ user-defined classes may be made more useful with operator overloading. We discovered how to efficiently overload both binary and unary operators through practical examples.