OBJECTIVE

- To understand the concept of operator overloading in C++.
- To implement operator overloading to perform operations on user-defined data types.
- To differentiate between overloading of unary and binary operators.

BACKGROUND THEORY

Operator Overloading is one of the most important features of Object-Oriented Programming in C++. It allows us to redefine the way operators work for user-defined data types (like classes and structures). By overloading operators, we can perform operations such as addition, subtraction, comparison, etc., on objects as if they were built-in data types.

In C++, operators such as +, -, *, ==, and many others can be overloaded by writing special functions known as *operator functions*. These functions are defined using the keyword operator followed by the operator symbol. The syntax is similar to a normal function but with a specific format. 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:

- Not all operators can be overloaded. Examples of non-overloadable operators include :: (scope resolution), . (member access), .* (pointer-to-member), and sizeof.
- Operator overloading must maintain the operator's original meaning as closely as possible to avoid confusing code.

- The overloaded operator should not violate logical expectations (e.g., overloading == to behave like != would make code hard 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:**

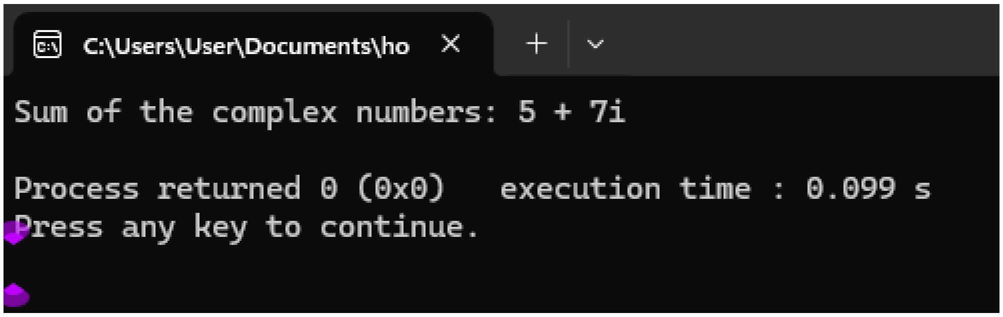
Syntax:

```
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

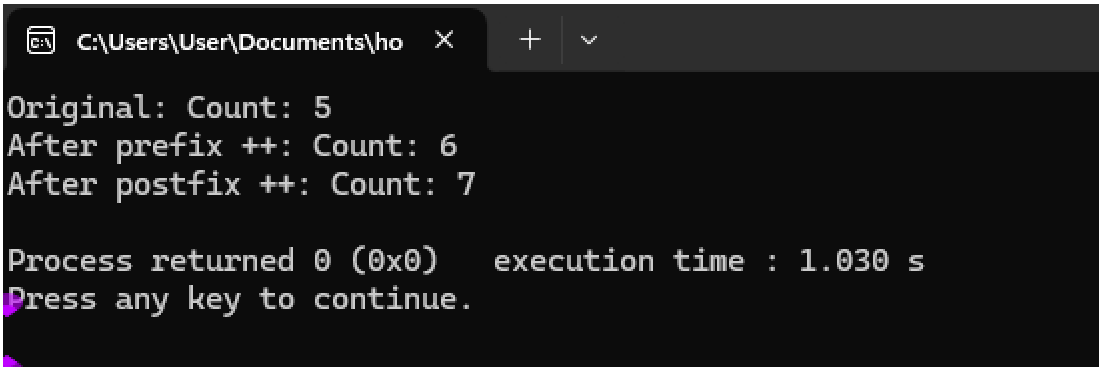


```
Sum of the complex numbers: 5 + 7i

Process returned 0 (0x0)   execution time : 0.099 s
Press any key to continue.
```

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:



```
Original: Count: 5
After prefix ++: Count: 6
After postfix ++: Count: 7

Process returned 0 (0x0)   execution time : 1.030 s
Press any key to continue.
```

Discussion

In this lab, we studied operator overloading in C++, a powerful feature that allows us to redefine the behavior of operators for user-defined types. By overloading the + operator for complex numbers and both prefix and postfix ++ operators for a counter class, we observed how custom implementations make objects interact more intuitively.

Conclusion

The lab successfully demonstrated how operator overloading can extend the functionality of user-defined classes in C++. Through hands-on examples, we learned how to overload both binary and unary operators effectively.