TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: -05

Title: - Operator Overloading

Submitted by: -	**Submitted To: -**
Name: - Diwas Pokhrel	**Department of Electronics and Computer Engineering**
Roll NO: - HCE081BEI014	**Checked by: -**

Date of submission: - 2082/03/13

OBJECTIVE:

- To understand the concept and importance of operator overloading in C++ programming.
- To learn how to use unary and binary operators with user-defined classes.
- To implement operator overloading for common operators such as +, -, *, and relational operators.
- To enable the use of built-in operators with class objects for intuitive operations.
- To explore the syntax and semantics of defining operator functions inside classes.
- To differentiate between member function and non-member function operator overloading.

TOOLS AND LIBRARIES USED:

• Programming Language C++
• IDE: Clang
• Libraries : include<iostream>,include<string>.

THEORY

In C++, operator overloading is a feature that allows developers to redefine the way operators work with user-defined types (like classes and structures). This means we can define custom behaviors for operators such as +, -, *, /, ==, >, <<, etc., when they are used with class objects.

Operator overloading helps make class objects behave more like built-in types, which improves code readability and intuitiveness. It is implemented using a special function called operator function.
Syntax:

```
return_type operator symbol (parameters) {
// body of function
}
```

Why Use Operator Overloading?

1. To perform operations between objects using familiar syntax (e.g., c1 + c2).
2. To increase program readability and maintainability.
3. To simplify complex object manipulations.
4. To mimic the behavior of standard data types with custom objects.

Rules for Operator Overloading

- At least one operand must be a user-defined type (i.e., an object of a class).
- Cannot create new operators – only existing C++ operators can be overloaded.
- Cannot change the precedence**,** associativity**,** or arity (number of operands) of operators.

Types of Operator Overloading:

1. Unary operators (e.g., ++, --) : can be overloaded using one operand.
2. Binary operators (e.g., +, -, *, ==) : use two operands.
3. Friend function operator overloading –: used when left-hand operand is not an object of the class.
4. **Note:** Certain operators like ::, .*, ., sizeof, typeid cannot be overloaded.

Example: Operator Overloading for Addition of Two Complex Numbers

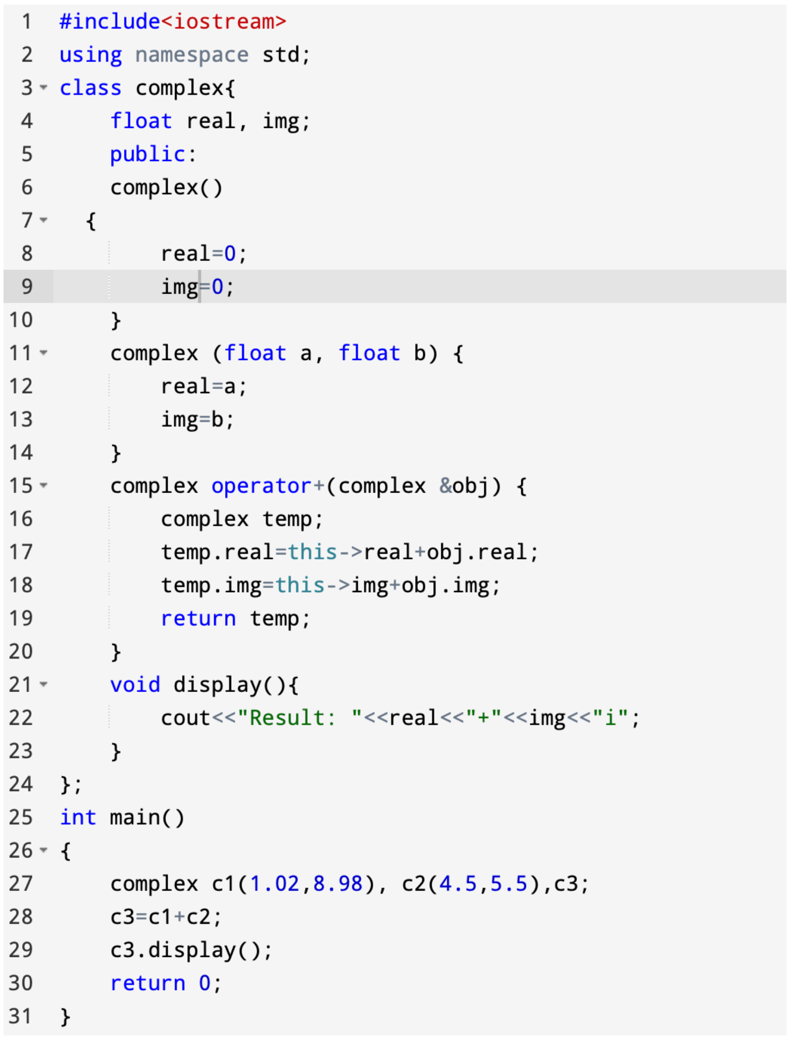
```
#include <iostream>

using namespace std;

class Complex {

   float real;
   float imag;
public:
   void getData() {
      cout << "Enter real and imaginary part: ";
      cin >> real >> imag;
   }

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

                                                  // Overload '+' operator using member function
   Complex operator + (Complex c) {
      Complex temp;
      temp.real = real + c.real;
      temp.imag = imag + c.imag;
      return temp;
   }
};

int main() {
   Complex c1, c2, c3;
   c1.getData();
   c2.getData();
   c3 = c1 + c2;
   cout << "Sum of complex numbers: ";
   c3.display();
   return 0;
}
```

Explanation:

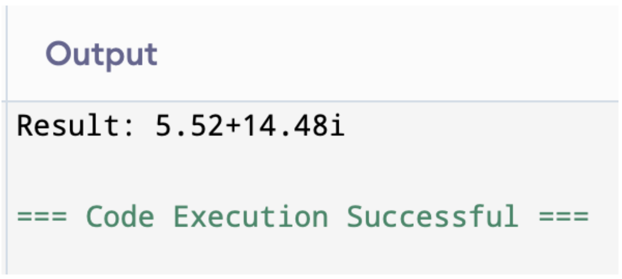
- The + operator is overloaded using a member function.
- Two Complex objects are added, and the result is returned as another Complex object.
- The overloading allows us to use c1 + c2 just like basic types.

RELATED PROGRAMS:

1) WRITE A PROGRAM TO ADD TWO COMPLEX NUMBERS USING OPERATOR OVERLOADING.



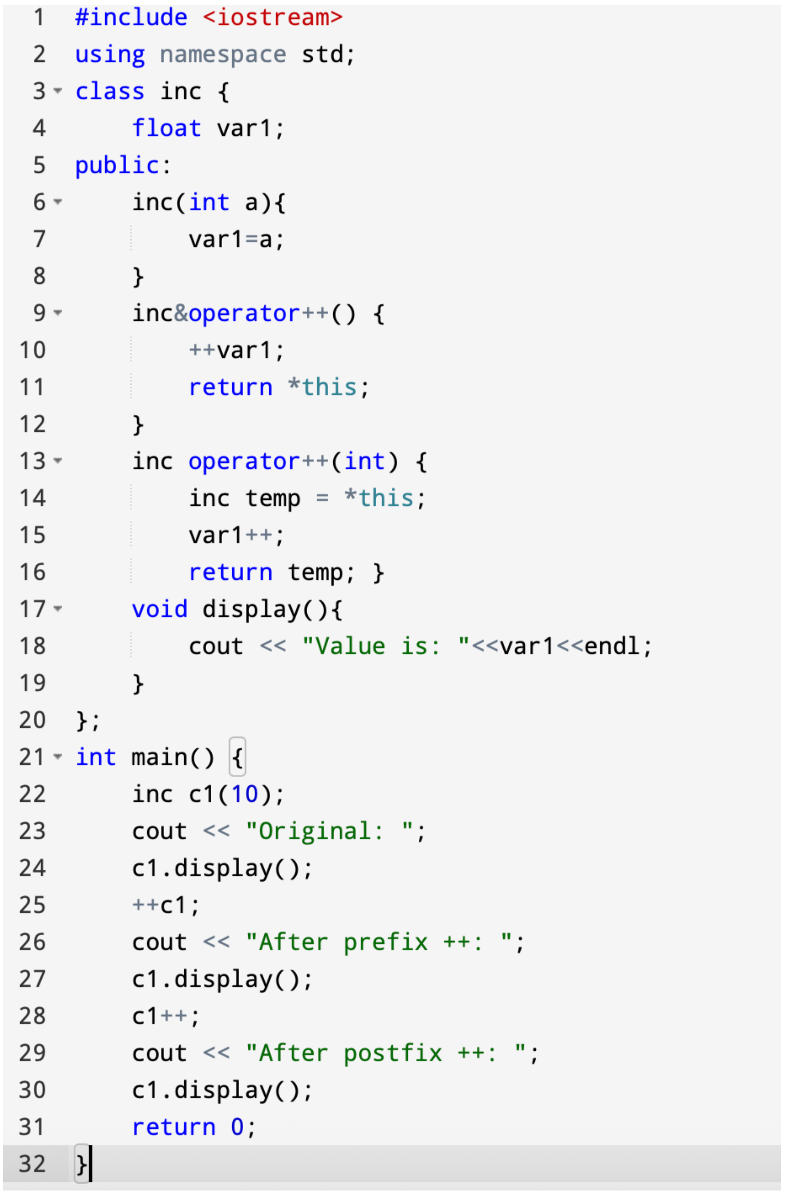
```cpp
#include<iostream>
using namespace std;
class complex{
    float real, img;
    public:
    complex()
  {
        real=0;
        img=0;
    }
    complex (float a, float b) {
        real=a;
        img=b;
    }
    complex operator+(complex &obj) {
        complex temp;
        temp.real=this->real+obj.real;
        temp.img=this->img+obj.img;
        return temp;
    }
    void display(){
        cout<<"Result: "<<real<<"+"<<img<<"i";
    }
};
int main()
{
    complex c1(1.02,8.98), c2(4.5,5.5),c3;
    c3=c1+c2;
    c3.display();
    return 0;
}
```



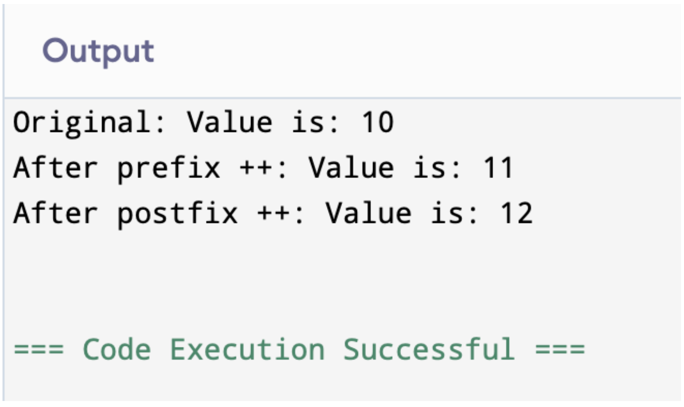
Output

```
Result: 5.52+14.48i


=== Code Execution Successful ===
```

2) WRITE A PROGRAM TO OVERLOAD ++ INCREMENT OPERATOR.



```cpp
#include <iostream>
using namespace std;
class inc {
    float var1;
public:
    inc(int a){
        var1=a;
    }
    inc&operator++() {
        ++var1;
        return *this;
    }
    inc operator++(int) {
        inc temp = *this;
        var1++;
        return temp; }
    void display(){
        cout << "Value is: "<<var1<<endl;
    }
};
int main() {
    inc c1(10);
    cout << "Original: ";
    c1.display();
    ++c1;
    cout << "After prefix ++: ";
    c1.display();
    c1++;
    cout << "After postfix ++: ";
    c1.display();
    return 0;
}
```



Output

```
Original: Value is: 10
After prefix ++: Value is: 11
After postfix ++: Value is: 12


=== Code Execution Successful ===
```

Discussion

During this lab, we implemented and explored the concept of operator overloading in C++. We created user-defined classes and successfully overloaded various operators like arithmetic (+, -), relational (==, >), and stream (<<, >>) operators.The lab emphasized how operator overloading makes object-oriented code more natural and intuitive. Instead of calling long member functions to perform actions on objects, we could use familiar operators—just like we do with primitive data types. This not only improves code readability but also helps build reusable and modular code structures.Common issues such as function signature mismatches and confusion between return-by-value and return-by-reference were encountered and resolved during the implementation phase.

Conclusion

The operator overloading lab in C++ provided a practical understanding of how to extend the functionality of built-in operators to work with user-defined classes. By overloading operators such as +, ==, and <<, we learned how to make class objects behave like primitive data types, thereby improving the readability and usability of code. The implementation of both member and friend functions illustrated the flexibility of C++ in supporting object-oriented principles like polymorphism. This lab reinforced the importance of operator overloading in creating clean, intuitive, and efficient object-based programs.