Objectives:

- Understand why and when to overload operators in C++.
- Practice overloading unary, binary, and stream (<<, >>) operators for custom types.

Tools and Libraries Used:

- Programming Language: C++
- IDE: G++
- Libraries: include <iostream>, include <string>

Theory:

Operator overloading lets built-in operators (like +, -, *, ==, ++, --, <<, >>) work with objects of your own classes. Instead of calling named functions (add(a,b)), you can write expressive code (a + b) that reads like built-in types. Under the hood, each overloaded operator is just a function—either a member (uses implicit left operand: this) or a non-member / friend (good for symmetry when the left operand isn't your class, e.g., stream insertion). Use overloading to hide representation details and give your class intuitive, type-safe behavior.

BASIC SYNTAX PATTERNS

Member form

```
class ClassName {
public:
explicit ClassName(data_type v) : variable(v) {}
  return_type operator<symbol>(const ClassName& other) const {
 return result;
  }
private:
   data_type variable;
};
```

Non-member / friend form

```
: Employee class ClassName {
public:
   explicit ClassName(data_type v) : variable(v)  {}
   friend return_type operator<symbol>(const ClassName& a, const ClassName& b);
private:
   data_type variable;
};
return_type operator<symbol>(const ClassName& a, const ClassName& b) {
   // access a.variable, b.variable  (friend grants access)
   return result;
}
```

KEY RULES

- You can only overload existing C++ operators (no new symbols).
- At least one operand must be a user-defined type.
- Precedence & associativity do not change.
- Arity (unary/binary) is fixed.
- These must be member overloads: =, (), [], ->.
- Use friend (or non-member) when the left operand isn't your type (e.g., operator<< for ostream).
- Make behavior intuitive and consistent (e.g., == implies logical equality; + shouldn't mutate operands).

Lab Questions:

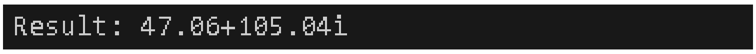
Q no 1:

Create a class complex in C++ that represents complex numbers. Implement operator overloading for the + operator to add two complex number objects and display the result.

Code:

```
 #include<iostream>
using namespace std;
class complex{
    float real, imz;
    public:
    complex()
    {
        real=0;
        imz=0;
    }
    complex (float a, float b)
    {
        real=a;
        imz=b;
    }
    complex operator+(complex &obj)
    {
        complex temp;
        temp.real=this->real+obj.real;
        temp.imz=this->imz+obj.imz;
        return temp;
    }
    void display()
    {
        cout<<"Result: "<<real<<"+"<<imz<<"i";
    }
};
int main()
{
    complex c1(1.56,8.94), c2(45.5,96.1),c3;
    c3=c1+c2;
    c3.display();
    return 0;
}
```

Output:



```
Result: 47.06+105.04i
```

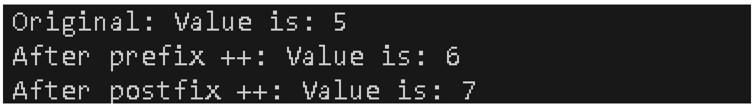
Q no 2:

Write a C++ program to overload both the prefix and postfix increment operators (++) for a class

Code:

```
#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(5);
       cout << "Original: ";
       c1.display();
       ++c1;
       cout << "After prefix ++: ";
       c1.display();
       c1++;
       cout << "After postfix ++: ";
       c1.display();
       return 0;
   }
```

Output:



```
Original: Value is: 5
After prefix ++: Value is: 6
After postfix ++: Value is: 7
```

Conclusion:

This lab explored operator overloading in C++, demonstrating how custom classes can use built-in operators like + and ++ with user-defined behavior. The programs showed how operator overloading improves code readability and makes objects act like primitive data types, reinforcing object-oriented design principles.