TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING
CHYASAL, LALITPUR

Lab Report No: - Operator Overloadin
Title: -05

Submitted by: -
Name: -Aditya Man Shrestha
Roll NO: -HCE081BEI005

Submitted To: -
Department Of
Checked by: -

Date of submission: -

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: Code::Blocks
- 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;
};
```

2.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.

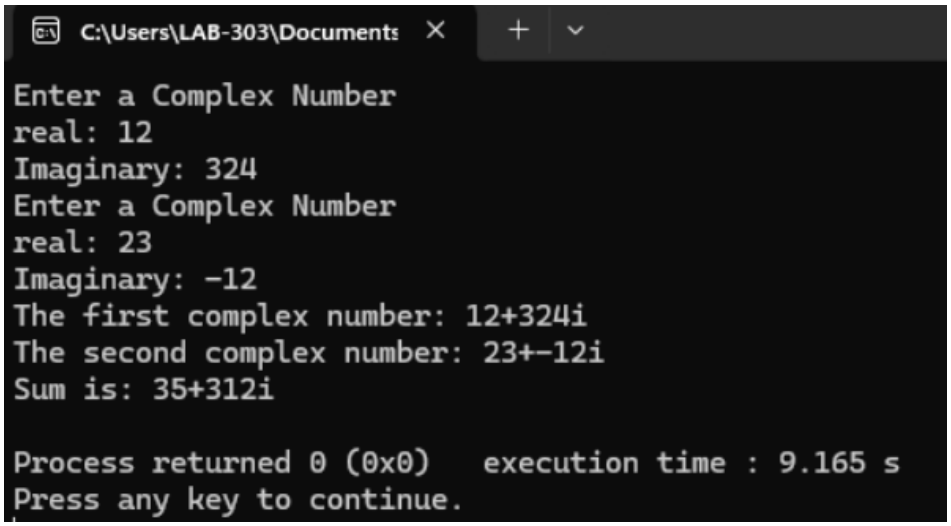
• These must be member overloads: =, (), [], ->.

• 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).

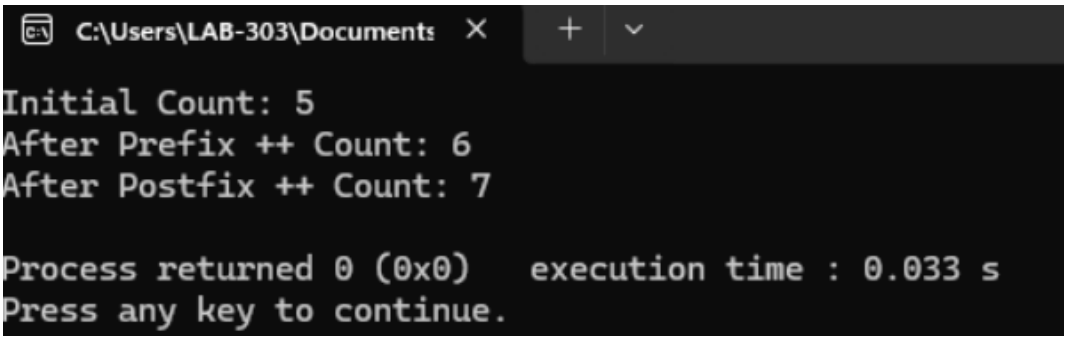
Lab Assignment
Qn1.

```
#include<iostream>
using namespace std;
class complex{
int real, img;
public:
complex(int r=0,int i=0): real(r),img(i){}
friend complex operator +(complex c1, complex c2);
void input(){
cout<<"Enter a Complex Number"<<endl;
cout<<"real: ";
cin>>real;
cout<<"Imaginary: ";
cin>>img;
}
void display(){
cout<<real<<"+"<<img<<"i"<<endl;
}
};
complex operator +(complex c1, complex c2){
complex temp;
temp.real=c1.real+c2.real;
temp.img=c1.img+c2.img;
return temp;
}
int main(){
complex c1, c2, c3;
c1.input();
c2.input();
cout<<"The first complex number: ";
c1.display();
cout<<"The second complex number: ";
c2.display();
c3= c1+c2;
cout<<"Sum is: ";
c3.display();
}
```



Qn2.

```
#include<iostream>
using namespace std;
class Counter {
   int count;
public:
   Counter(int c = 0) : count(c) {}
   Counter operator++() {
      ++count;
      return *this;
   }
   Counter operator++(int) {
      Counter temp = *this;
      count++;
      return temp;
   }
   void display() {
      cout << "Count: " << count << endl;
   }
};
int main() {
   Counter c1(5);
   cout << "Initial ";
   c1.display();
   ++c1;
   cout << "After Prefix ++ ";
   c1.display();
   c1++;
   cout << "After Postfix ++ ";
   c1.display();
   return 0;
}
```



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.