TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

CHYASAL, LALITPUR

Lab Report No: -05

Title: -Operator Overloading

Submitted by: -

Name: -Gaurab Pandey

Roll NO: -15

Submitted To: -

Department Of

Checked by: Date of submission: -

OBJECTIVE:

- To understand and implement the concept of operator overloading and type conversion in C++ using user defined classes.

THEORY:

Operator overloading

Operator overloading in C++ is a feature that allows the redefinition of the behavior of standard C++ operators (like +, -, *, /, ==, <<, >>, etc.) when applied to user-defined types (classes and structs).

Overloadable and Non-overloadable Operators:

In C++, most operators can be overloaded to provide customized behavior when working with class objects. Common examples include +, -, *, /, ==, [], (), and many others. However, there are certain operators that cannot be overloaded. These include the scope resolution operator **::,** the member access operator **.,** the pointer-to-member operator .*, the ternary conditional operator ?:, as well as sizeof and typeid**.**

Syntax of Operator Overloading:

Operator overloading uses the keyword operator followed by the operator symbol.

It is defined either inside the class or as a friend function.

Example Syntax:

```
return_type class_name::operator op (parameters) {
// code
}
```

Operator Overloading Using Member Operator Functions:

A member function can be used to overload an operator. It has access to the current object (this) And is defined inside the class.

Unary Operator Overloading

Unary operator overloading in C++ is polymorphism in which we overload an operator to perform a similar operation with the class objects.

Example:

```
class A {
public:
   int value;
   A(int v) : value(v)
   {} A operator-() {
      return A(-value);
   }
};
```

Binary Operator Overloading

Binary operator overloading in C++ allows programmers to redefine the behavior of operators like
+, -, *, / when used with user-defined data types (classes and structures). This means we can make these operators work in a meaningful way with objects, rather than just built-in data types.

Example:

```
class B {
   int data;
public:
   B(int d) : data(d) {}
   friend B operator/(const B& obj1, const B& obj2);
};
B operator/(const B& obj1, const B&
   obj2) { return B(obj1.data /)
```

Qno.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.

```cpp
#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imaginary;

public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imaginary(i) {}
    double getReal() const { return real; }
    double getImaginary() const { return imaginary; }

    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imaginary + other.imaginary);
    }
    void display() const {
        cout << real;
        if (imaginary >= 0) {
            cout << " + " << imaginary << "i";
        } else {
            cout << " - " << -imaginary << "i";
        }
    }
};
```

```cpp
int main() {
    Complex c1(3.0, 4.0);
    Complex c2(1.5, -2.5);
    Complex result = c1 + c2;
    cout << "First complex number: ";
    c1.display();
    cout << endl;

    cout << "Second complex number: ";
    c2.display();
    cout << endl;

    cout << "Result of addition: ";
    result.display();
    cout << endl;

    return 0;
}
```

```
First complex number: 3 + 4i
Second complex number: 1.5 - 2.5i
Result of addition: 4.5 + 1.5i
```

Qno.2

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

```cpp
#include <iostream>
using namespace std;

class Counter {
private:
    int count;

public:
    Counter(int initial = 0) : count(initial) {}
    int getCount() const { return count; }
    Counter& operator++() {
        ++count;
        return *this;
    }
    Counter operator++(int) {
        Counter temp = *this;
        ++count;
        return temp;
    }
    void display() const {
        cout << "Count: " << count << endl;
    }
};
```

```cpp
int main() {
    Counter c1(5);
    Counter c2(5);

    cout << "Initial values:" << endl;
    c1.display();
    c2.display();
    Counter c3 = ++c1;
    cout << "\nAfter prefix increment (++c1):" << endl;
    cout << "c1: ";
    c1.display();
    cout << "c3 (result of ++c1): ";
    c3.display();
    Counter c4 = c2++;
    cout << "\nAfter postfix increment (c2++):" << endl;
    cout << "c2: ";
    c2.display();
    cout << "c4 (result of c2++): ";
    c4.display();

    return 0;
}
```

```
Initial values:
Count: 5
Count: 5

After prefix increment (++c1):
c1: Count: 6
c3 (result of ++c1): Count: 6

After postfix increment (c2++):
c2: Count: 6
c4 (result of c2++): Count: 5
```

Discussion

The first C++ program demonstrates operator overloading for complex number addition, allowing intuitive syntax (c1 + c2) while handling real and imaginary components. The second program overloads both prefix (++obj) and postfix (obj++) increment operators for a Counter class, distinguishing them through function signatures and maintaining expected increment behaviour. Both examples showcase how operator overloading enhances code readability and makes user-defined types behave like built-in types, with the first focusing on mathematical operations and the second on increment semantics, illustrating C++'s flexibility in customizing operators for custom classes.

Conclusion

Operator overloading in C++ allows user-defined types to mimic built-in operations, improving code clarity and usability. The complex number example demonstrates intuitive arithmetic operations, while the increment overloading showcases how to replicate familiar behaviours like ++obj and obj++. Together, these programs highlight C++'s power in creating expressive and natural syntax for custom classes, making them as easy to use as primitive data types. Mastering operator overloading leads to cleaner, more maintainable, and intuitive object-oriented code.