Himalaya College of Engineering

Advanced C++ Programming Lab Report

Lab 5: Operator overloading in C++

Prepared By : Nawnit Paudel (HCE081BEI024)

Subject : Object-Oriented Programming (OOP)

Program : Bachelor of Electronics, Communication and Information Engineering

Institution : Himalaya College of Engineering

OBJECTIVE

- To understand the concept and purpose of operator overloading in C++.
- To implement various types of operator overloading for custom data types.
- To learn how to overload binary, unary, and I/O stream operators.
- To enhance code readability and provide intuitive syntax for user-defined classes.

BACKGROUND THEORY

Operator overloading is a powerful feature in C++ that allows operators to be redefined or overloaded for user-defined data types. This means you can use operators like +, -, *, /, ==, ++, --, etc., with objects of your own classes, giving them special meaning for those classes. The primary purpose of operator overloading is to provide a more natural and intuitive syntax for operations involving user-defined types, making the code more readable and easier to understand.

When an operator is overloaded, it is implemented as a function. This function can be either a member function of the class or a non-member (friend) function. The choice between a member function and a friend function often depends on the type of operator and the number of operands.

General Syntax for Operator Overloading:

1. A member function inside a

class: 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: 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;

};

// Define non-member operator function return_type operator<symbol>(const

ClassName& obj1, const ClassName& obj2) {

// Define operator behavior return

result;

}

Importance of Operator Overloading

1. **Makes Code Easy to Read:**
   Operator overloading lets us use symbols like +, -, or == with objects, making the code look simple and clear.
2. **Hides Internal Details:**
   It helps create user-friendly data types where complex details are hidden, and only useful operations are shown.
3. **Saves Time and Effort:**
   Once an operator is overloaded in a class, it can be reused easily. This reduces repeated code and makes updates easier.
4. **Supports OOP Concepts:**
   Operator overloading follows object-oriented ideas, like **polymorphism**, by letting the same operator work in different ways for different objects.

Rules of Operator Overloading in C++

1. Only existing operators can be overloaded; new operator symbols cannot be created.
2. At least one operand must be a user-defined type like a class or structure.
3. Operator precedence and associativity remain unchanged after overloading.
4. The number of operands for an operator cannot be changed.
5. Operators such as =, (), [], and -> must be overloaded as member functions.
6. Friend functions can be used to overload operators needing access to private members.
7. Overloaded operators should behave in a meaningful and expected way.

LAB ASSIGNMENT:

1.)Design a class Matrix of dimension 3x3. Overload + operator to find sum of two matrices.

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

class Matrix {
private:
   int mat[3][3];

public:
   Matrix() {
      for (int i = 0; i < 3; ++i)
         for (int j = 0; j < 3; ++j)
            mat[i][j] = 0;
   }

   void input() {
      cout << "Enter elements of 3x3 matrix:\n";
      for (int i = 0; i < 3; ++i) {
         for (int j = 0; j < 3; ++j) {
            cin >> mat[i][j];
         }
      }
   }

   void display() const {
      for (int i = 0; i < 3; ++i) {
         for (int j = 0; j < 3; ++j) {
            cout << mat[i][j] << " ";
         }
         cout << endl;
      }
```

```
    }

    Matrix operator+(const Matrix& m) const {
        Matrix temp;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                temp.mat[i][j] = mat[i][j] + m.mat[i][j];
            }
        }
        return temp;
    }
};

int main() {
    Matrix m1, m2, sum;

    cout << "Matrix 1:" << endl;
    m1.input();

    cout << "Matrix 2:" << endl;
    m2.input();

    sum = m1 + m2;

    cout << "Sum of the two matrices:" << endl;
    sum.display();

    return 0;
}
```

2. Define a class string and use + and > operators to concatenate and compare two strings respectively.

```
#include <iostream>

#include <string>

using namespace std;


class String {

    string str;


public:
```

```
    void input() {
        cout << "Enter String: ";
        cin >> str;
    }

    void display() {
        cout << str;
    }

    String operator+(String& s) {
        String temp;
        temp.str = str + s.str;
        return temp;
    }

    bool operator>(String& s) {
        return str > s.str;
    }
};

int main() {
    String s1, s2, s3;

    cout << "Input first String:" << endl;
    s1.input();

    cout << "Input second String:" << endl;
    s2.input();
```

```
    s3 = s1 + s2;

    cout << "Concatenated String: " << endl;
    s3.display();
    cout << endl;

    if (s1 > s2)
        cout << "First String is greater." << endl;
    else
        cout << "Second String is greater." << endl;

    return 0;
}
```

3. Write a program to implement vector addition and subtraction using operator overloading.

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

class Vector {
    int x, y, z;

public:
    Vector() {
        x = 0;
        y = 0;
        z = 0;
    }

    Vector(int a, int b, int c) {
        x = a;
        y = b;
        z = c;
    }

    void input() {
        cin >> x >> y >> z;
    }
```

```
    void display() {
        cout << "(" << x << ", " << y << ", " << z << ")";
    }

    Vector operator+(const Vector& v) const {
        return Vector(x + v.x, y + v.y, z + v.z);
    }

    Vector operator-(const Vector& v) const {
        return Vector(x - v.x, y - v.y, z - v.z);
    }
};

int main() {
    Vector v1, v2, sum, diff;

    cout << "Enter components of first vector: ";
    v1.input();

    cout << "Enter components of second vector: ";
    v2.input();

    sum = v1 + v2;
    diff = v1 - v2;

    cout << "Sum of vectors: ";
    sum.display();
    cout << endl;

    cout << "Difference of vectors: ";
    diff.display();
    cout << endl;

    return 0;
}
```

4. Design a class Matrix, overload ++ and -- operator to increment and decrement each element of the matrix by 1.

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

class Matrix {
    int mat[3][3];

public:
    Matrix() {
```

```
        for (int i = 0; i < 3; ++i)
            for (int j = 0; j < 3; ++j)
                mat[i][j] = 0;
    }

    void input() {
        for (int i = 0; i < 3; ++i)
            for (int j = 0; j < 3; ++j)
                cin >> mat[i][j];
    }

    void display() {
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j)
                cout << mat[i][j] << " ";
            cout << endl;
        }
    }

    Matrix& operator++() { // Pre-increment
        for (int i = 0; i < 3; ++i)
            for (int j = 0; j < 3; ++j)
                ++mat[i][j];
        return *this;
    }

    Matrix operator++(int) { // Post-increment
        Matrix temp = *this;
        ++(*this);
        return temp;
    }

    Matrix& operator--() { // Pre-decrement
        for (int i = 0; i < 3; ++i)
            for (int j = 0; j < 3; ++j)
                --mat[i][j];
        return *this;
    }

    Matrix operator--(int) { // Post-decrement
        Matrix temp = *this;
        --(*this);
        return temp;
    }
};
```

```
int main() {
    Matrix m, result;

    cout << "Enter elements of 3x3 matrix:" << endl;
    m.input();

    cout << "Original matrix:" << endl;
    m.display();

    result = ++m;
    cout << "Result of pre-increment:" << endl;
    result.display();

    result = m++;
    cout << "Result of post-increment:" << endl;
    result.display();

    result = --m;
    cout << "Result of pre-decrement:" << endl;
    result.display();

    result = m--;
    cout << "Result of post-decrement:" << endl;
    result.display();

    return 0;
}
```

DISCUSSION:

Operator overloading lets us give special meaning to symbols like +, -, and * when we use them with our own classes. This way, we can add or compare objects just like we do with numbers. It makes the code easier to read and use because we don't have to call long function names every time.

CONCLUSION:

To sum up, operator overloading helps us use normal operators with our own types, making programming simpler and clearer. It allows objects to interact in natural ways, just like built-in types such as integers or strings. But we should use it carefully so the code stays easy to understand and doesn't become confusing.