Objectives:

- Understand the concept of operator overloading in C++.
- Demonstrate syntax and rules of overloading.
- Use both member and friend functions for operator overloading.

1. Introduction to Operator Overloading

Operator overloading allows built-in C++ operators to work with user-defined types (classes/ objects). It Enhances code readability, making objects behave like basic types in expressions. Only existing operators can be overloaded. New operators cannot be created.

2. Syntax of Operator Overloading

Operator overloading is done using a special function with the operator keyword.

Syntax:

```
returnType operator op(parameterList);
```

Example:

```
class Complex {
float real, imag;
public:
Complex(float r = 0, float i = 0): real(r), imag(i) {}
Complex operator+(const Complex& c) {
return Complex(real + c.real, imag + c.imag);
}
};
```

3. Overloading Operators Using Member Functions

Member functions are used when the left operand is the object invoking the operator. It is best for operators that modify or compare object data.

Syntax:

```
class ClassName
{ public:
returnType operator op(const ClassName &obj); };
```

Example:

```
class Point {
int x, y;
public:
Point(int a = 0, int b = 0): x(a), y(b) {}
Point operator+(const Point& p)
{  return Point(x + p.x, y + p.y);
}
};
```

4. Overloading Operators Using Friend Functions

It is used when the left operand is not an object of the class. Friend functions are defined outside the class but declared inside with friend.

Syntax:

```
friend returnType operator op(const ClassName &lhs, const ClassName &rhs);
```

Example:

```
class Distance
{ int meters;
public:
Distance(int m = 0): meters(m) {}
friend Distance operator+(const Distance& d1, const Distance& d2);
};
Distance operator+(const Distance& d1, const Distance& d2)
{ return Distance(d1.meters + d2.meters);
}
```

5. Operators That Can Be Overloaded

Arithmetic: +, -, *, /, %
Relational: ==, !=, <, >, <=, >=
Assignment: =, +=, -=, etc.
Unary: ++, --
,
-, !
Stream: <<, >> (friend functions)

6. Operators That Cannot Be Overloaded

Scope resolution: ::
Member access: .
Member pointer: .*
sizeof, typeid, ?: (ternary)

7. Unary Operator Overloading

Applies to one operand like ++, --
.
Prefix Syntax: ClassName& operator++();
Postfix Syntax: ClassName operator++(int);Example:

```
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;
}
};
```

8. Binary Operator Overloading

Operates on two operands. First is implicit (this), second is explicit (parameter).
Syntax:

```
ClassName operator+(const ClassName&);
```

Example:

```
class Complex {
float real, imag;
public:
Complex(float r = 0, float i = 0): real(r), imag(i) {}
Complex operator+(const Complex& c) {
return Complex(real + c.real, imag + c.imag); }
};
```

9. Stream Insertion and Extraction Operators

It is used for input/output operations on objects. It must be defined as friend functions.
Syntax:

```
friend ostream& operator<<(ostream &out, const ClassName &obj);
friend istream& operator>>(istream &in, ClassName &obj);
```

Example:

```
class Student {
string name;
```

```
int age;
public:
friend ostream& operator<<(ostream& out, const Student& s);
friend istream& operator>>(istream& in, Student& s); };
ostream& operator<<(ostream& out, const Student& s) {
out << "Name: " << s.name << ", Age: " << s.age;
return out; }
istream& operator>>(istream& in, Student& s)
{ in >> s.name >> s.age;
return in;
}
```

Q1) Design a class Matrix of dimension 3x3. Overload + operator to find sum of two matrices.

Code:

```
#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 setMatrix() {
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
cin >> mat[i][j];
}
}
}
void displayMatrix() {
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
cout << mat[i][j] << " ";
}
cout << endl;
}
}
Matrix operator+(Matrix& other)
{ Matrix result;
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
result.mat[i][j] = mat[i][j] + other.mat[i][j];
}
}
return result;
}
};
int main() {
Matrix mat1, mat2;
cout << "Enter the first matrix: " << endl;
mat1.setMatrix();
cout << "Enter the second matrix: " << endl;
mat2.setMatrix();
cout << "Sum of the matrices: " << endl;
Matrix sum = mat1 + mat2;
sum.displayMatrix();
return 0;
}
```

Output:

```
Enter the first matrix:
2
3
4
2
4
5
3
4
5
Enter the second matrix:
3
7
4
2
3
4
8
0
0
Sum of the matrices:
5 10 8
4 7 9
11 4 5
```

Q2) Define a class string and use + and > operators to concatenate and compare two strings
respectively.

Code:

```
#include <iostream>
#include <cstring>
using namespace std;
class String
{   private:
char str[100];
public:
String()
{ str[0] =
'\0';
}
void setString()
{ cin.getline(str, 100);
}
void displayString()
{ cout << str << endl;
}
String operator+(String& other)
{ String result;
strcpy(result.str, str);
strcat(result.str, other.str);
return result;
}
bool operator>(String& other)
{ return strlen(str) >
strlen(other.str);
```

```
}
```

```
};
int main()
{ String str1,
str2;
cout << "Enter the first string: ";
str1.setString();
cout << "Enter the second string: ";
str2.setString();
cout << "Concatenated string: ";
String concatenated = str1 + str2;
concatenated.displayString();
cout << "First string is greater than second string(0 for false, 1 for true): " << (str1 > str2) << endl;
return 0;
}
```

Output:

```
Enter the first string: Aditya
Enter the second string:Sharma
Concatenated string:AdityaSharma
First string is greater than second string(0 for false, 1 for true): 1
```

Q3) Write a program to implement vector addition and subtraction using operator overloading.

Code:

```
#include <iostream>
using namespace std;
class Vector
{  private:
public:
double x, y, z;
Vector(double x=0, double y=0, double z=0) : x(x), y(y), z(z) {}
void setVector() {
cout << "Enter the x, y, z components of the vector: ";
cin >> x >> y >> z;
}
Vector operator+(Vector& other) {
return Vector(x + other.x, y + other.y, z + other.z);
}
Vector operator-(Vector& other) {
return Vector(x - other.x, y - other.y, z - other.z);
}
void displayVector() {
cout << "Vector: (" << x << ", " << y << ", " << z << ")" << endl;
}
};
int main()
{ Vector v1, v2;
cout << "Enter the first vector: ";
v1.setVector();
cout << "Enter the second vector: ";
v2.setVector();
cout << "Sum of the vectors: ";
Vector sum = v1 + v2;
sum.displayVector();
cout << "Difference of the vectors: ";
Vector diff = v1 - v2;
diff.displayVector();
return 0;
}
```

Output:

```
Enter the first vector: Enter the x, y, z components of the vector: 4
```

1
Enter the second vector: Enter the x, y, z components of the vector: 5
7
9
Sum of the vectors: Vector: (9, 13, 10)
Difference of the vectors: Vector: (-1,
-1,
-8)

Q4) Design a class Matrix, overload ++ and -- operator to increment and decrement each element of the matrix by 1

Code:

```
#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 setMatrix() {
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
cin >> mat[i][j];
}
}
}
void displayMatrix() {
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
cout << mat[i][j] << " ";
}
cout << endl;
}
}
// Prefix increment
Matrix operator++() {
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
mat[i][j]++;
}
}
return *this;
}
// Postfix increment
Matrix operator++(int)
{ Matrix temp = *this;
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
mat[i][j]++;
}
}
return temp;
}
// Prefix decrement
Matrix operator--() {
for (int i = 0; i < 3; i++) {for (int j = 0; j < 3; j++) {
mat[i][j]--;
}
```

```
}
return *this;
}
// Postfix decrement
Matrix operator--(int)
{ Matrix temp = *this;
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
mat[i][j]--;
}
}
return temp;
}
};
int main()
{ Matrix
mat;
cout << "Enter the matrix:\n";
mat.setMatrix();
cout << "Matrix:\n";
mat.displayMatrix();
cout << "\nPrefix incremented matrix:\n";
++mat;
mat.displayMatrix();
cout << "\nPostfix incremented matrix:\n";
mat++;
mat.displayMatrix();
cout << "\nPrefix decremented matrix:\n";
--mat;
mat.displayMatrix();
cout << "\nPostfix decremented matrix:\n";
mat--;
mat.displayMatrix();
return 0;
}
```

Output:

```
Enter the matrix:
2
5
3
5
4
6
3
6
0
Matrix:
2 5 3
5 4 6
3 6 0
Prefix incremented matrix:
3 6 4
6 5 7
4 7 1
Postfix incremented matrix:
4 7 5
7 6 8
5 8 2
Prefix decremented matrix:
3 6 4
6 5 7
4 7 1
Postfix decremented matrix:
2 5 3
5 4 6
```

360

Q5) Write a program to access elements of a vector class with index operator.

Code:

```
#include <iostream>
using namespace std;
class Vector
{  private:
double vec[5];
int size;
public:
Vector() : size(5) {
for (int i = 0; i < size; i++)
{ vec[i] = 0;
}
}
void setVector() {
for (int i = 0; i < size; i++)
{ cin >> vec[i];
}
}
void displayVector() {
for (int i = 0; i < size; i++)
{ cout << vec[i] << " ";
}
cout << endl;
}
double operator[](int index) {
if (index >= 0 && index < size)
{ return vec[index];
}
else {
cout << "Index out of bounds" << endl;
return 0;
}
}
};
int main()
{ Vector v;
cout << "Enter the elements of the vector: ";
v.setVector();
cout << "The vector is: ";
v.displayVector();
cout << "The element at index 2 is: " << v[1] << endl;
return 0;
}
```

Output:

```
Enter the elements of the vector: 3
5
8
5
0
The vector is: 3 5 8 5 0
The element at index 2 is: 5
```

Discussion:

Operator overloading lets us use standard operators (like +, -, ==) with user-defined objects. This makes code easier to read and understand. In the lab, we used member and friend functions to overload operators. For example, overloading + in a Complex class allowed us to add two complex numbers directly using c1 + c2.

Conclusion:

Through this lab, we learned how operator overloading makes object-oriented programs more readable and easier to manage. It helps apply operators directly to class objects, just like built-in types, enhancing usability and simplifying code.