**Lab 5: Operator Overloading**

**Objective:**

* To understand the concept of operator overloading in C++.
* To learn how to overload operators to work with user-defined types.
* To demonstrate the syntax and rules of operator overloading.
* To apply member and friend functions in operator overloading.

**Theory:**

**Introduction to operator overloading**

Operator overloading is the process of assigining a new and class-specific meaning to an existing operator, allowing it to perform customized operations on objects of a user-defined types. This mechanism is a form of compile-time polymorphism, where the operator’s behavior is determined at the time of compilation based on the types of operands involved. It helps to improve the code readability and also it helps in supporting the polymorphism and encapsulation.

**Rules of operator overloading**

Operator overloading is the process of giving an existing operator a new, class-specific meaning, enabling it to perform customized operations on objects of user-defined types. This feature in C++ allows operators like +, -, \*, and others to be redefined so they can work intuitively with objects, just as they do with built-in data types. By overloading operators, developers can write more readable and expressive code that naturally reflects the behavior of the objects being manipulated.

**Operator overloading using member operator function**

When using member function for overloading, the operators logic is implementrs directly inside the class using a special function known as the operator function.

**Syntax**

class ClassName {

public:

ReturnType operator<symbol>(ParameterList) {

}};

**Operator overloading using friend function**

Operator overloading using a non-member function involves defining the operator function outside the class. This approach is useful when the left-hand side (LHS) operand is not an object of the class.

**Syntax**

class ClassName {

public:

friend ReturnType operator<symbol>(ClassName obj1, ClassName obj2);

};

ReturnType operator<symbol>(ClassName obj1, ClassName obj2) {

}

**Operators that can be overloaded**

. Operators that can be overloaded include arithmetic operators such as +, -, \*, /, and %, which allow mathematical operations between objects. Relational operators like ==, !=, >, <, >=, and <= can be overloaded to compare object values. Assignment operators such as =, +=, -=, and others enable object assignment and modification. Unary operators including ++, --, - (unary minus), and ! can be used for operations like increment, decrement, and logical negation. Additionally, stream operators << and >> are often overloaded to enable customized input and output handling for objects, and must be defined as friend functions.

**Operators that cannot can be overloaded**

While C++ supports overloading many operators, there are certain operators that cannot be overloaded due to language design restrictions and to preserve the core behavior of the language. These include the scope resolution operator (::), which is used to define the scope of classes and functions, and the member access operators (. and .\*), which access class members directly. The sizeof operator, used to determine the size of a data type or object, and the ternary conditional operator (?:) also cannot be overloaded.

**Unary operator overloading**

Unary operator overloading allows customization of the behavior of unary operators like ++ and -- for user-defined types. The prefix version is defined using the syntax

ClassName& operator++(); // Prefix increment

The postfix version is distinguished by an int dummy parameter and uses the syntax

ClassName operator++(int); // Postfix increment (dummy 'int' distinguishes it).

An example of postfix and prefix

class Counter {

int count;

public:

// Prefix ++ (++obj)

Counter& operator++() {

++count;

return \*this;

}

// Postfix ++ (obj++)

Counter operator++(int) {

Counter temp = \*this; // Save original value

++count; // Modify the object

return temp; // Return the old value

}

// Similarly for -- operators

};

**Binary operator overloading**

When overloading a binary operator as a member function, it takes one explicit parameter in addition to the implicit this pointer, representing the left-hand operand. The operator is then applied between the calling object (the implicit operand) and the argument passed to the function (the explicit operand). This enables intuitive operations between objects or between an object and another value, enhancing code readability and flexibility.

**Stream insertion and extraction operators**

The stream insertion (<<) and extraction (>>) operators are overloaded to enable input and output of object data using standard streams like cin and cout. Unlike most operators, these must be defined as friend functions because the left-hand operand (such as cout or cin) is not an object of the user-defined class. By making the function a friend, it can access the private members of the class to read or write data, allowing seamless integration with C++ stream syntax for object I/O.