Experiment No- 4 **Date-**

Aim – To study fundamentals of Operator overloading

Theory –

Operator Overloading

Operator Overloading is a feature in C++ that allows developers to redefine the way operators work for user-defined types, such as classes and structures. By overloading an operator, you can define custom behavior for standard operations (like addition, subtraction, comparison, etc.) when applied to objects of your class.

For example, you might overload the + operator to add two objects of a Vector class, making it possible to use vector1 + vector2 in a way that makes sense for your specific type. Operator overloading helps make your code more intuitive and readable, as it allows objects to interact using familiar syntax.

However, it's important to use operator overloading judiciously to ensure that the overloaded operators behave in a manner consistent with their conventional meanings, avoiding confusion for anyone reading your code.

Types of operator overloading

Operator overloading in C++ can be classified into several types based on the nature of the operator and how it's used. Here's a brief overview:

1. Unary Operator Overloading

Unary operators operate on a single operand. Common examples include ++, --, !, and -. When overloaded, these operators can modify or return a value from the object on which they are invoked.

Basic Syntax:

```
class ClassName {
public:
  // Overloading the unary minus (-) operator
  ClassName operator-() {
    // Implementation logic
  }
};
```

2. Binary Operator Overloading

Binary operators operate on two operands. Examples include +, -, *, /, ==, and !=. Overloading these operators allows custom behavior when two objects of the same class are involved.

Basic Syntax:

```
class ClassName {
```

```
public:
  // Overloading the addition (+) operator
  ClassName operator+(const ClassName &obj) {
    // Implementation logic
  }
};
```

3. Relational Operator Overloading

Relational operators, such as ==, !=, <, >, <=, and >=, compare two operands. Overloading these operators is useful for comparing objects of a class.

Basic Syntax:

```
class ClassName {
public:
  // Overloading the equality (==) operator
  bool operator==(const ClassName &obj) {
    // Implementation logic
  }
};
```

4. Stream Operator Overloading

Stream operators << (output) and >> (input) are used for input and output operations. Overloading these operators enables custom input/output functionality for objects of a class.

Basic Syntax:

```
#include <iostream>
class ClassName {
public:
  // Overloading the output (<<) operator
  friend std::ostream& operator<<(std::ostream &out, const ClassName &obj) {
    // Implementation logic
  }
  // Overloading the input (>>) operator
  friend std::istream& operator>>(std::istream &in, ClassName &obj) {
    // Implementation logic
  }
};
```

5. Function Call Operator Overloading

The function call operator () can be overloaded to allow objects of a class to be used as if they were functions.

Basic Syntax:

```
class ClassName {
public:
  // Overloading the function call operator
  void operator()(int x) {
    // Implementation logic
  }
};
```

6. Assignment Operator Overloading

The assignment operator = can be overloaded to handle deep copying of objects, ensuring that objects are assigned correctly.

Basic Syntax:

```
class ClassName {
public:
  // Overloading the assignment (=) operator
  ClassName& operator=(const ClassName &obj) {
    // Implementation logic
  }
};
```

7. Subscript Operator Overloading

The subscript operator [] is commonly used to access elements in arrays. By overloading this operator, you can allow objects of a class to be indexed in a similar manner.

Basic Syntax:

```
class ClassName {
public:
  // Overloading the subscript ([]) operator
  int& operator[](int index) {
    // Implementation logic
  }
};
```

Overloaded operators must be defined as a member function or a friend function.

Not all operators can be overloaded (e.g., ::, .*, .).

Overloading should be done carefully to maintain the intuitive behavior of operators.