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.

4)a)Write a C++ program to understand overloading of unary prefix &postfix operators to perform increment and decrement operations on objects

Program :

```
#include <iostream>

using namespace std;


class Counter {

   int count;

public:

   Counter(int c) {   count = c; } // Basic
Constructor

   Counter& operator++() {

      cout << "Before Prefix Increment: " <<
count << endl;

      ++count;

      return *this;

   }  // Prefix ++

   Counter operator++(int) {

      cout << "Before Postfix Increment: " <<
count << endl;

      Counter temp = *this;

      ++count;

 return temp;

   }  // Postfix ++

   Counter& operator--() {

      cout << "Before Prefix Decrement: " <<
count << endl;

      --count;

      return *this;

   }  // Prefix --

   Counter operator--(int) {

      cout << "Before Postfix Decrement: " <<
count << endl;

      Counter temp = *this;

      --count;

      return temp;

   }  // Postfix --

   void display() { cout << count << endl; }

};

int main() {

   int userInput;

   cout << "Initial count: ";

   cin >> userInput;

   Counter c(userInput);

   c.display();

   ++c; // Prefix Increment

   c.display();

   c++; // Postfix Increment

   c.display();

   --c; // Prefix Decrement

   c.display();

   c--; // Postfix Decrement

   c.display();

   return 0;}
```

Output

```
Initial count: 6
6
Before Prefix Increment: 6
7
Before Postfix Increment: 7
8
Before Prefix Decrement: 8
7
Before Postfix Decrement: 7
6
```

4)b)Write a C++ program to understand overloading of binary operatorto perform the following operations on the object of the class :

i) x =5+y ii)x=x*y iii)x=y-5

Program –

```
#include <iostream>

using namespace std;


class Number {
   int value;
public:
   Number(int v) { value = v; }
   Number operator+(const Number& other) const
{   return Number(value + other.value);  }
   Number operator*(const Number& other) const
{   return Number(value * other.value);
   }
   Number operator-(int n) const {
      return Number(value - n);  }
   friend Number operator+(int n, const Number&
obj) {
      return Number(n + obj.value);}
   void display() const {
      cout << "Result: " << value << endl;  }
   Number reset(int v) const {
      return Number(v);
   }
};

int main() {
   int input1, input2;
   cout << "Enter two numbers: ";
   cin >> input1 >> input2;
   Number x(input1), y(input2);
   Number originalX = x;
   cout << "Performing x = 5 + y:" << endl;
   x = 5 + y;
   x.display();
   x = originalX;
   cout << "Performing x = x * y:" << endl;
   x = x * y;
   x.display();
   x = originalX;
   cout << "Performing x = y - 5:" << endl;
   x = y - 5;
   x.display();
   return 0;
}
```

OUTPUT

```
Enter two numbers: 6
7
Performing x = 5 + y:
Result: 12
Performing x = x * y:
Result: 42
Performing x = y - 5:
Result: 2
```

4)c)Write a C++ program to overload binary steam insertion (<<) and extraction (>>) operators when used with objects

Program-

```
#include <iostream>

using namespace std;


class Number {

   int value;

public:

   Number(int v) { value = v; } // Basic Constructor with explicit value

   // Overload stream insertion operator

   friend ostream& operator<<(ostream& os, const Number& num) {

      os << num.value;

      return os;

   }

   // Overload stream extraction operator

   friend istream& operator>>(istream& is, Number& num) {

      is >> num.value;

      return is;  }};


int main() {

   Number num(0); // Initialize with a default value

   cout << "Enter a number: ";

   cin >> num; // Use overloaded >> operator

   cout << "You entered: " << num << endl; // Use overloaded << operator

   return 0;

}
```

Output

```
Enter a number: 65
You entered: 65
```

4)d)Write a C++ program using class String to create two stringsand perform the following operations on the strings

i)to add two String type objects (s1=s2+s3) where s1,s2,s3 are objects

ii)To compare two string length to print which string is smaller and print accordingly

Program –

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

class String {
  char* str;
public:
  String(const char* s = "") {
    str = new char[strlen(s) + 1];
    strcpy(str, s);
  }
  String(const String& other) {
    str = new char[strlen(other.str) + 1];
    strcpy(str, other.str);
  }
  ~String() {
    delete[] str;
  }
  String operator+(const String& other) const {
    char* temp = new char[strlen(str) +
strlen(other.str) + 1];
    strcpy(temp, str);
    strcat(temp, other.str);
    String result(temp);
    delete[] temp;
    return result;
  }
  size_t length() const {
    return strlen(str); }
  void display() const {
    cout << str;}};

int main() {
  char input2[100], input3[100];
  cout << "Enter the second string: ";
  cin.getline(input2, 100);
  cout << "Enter the third string: ";
  cin.getline(input3, 100);
  String s2(input2), s3(input3);
  String s1 = s2 + s3;
  cout << "Concatenated String (s1 = s2 + s3): ";
  s1.display();
  cout << endl;
  if (s2.length() < s3.length()) {
    cout << "The second string is shorter than the
third string." << endl;
  } else if (s2.length() > s3.length()) {
    cout << "The second string is longer than the
third string." << endl; } else {
    cout << "The second and third strings are of
equal length." << endl; }
  return 0;
}
```

Output –

```
Enter the second string: Good
Enter the third string: Morning
Concatenated String (s1 = s2 + s3): Good Morning
The second string is shorter than the third string.
```

4)e)Write a C++ program to create a vectorof 'n' elements(allocate the memory dynamically) and then multiply a scalar vector with each element of a vector .Also show the result of addition of two vectors .

Program –

```
#include <iostream>

using namespace std;


class Vector {
   int* elements;
   int size;
public:
   Vector(int n) : size(n) {
      elements = new int[size];
   }
   ~Vector() {
      delete[] elements; }
   void inputElements() {
      for (int i = 0; i < size; ++i) {
         cin >> elements[i];
      } }
   void multiplyByScalar(int scalar) {
      for (int i = 0; i < size; ++i) {
         elements[i] *= scalar;
      } }
   void add(const Vector& other, Vector& result)
const {
      for (int i = 0; i < size; ++i) {
         result.elements[i] = elements[i] +
other.elements[i]; }}
   void display() const {
      for (int i = 0; i < size; ++i) {
         cout << elements[i] << " ";}
      cout << endl;}};

int main() {
   int n, scalar;
   cout << "Enter number of elements: ";
   cin >> n;
   Vector vec1(n), vec2(n), result(n);
   cout << "Enter elements of the first vector: ";
   vec1.inputElements();
   cout << "Enter elements of the second vector: ";
   vec2.inputElements();
   cout << "Enter scalar value: ";
   cin >> scalar;
   vec1.multiplyByScalar(scalar);
   cout << "First vector after multiplication by
scalar: ";
   vec1.display();
   vec1.add(vec2, result);
   cout << "Resultant vector after addition: ";
   result.display();
   return 0;
}
```

OUTPUT

```
Enter number of elements: 3
Enter elements of the first vector: 44
34
34
Enter elements of the second vector: 23
45
33
Enter scalar value: 6
```

4)f)Create a class Polar that represents the points on the plainas polar coordinates (radius and angle).Create a overloaded +operator for addition of two Polar quantities .”Adding” two points on the plain can be accomplished by adding their X coordinates and then adding their Y coordinates.This gives the X and Y coordinates of the “answer”.Thus you’ll need to convert two sets of polar coordinates to rectangular coordinates,add them ,then convert the resulting rectangular representation back to polar .

Program –

```
#include <iostream>

#include <cmath>

using namespace std;


class polar ;

class Rectangular {

float x ;

float y ;

public :

Rectangular operator +(Rectangular &a){

Rectangular tmp ;

tmp.x = x+a.x ;

tmp.y=y+a.y ;

return tmp ;

}

void friend polrtorect (Rectangular &a,polar &b);

void friend recttopolar (Rectangular &a,polar &b);

};
```

```
class polar {
   float radius ;
   float angle ;
   public :
 void getvalue (){
cout <<"Enter the  radius the point makes from origin  \n" ;
cin>>radius ;
cout <<"Enter angle it is inclined from the origin \n" ;
cin>>angle;
}
void display (){
   cout<<"The radius the point makes from origin is "<<radius<<" Units"<<" and it is inclined by "<<angle<<" radians"<<endl;
}
void friend polrtorect (Rectangular &a,polar &b);
void friend recttopolar (Rectangular &a,polar &b);
};
void recttopolar (Rectangular &a,polar &b){
   float r,ang ;
   r = (a.x*a.x) + (a.y*a.y) ;
   r = sqrt(r) ;
   ang = a.x/a.y ;
   ang = atan(ang) ;   //atan() function is used to calculate tan inverse of a function
   b.radius = r ;
   b.angle = ang ;
}
void polrtorect (Rectangular &a,polar &b){
  a.x = (b.radius)*(cos(b.angle)) ;
  a.y = (b.radius)*(sin(b.angle)) ;
}

int main (){
polar P,Q,R;
```

```
Rectangular A,B,C ;
P.getvalue() ;
Q.getvalue() ;
polrtorect(A,P) ;
polrtorect(B,Q) ;
C = A+B ;
recttopolar(C,R) ;
R.display();
   return 0;
}
```

Output –

```
Enter the  radius the point makes from origin
5
Enter angle it is inclined from the origin
76
Enter the  radius the point makes from origin
6
Enter angle it is inclined from the origin
54
The radius the point makes from origin is 1.00117 Units and it is inclined by 1.02209 radians
```

Conclusion – All the programs were successfully executed without error