# **Unit-3**

**Experiment No: 1**

**Aim:**

Unary & Binary Operator Overloading – Member Function

**Description:**  
This program demonstrates **operator overloading** in C++ using a **member function**, including unary (++) and binary (+) operators.

**Source Code:**

#include <iostream>

using namespace std;

class Number {

public:

int value;

Number(int v = 0) { value = v; }

void operator++() { ++value; } // Unary operator overloading

Number operator+(Number obj) { // Binary operator overloading

Number temp;

temp.value = value + obj.value;

return temp;

}

void display() { cout << "Value: " << value << endl; }

};

int main() {

Number n1(5), n2(10), n3;

cout << "Before Unary Operation:" << endl;

n1.display();

++n1;

cout << "After Unary Operation (++n1):" << endl;

n1.display();

n3 = n1 + n2;

cout << "After Binary Operation (n1 + n2):" << endl;

n3.display();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated **unary and binary operator overloading**.

**Experiment No: 2**

**Aim:**

Unary & Binary Operator Overloading –Friend Function

**Description:**  
This program demonstrates **operator overloading** in C++ using a **friend function**, including unary (++) and binary (+) operators.

**Source Code:**

#include <iostream>

using namespace std;

class Number

{ public: int value;

Number(int v = 0) // Constructor

{

value = v;

}

// Friend function declarations

friend Number operator++(Number &n); // Unary

friend Number operator+(Number n1, Number n2); // Binary

void display()

{

cout << "Value: " << value << endl;

}

};

// Unary operator overloading (prefix ++)

Number operator++(Number &n) {

n.value++;

return n;

}

// Binary operator overloading (+)

Number operator+(Number n1, Number n2) {

return Number(n1.value + n2.value);

}

int main() {

Number n1(5), n2(10), n3;

cout << "Before Unary Operation:" << endl;

n1.display();

// Unary operator overloading

++n1;

cout << "After Unary Operation (++n1):" << endl;

n1.display();

// Binary operator overloading

n3 = n1 + n2;

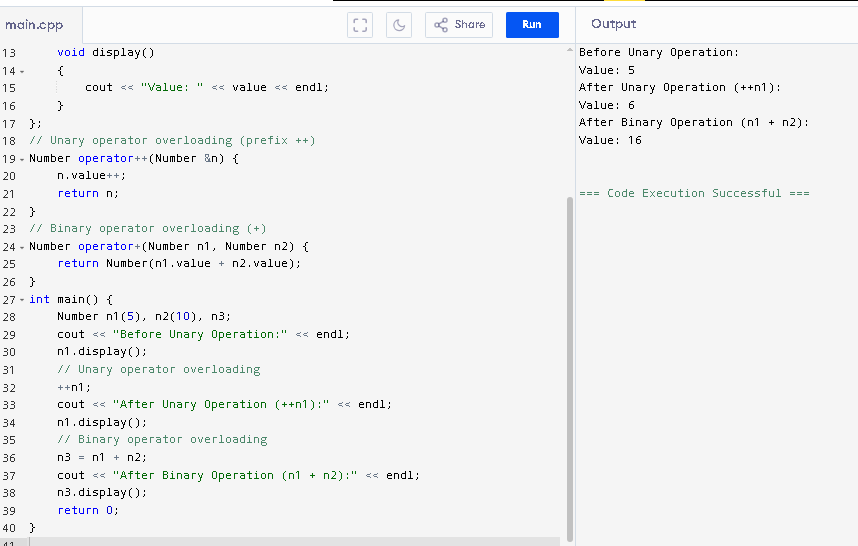
cout << "After Binary Operation (n1 + n2):" << endl;

n3.display();

return 0;

}

## **Sample Input and Output:**



**Result:**  
The program executed successfully and demonstrated **unary and binary operator overloading**.

**Experiment No: 3**

**Aim:**

Inheritance – Types

**Description:**  
This program demonstrates different **types of inheritance** in C++: Single, Multiple, Multi-level, Hierarchical, and Hybrid inheritance.

**Source Code:**

#include <iostream>

using namespace std;

class A

{ public:

void displayA()

{

cout << "This is class A (Base Class - Single Inheritance)" << endl;

}

};

class B : public A

{ public:

void displayB()

{

cout << "This is class B (Derived from A - Single Inheritance)" << endl;

}

};

class X

{ public:

void displayX() {

cout << "This is class X (Base Class for Multiple Inheritance)" << endl;

}

};

class Y

{ public:

void displayY() {

cout << "This is class Y (Base Class for Multiple Inheritance)" << endl;

}

};

class Z : public X, public Y //Multiple Inheritance

{ public:

void displayZ() {

cout << "This is class Z (Derived from X and Y - Multiple Inheritance)" << endl;

}

};

class Parent

{ public:

void displayParent() {

cout << "This is Parent Class (Multi-level Inheritance)" << endl;

}

};

class Child : public Parent //Multi-level Inheritance

{ public:

void displayChild() {

cout << "This is Child Class (Derived from Parent)" << endl; }

};

class GrandChild : public Child //Multi-level Inheritance

{ public:

void displayGrandChild() {

cout << "This is GrandChild Class (Derived from Child - Multi-level)" << endl;

}

};

class Base

{ public:

void displayBase() {

cout << "This is Base Class (Hierarchical Inheritance)" << endl;

}

};

class Derived1 : public Base //Hierarchical Inheritance

{ public:

void displayDerived1() {

cout << "This is Derived1 (from Base)" << endl;

}

};

class Derived2 : public Base //Hierarchical Inheritance

{

public:

void displayDerived2() {

cout << "This is Derived2 (from Base)" << endl;

}

};

class M

{

public:

void displayM() {

cout << "This is class M (Base of Hybrid Inheritance)" << endl;

}

};

class N : public M //Hybrid Inheritance

{

public:

void displayN() {

cout << "This is class N (Derived from M)" << endl;

}

};

class O {

public:

void displayO() {

cout << "This is class O (Independent Base for Hybrid Inheritance)" << endl;

}

};

class P : public N, public O //Hybrid Inheritance

{ public:

void displayP() {

cout << "This is class P (Derived from both N and O - Hybrid)" << endl;

}

};

int main()

{ cout << "\n--- Single Inheritance ---" << endl;

B b;

b.displayA();

b.displayB();

cout << "\n--- Multiple Inheritance ---" << endl;

Z z;

z.displayX();

z.displayY();

z.displayZ();

cout << "\n--- Multi-level Inheritance ---" << endl;

GrandChild gc;

gc.displayParent();

gc.displayChild();

gc.displayGrandChild();

cout << "\n--- Hierarchical Inheritance ---" << endl;

Derived1 d1;

Derived2 d2;

d1.displayBase();

d1.displayDerived1();

d2.displayBase();

d2.displayDerived2();

cout << "\n--- Hybrid Inheritance ---" << endl;

P p;

p.displayM();

p.displayN();

p.displayO();

p.displayP();

return 0;

}

## **Sample Input and Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Result:**  
The program executed successfully and demonstrated **all types of inheritance**.

**Experiment No: 4**

**Aim:**

Constructor and Destructor Order

**Description:**  
This program demonstrates how **constructors and destructors** are called in inheritance hierarchy.

**Source Code:**

#include <iostream>

using namespace std;

class Base {

public:

Base() { cout << "Base class constructor called" << endl; }

~Base() { cout << "Base class destructor called" << endl; }

};

// Intermediate Derived class

class Derived1 : public Base {

public:

Derived1() { cout << "Derived1 class constructor called" << endl; }

~Derived1() { cout << "Derived1 class destructor called" << endl; }

};

// Further Derived class

class Derived2 : public Derived1 {

public:

Derived2() { cout << "Derived2 class constructor called" << endl; }

~Derived2() { cout << "Derived2 class destructor called" << endl; }

};

int main() {

cout << "Creating object of Derived2 class..." << endl;

Derived2 obj;

cout << "Exiting main..." << endl;

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated the **order of constructor and destructor calls**.

**Experiment No: 5\_1**

**Aim:**

Object as a Class Member

**Description:**  
This program demonstrates how an **object can be used as a member of another class**.

**Source Code:**

#include <iostream>

using namespace std;

class Address {

string city;

public:

Address(string c) { city = c; }

void show() { cout << "City: " << city << endl; }

};

class Student {

string name;

Address addr;

public:

Student(string n, string c) : name(n), addr(c) {}

void display() {

cout << "Name: " << name << endl;

addr.show();

}

};

int main() {

Student s("Kavitha", "Ramavaram");

s.display();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated **object as a member of a class**.

**Experiment No: 5\_2**

**Aim:**

Pointer to a Class

**Description:**  
This program demonstrates how a **pointer can be used to access class members**.

**Source Code:**

#include <iostream>

using namespace std;

class Box {

int length;

public:

void setLength(int l) { length = l; }

void showLength() { cout << "Length: " << length << endl; }

};

int main() {

Box b1;

Box \*ptr = &b1;

ptr->setLength(15);

ptr->showLength();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated the concept of **pointer to a class**.

**Experiment No: 5\_3**

**Aim:**

Demonstration of "this" Pointer

**Description:**  
This program demonstrates how the **this pointer** is used to resolve ambiguity between class members and local variables.

**Source Code:**

#include <iostream>

using namespace std;

class Test {

int x;

public:

Test(int x) { this->x = x; } // Resolving ambiguity

void show() { cout << "Value of x: " << this->x << endl; }

};

int main() {

Test t(50);

t.show();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated the **use of this pointer**.

**Experiment No: 5\_4**

**Aim:**

Demonstration of Virtual Base Class

**Description:**  
This program demonstrates **virtual base classes** in C++ to avoid ambiguity in multiple inheritance.

**Source Code:**

#include <iostream>

using namespace std;

class Person {

public:

void show()

{ cout << "I am a Person" << endl;

}

};

class Student : virtual public Person {

public:

void studentInfo()

{ cout << "I am a Student" << endl; } };

class Teacher : virtual public Person

{ public:

void teacherInfo()

{ cout << "I am a Teacher" << endl; } };

class TA : public Student, public Teacher {

public:

void display() { show(); }

};

int main() {

TA obj;

obj.display();

obj.studentInfo();

obj.teacherInfo();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated the **use of virtual base class**.

**Experiment No: 6**

**Aim:**

Demonstration of Virtual Functions

**Description:**  
This program demonstrates **runtime polymorphism** in C++ using **virtual functions**.

**Source Code:**

#include <iostream>

using namespace std;

class Base {

public:

virtual void display() { cout << "Display from Base class" << endl; }

};

class Derived1 : public Base {

public:

void display() override { cout << "Display from Derived1 class" << endl; }

};

class Derived2 : public Base {

public:

void display() override { cout << "Display from Derived2 class" << endl; }

};

int main() {

Base\* basePtr;

Base b; Derived1 d1; Derived2 d2;

basePtr = &b; basePtr->display();

basePtr = &d1; basePtr->display();

basePtr = &d2; basePtr->display();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated **virtual functions and polymorphism**.

**Experiment No: 7**

**Aim:**

Demonstration of Pure Virtual Function (Abstract Class)

**Description:**  
This program demonstrates **abstract classes and pure virtual functions** in C++.

**Source Code:**

#include <iostream>

#include <cmath>

using namespace std;

class Shape {

public:

virtual void area() = 0; // Pure virtual function

};

class Circle : public Shape {

float radius;

public:

Circle(float r) : radius(r) {}

void area() { cout << "Area of Circle = " << 3.14159 \* radius \* radius << endl; }

};

class Rectangle : public Shape {

float length, breadth;

public:

Rectangle(float l, float b) : length(l), breadth(b) {}

void area() { cout << "Area of Rectangle = " << length \* breadth << endl; }

};

class Triangle : public Shape {

float base, height;

public:

Triangle(float b, float h) : base(b), height(h) {}

void area() { cout << "Area of Triangle = " << 0.5 \* base \* height << endl; }

};

int main() {

Shape \*s;

Circle c(5);

Rectangle r(4, 6);

Triangle t(3, 7);

s = &c; s->area();

s = &r; s->area();

s = &t; s->area();

return 0;

}

## **Sample Input and Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Result:**  
The program executed successfully and demonstrated **pure virtual functions and abstract classes**.