# **Unit-3**

**Experiment No: 1**

**Aim:**

Unary & Binary Operator Overloading – Member Function

**Description:**  
This program demonstrates **operator overloading** in C++.  
The class Number overloads the unary ++ operator to increment the object’s value and the binary + operator to add two Number objects, returning a new object.  
In main(), n1 is incremented with ++n1, and n3 is assigned the sum of n1 and n2 using the overloaded + operator.  
The output shows the value changes after each operation, confirming custom operator behavior.

**Code:**

#include <iostream>

using namespace std;

class Number {

public:

int value;

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

void operator++() { ++value; }

Number operator+(Number obj) {

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;

}

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

A screenshot of a computer

AI-generated content may be incorrect.

**Experiment No: 1**

**Aim:**

Unary & Binary Operator Overloading –Friend Function

**Description:**

This program demonstrates **friend functions for operator overloading** in C++.

* The class Number has a constructor, a value member, and declares two friend functions: one to overload the **unary prefix ++** operator and another to overload the **binary +** operator.
* Because they are friends, these functions can directly access and modify the private data of Number objects.
* In main(), ++n1 increments n1.value, and n1 + n2 creates a new Number whose value is the sum of both, showing how external friend functions can implement custom operator behavior.

**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;

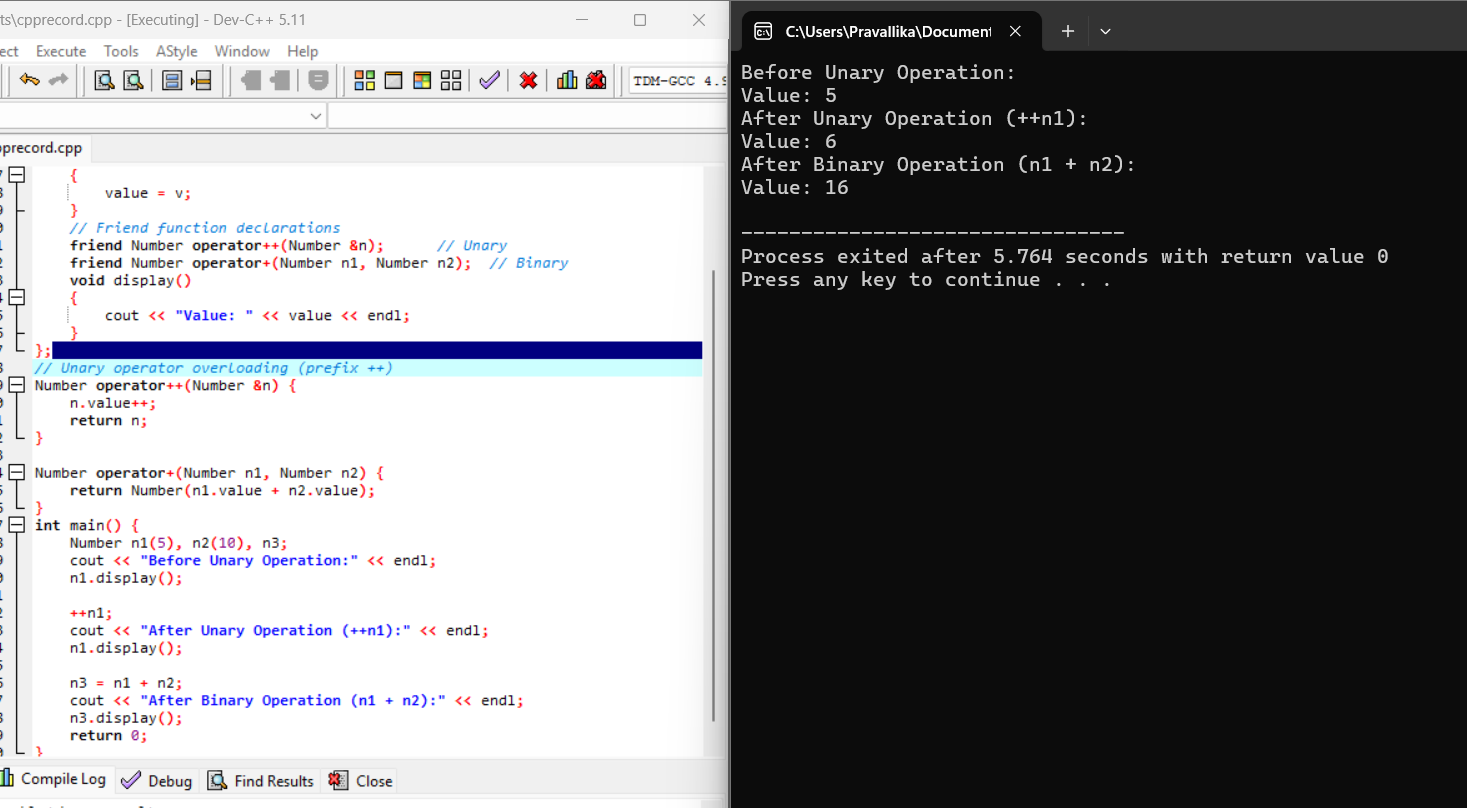
n3.display();

return 0;

}

**Result:**

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



**Experiment No: 3**

**Aim:**

Inheritance – Types

**Description:**  
This program demonstrates **all major types of inheritance in C++** with simple class examples.

* **Single inheritance:** Class B inherits from A, letting B use displayA() and its own displayB().
* **Multiple inheritance:** Class Z inherits from both X and Y, so it can call methods from both bases.
* **Multi-level inheritance:** GrandChild inherits from Child, which inherits from Parent, forming a chain of access to all ancestor methods.
* **Hierarchical inheritance:** Classes Derived1 and Derived2 both inherit from Base, each getting displayBase() plus their own functions.
* **Hybrid inheritance:** Class P inherits from N (which itself inherits from M) and from O, combining multiple and multi-level inheritance in one structure.

**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;

}

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

A screenshot of a computer

AI-generated content may be incorrect.

**Experiment No: 4**

**Aim:**

Constructor and Destructor Order

**Description:**  
This program illustrates **constructor and destructor call order in multi-level inheritance**.

* Derived2 inherits from Derived1, which inherits from Base.
* When obj is created, constructors run **from base to derived**: Base → Derived1 → Derived2.
* When the program ends, destructors run in **reverse order**: Derived2 → Derived1 → Base, showing the cleanup sequence.

**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;

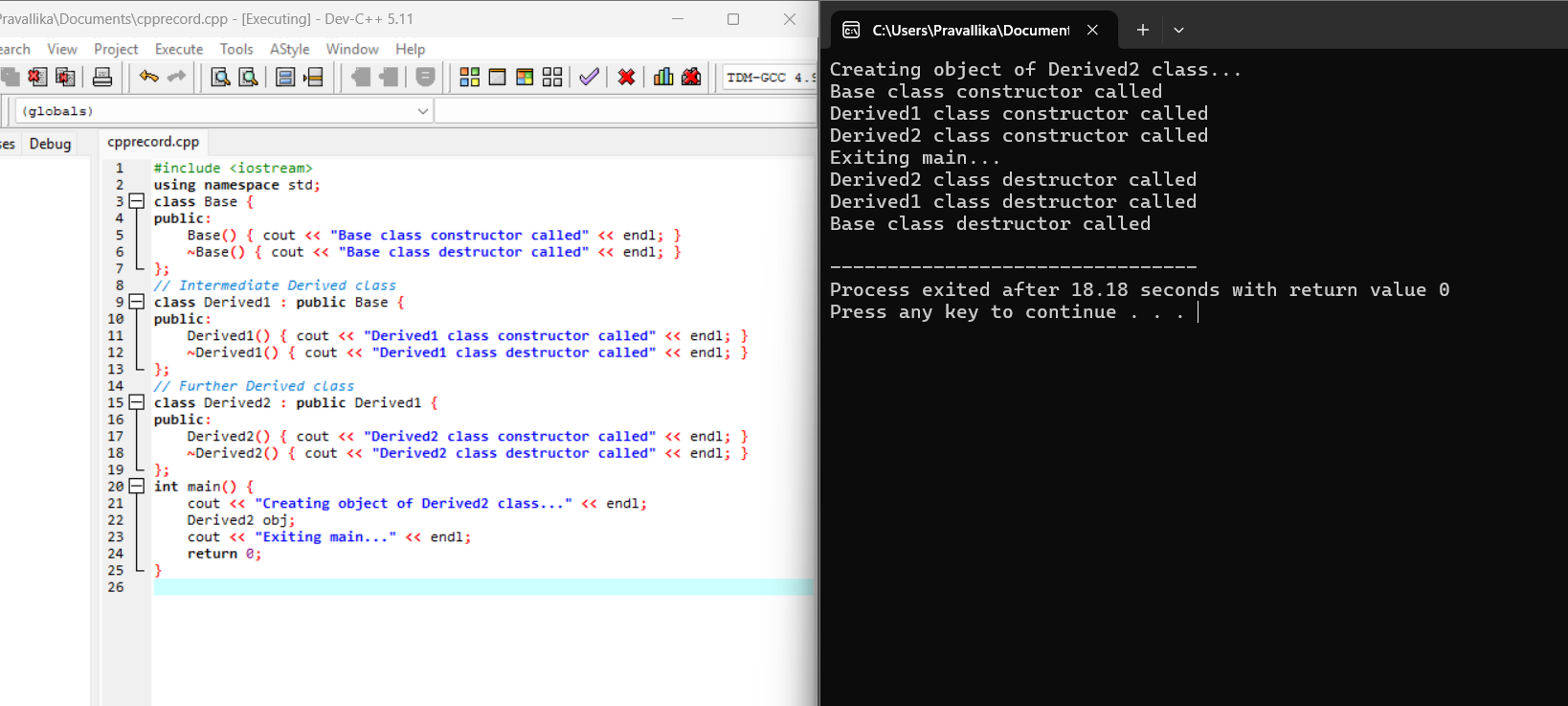
Derived2 obj;

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

return 0;

}

**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 **composition (has-a relationship)** in C++.  
The Student class contains an Address object as a member, meaning each student *has an* address.  
The constructor of Student uses an **initializer list** to pass the city name to the Address constructor.  
In main(), a Student object is created with a name and city, and display() prints both the student’s name and their city through the composed Address object.

**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() {

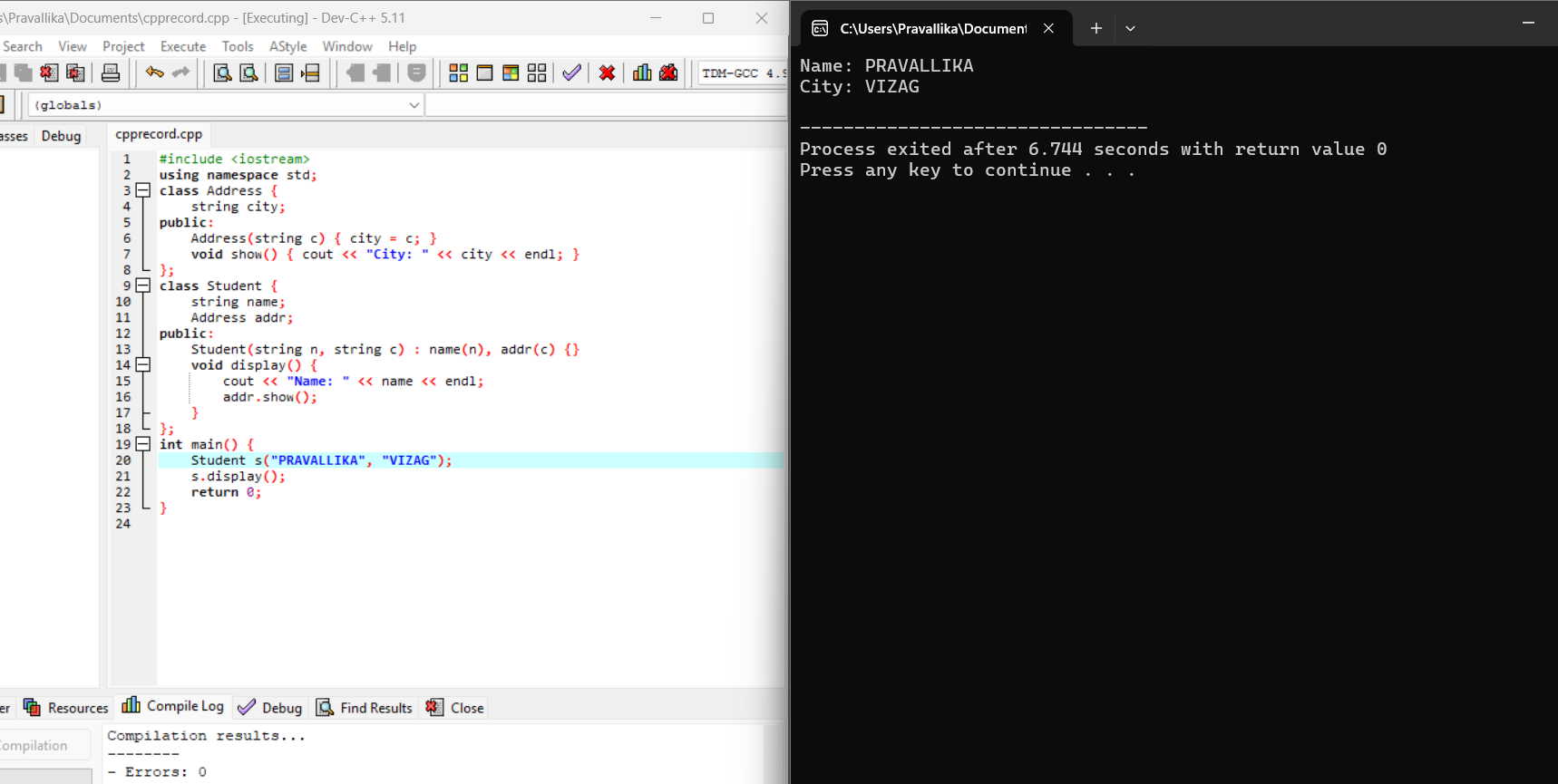
Student s("PRAVALLIKA", "VIZAG");

s.display();

return 0;

}

**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 shows how to use a **pointer to an object** in C++.  
A Box object b1 is created, and a pointer ptr is assigned its address.  
Using the arrow operator (->), the program calls setLength() to set the length and showLength() to display it.  
The output confirms that member functions can be accessed directly through an object pointer.

**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;

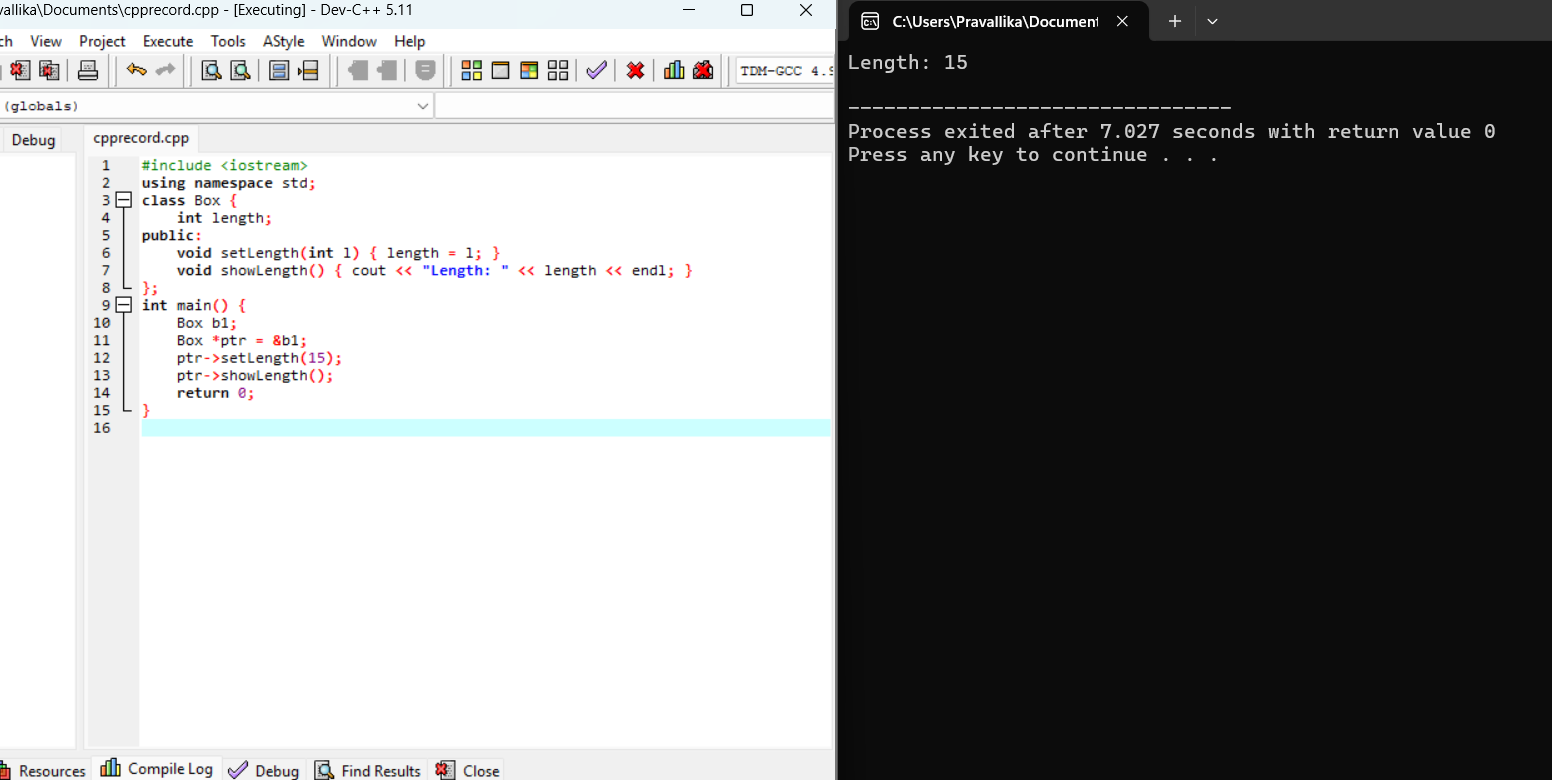
ptr->setLength(15);

ptr->showLength();

return 0;

}

**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 the use of the **this pointer** in C++.  
The constructor parameter and the data member share the same name x, creating ambiguity.  
By writing this->x = x;, the this pointer clarifies that the class member x is being assigned the constructor argument.  
When show() is called, it prints Value of x: 50.

**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() {

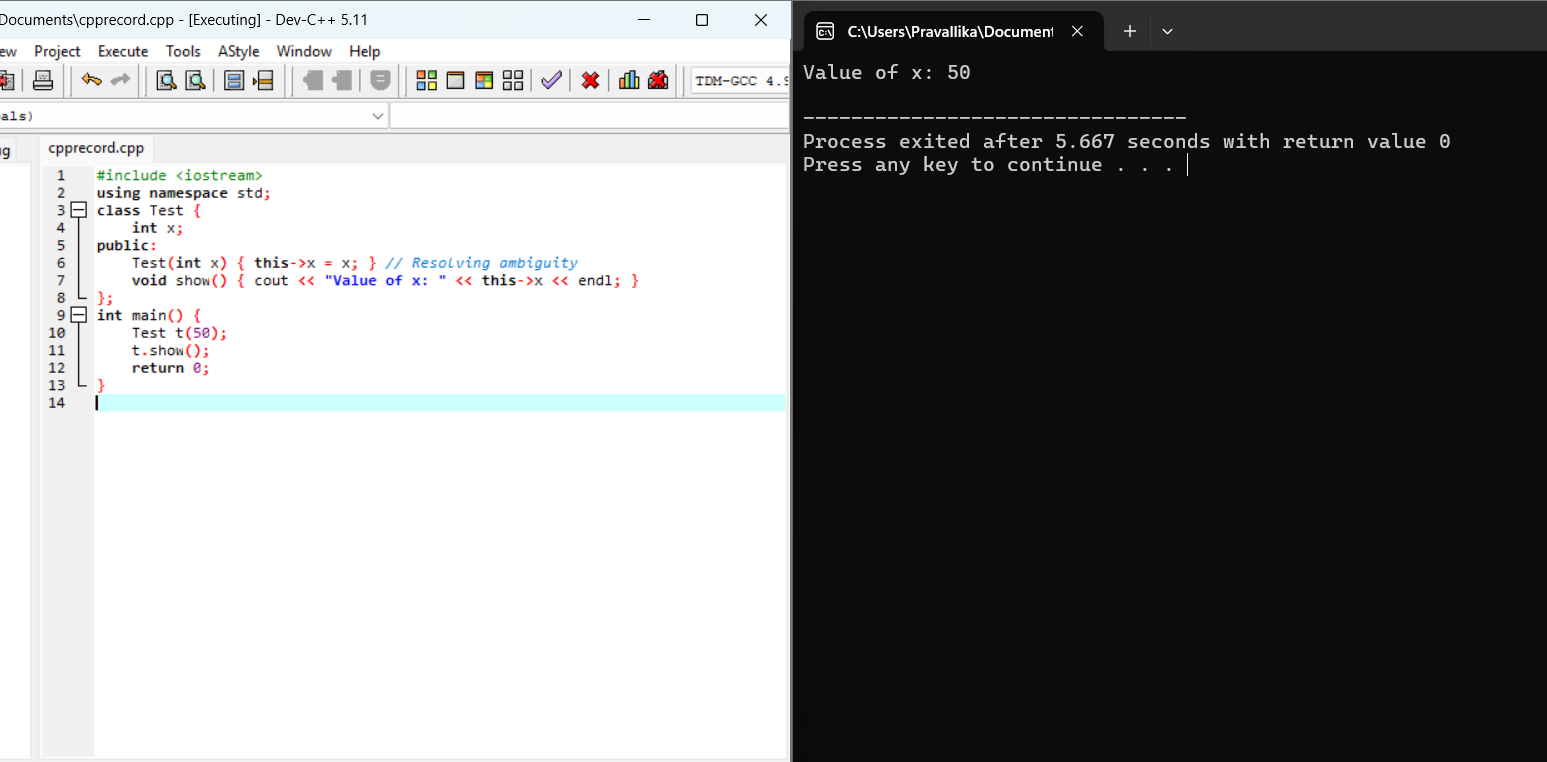
Test t(50);

t.show();

return 0;

}

**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 illustrates the **diamond problem** and how **virtual inheritance** solves it in C++.

* Student and Teacher both inherit from Person, and TA inherits from both Student and Teacher.
* By using virtual public Person, only **one shared copy** of Person exists in TA, preventing duplicate show() members.
* In main(), calling obj.display() successfully calls Person’s show() without ambiguity, and the other functions show the roles of Student and Teacher.

**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();

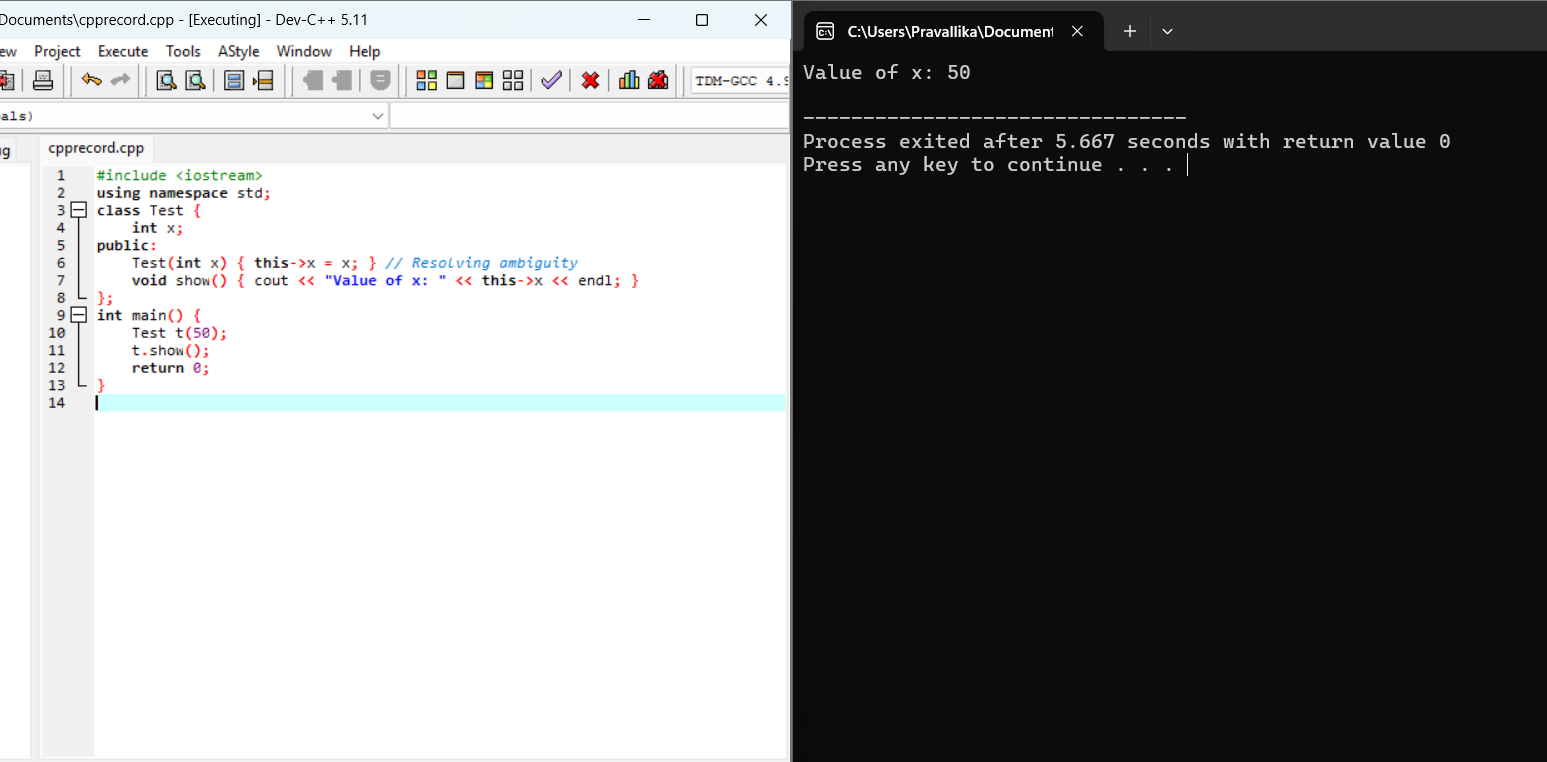
obj.studentInfo();

obj.teacherInfo();

return 0;

}

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



**Experiment No: 6**

**Aim:**

Demonstration of Virtual Functions

**Description:**  
This program illustrates the **diamond problem** and how **virtual inheritance** solves it in C++.

* Student and Teacher both inherit from Person, and TA inherits from both Student and Teacher.
* By using virtual public Person, only **one shared copy** of Person exists in TA, preventing duplicate show() members.
* In main(), calling obj.display() successfully calls Person’s show() without ambiguity, and the other functions show the roles of Student and Teacher.

**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();

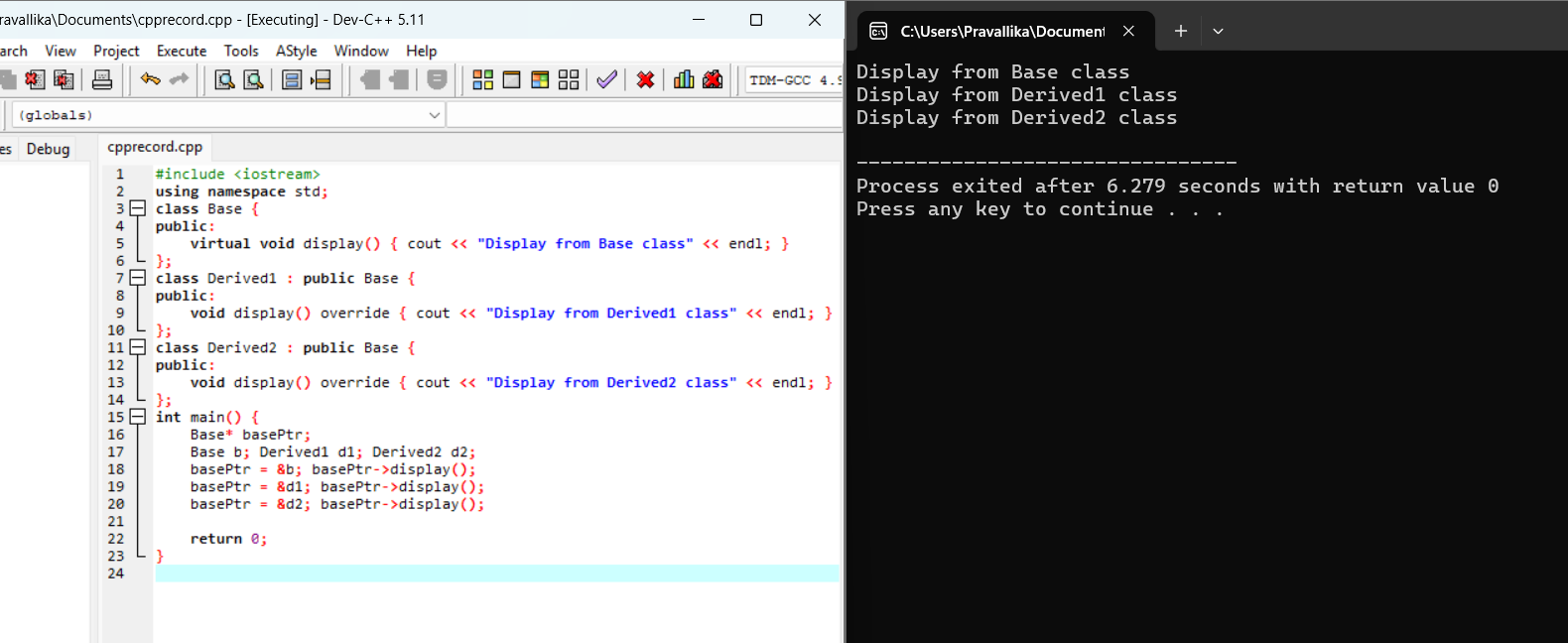
basePtr = &d1; basePtr->display();

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

return 0;

}

**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 **runtime polymorphism using abstract classes** in C++.

* Shape is an **abstract base class** because it has a pure virtual function area().
* Derived classes Circle, Rectangle, and Triangle override area() to calculate their respective areas.
* In main(), a Shape\* pointer is used to point to different derived objects, and the correct area() function is called at runtime, showing dynamic dispatch.

**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;

}

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

