**UNIT-3**

**EXPERIMENT 1**

**AIM:**

To write a C++ program to overload unary and binary operators using member functions.

**DESCRIPTION:**

Operator overloading allows us to redefine the meaning of operators for user-defined types (like classes). This improves code readability and usability for custom objects.

There are two types of operator overloading:

1. Unary Operator Overloading:  
   Operators that act on a single operand (e.g., -, ++, --, !) can be overloaded using member functions with no arguments.
2. Binary Operator Overloading:  
   Operators that act on two operands (e.g., +, -, \*, /) can be overloaded using member functions with one argument.

**PROGRAM:**

#include <iostream>

using namespace std;

class Number

{ public: int value;

Number(int v = 0) // Constructor

{

value = v;

}

void operator++() // Unary operator overloading (increment)

{

++value;

}

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

{ 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();

// Unary operator overloading

++n1;

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

n1.display();

// Binary operator overloading

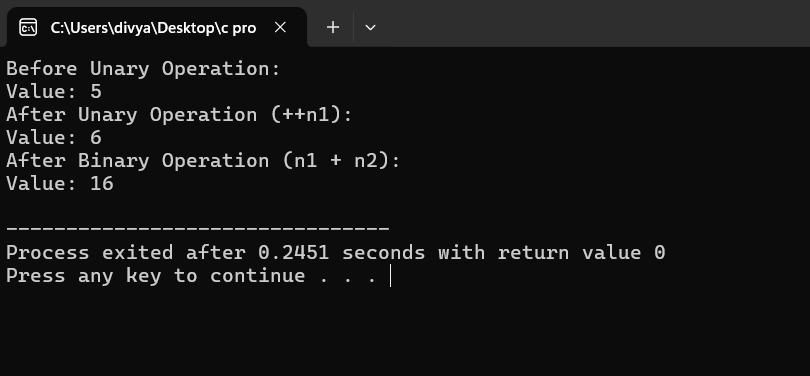
n3 = n1 + n2;

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

n3.display();

return 0;}

**OUTPUT:**



**RESULT:**

The program was successfully compiled and executed.  
It demonstrated:

* Unary operator overloading using a member function with no parameters.
* Binary operator overloading using a member function with one parameter.

**EXPERIMENT 2**

**AIM:**

To write a C++ program to overload unary and binary operators using friend functions.

DESCRIPTION:

In C++, operator overloading allows us to redefine the behavior of operators for user-defined types (like classes).  
A friend function is a non-member function that is given special access to the private members of a class.

Types of Operator Overloading:

* Unary Operator Overloading: Operates on one operand (e.g., -, ++). Using a friend function, it takes one argument (the object).
* Binary Operator Overloading: Operates on two operands (e.g., +, -, \*). Using a friend function, it takes two arguments (the left and right operands).

**PROGRAM:**

// Unary and Binary Operator Overloading - Friend Function

#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;

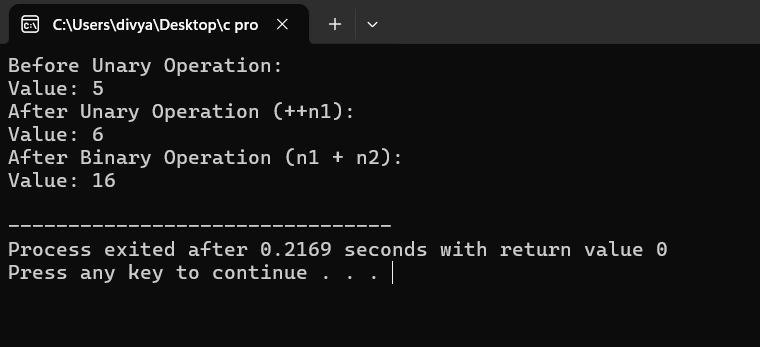
cout << "After Binary Operation (n1 + n2):" << endl;

n3.display();

return 0;

}

**OUTPUT:**



**RESULT:** The program was successfully compiled and executed.

* **Unary operator overloading** using a friend function with one argument.
* **Binary operator overloading** using a friend function with two arguments.

**EXPERIMENT 3**

**AIM:**

To write a C++ program to demonstrate various forms of inheritance, namely:  
(i) Single Inheritance, (ii) Multiple Inheritance, (iii) Multilevel Inheritance,  
(iv) Hierarchical Inheritance, and (v) Hybrid Inheritance.

**DESCRIPTION**:

Inheritance is one of the core features of object-oriented programming (OOP), allowing a class (derived class) to acquire properties and behaviors of another class (base class). This promotes code reusability and hierarchical classification.

Types of Inheritance:

1. Single Inheritance: One derived class inherits from one base class.
2. Multiple Inheritance: One derived class inherits from more than one base class.
3. Multilevel Inheritance: A class is derived from another derived class, forming a chain.
4. Hierarchical Inheritance: Multiple classes inherit from a single base class.
5. Hybrid Inheritance: A combination of more than one type of inheritance.

**PROGRAM:**

// Inheritance - Types

#include <iostream>

using namespace std;

class A

{ public:

void displayA()

{

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

}

};

class B : public A // Single Inheritance

{ 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();

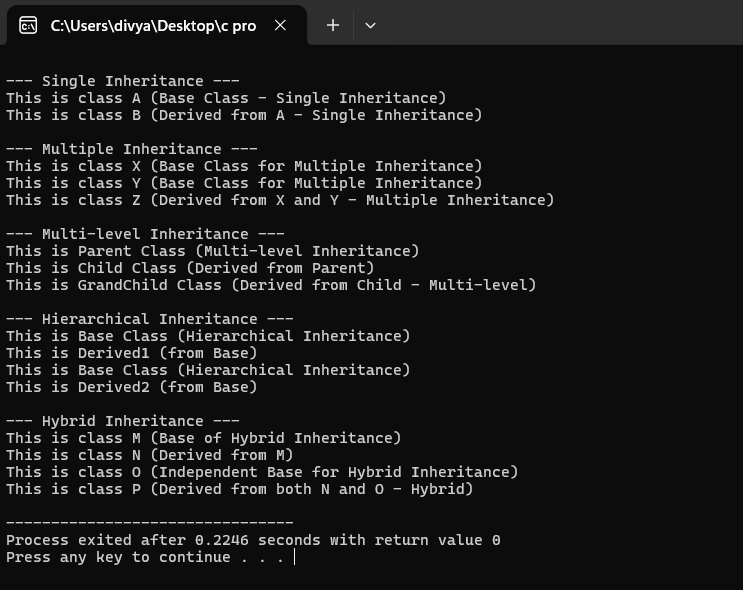
p.displayO();

p.displayP();

return 0;

}

**OUTPUT:**



**RESULT:**

The C++ program was successfully compiled and executed. Thus, all the types of inheritance in C++ were successfully implemented and verified.

**EXPERIMENT 4**

**AIM:**

To write a C++ program to illustrate the order of execution of constructors and destructors in inheritance.

DESCRIPTION:

In C++, when inheritance is involved:

* Constructors are executed in the order of inheritance hierarchy — from the base class to the derived class.
* Destructors are executed in the reverse order — from the derived class to the base class.

This ensures that the base part of an object is initialized first and cleaned up last, maintaining integrity of object construction and destruction.

**PROGRAM:**

#include <iostream>

using namespace std;

// Base class

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;

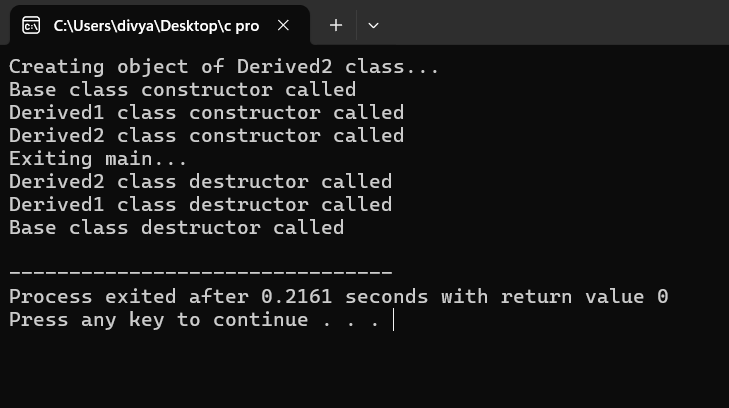
Derived2 obj; // Object creation

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

return 0;

}

OUTPUT:



**RESULT:**

The program was successfully compiled and executed.

* **Constructor Execution Order**: Base → Derived
* **Destructor Execution Order**: Derived → Base

**EXPERIMENT 5**

**AIM:** To write a C++ program to illustrate the use of:

1. Objects as class members
2. Pointer to a class
3. this pointer
4. Virtual base class

**DESCRIPTION:**

1. Objects as Class Members:  
   A class can have objects of another class as its members. This promotes composition and better code organization.
2. Pointer to a Class:  
   A pointer can point to an object of a class, allowing dynamic access to members using the arrow operator (->).
3. this Pointer:  
   this is an implicit pointer available in all non-static member functions that points to the calling object.
4. Virtual Base Class:  
   In multiple inheritance, a virtual base class ensures that only one copy of the base class is inherited by the grandchild derived class, preventing ambiguity (diamond problem).

**PROGRAM:**

1) // Object as a Class member

#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; // Object as a class member

public:

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

void display() {

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

addr.show();

}

};

int main() {

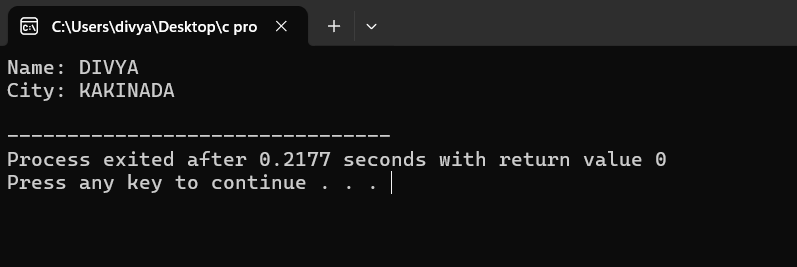
Student s("Ravi", "Hyderabad");

s.display();

return 0;

}

**OUTPUT:**



2) // Demonstration :: Pointer to a Class

#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; // Pointer to object

ptr->setLength(15);

ptr->showLength();

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

3) // Demonstration of "this" Pointer

#include <iostream>

using namespace std;

class Test {

int x;

public:

Test(int x) // local varible is X , argument also X so, this pointer will be used

{

this->x = x; // Resolving ambiguity

}

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

};

int main() {

Test t(50);

t.show();

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

4) // Demonstrtion of Virtual Base Class

#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(); // Only one copy of Person is inherited

}

};

int main() {

TA obj;

obj.display();

obj.studentInfo();

obj.teacherInfo();

return 0;

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**RESULT:**

The program was compiled and executed successfully. It demonstrated:

* Use of objects as class members to implement composition.
* Use of pointer to a class to access object members dynamically.
* Use of the this pointer to resolve naming conflicts in member functions.
* Use of virtual base class to avoid ambiguity in multiple inheritance (diamond problem).

Thus, the concepts of objects as class members, pointers to class, this pointer, and virtual base classes were successfully implemented and verified.

**EXPERIMENT 6**

**AIM:**

To write a C++ program to illustrate the use of virtual functions.

**DESCRIPTION:**

A virtual function is a member function in a base class that is overridden in a derived class and is declared using the virtual keyword.

Virtual functions enable runtime polymorphism in C++. When a base class pointer or reference points to a derived class object, the appropriate overridden function is called based on the actual object, not the pointer type.

🔹 Key Points:

* Declared using the keyword virtual in the base class.
* Supports dynamic binding / late binding.
* Must be accessed through a base class pointer or reference.

**PROGRAM:**

// Demonstration of Virtual Functions

#include <iostream>

using namespace std;

// Base Class

class Base {

public:

virtual void display() { // Virtual Function

cout << "Display from Base class" << endl;

}

};

// Derived Class 1

class Derived1 : public Base {

public:

void display() override {

cout << "Display from Derived1 class" << endl;

}

};

// Derived Class 2

class Derived2 : public Base {

public:

void display() override {

cout << "Display from Derived2 class" << endl;

}

};

int main() {

Base\* basePtr; // Base class pointer

Base b;

Derived1 d1;

Derived2 d2;

// Base pointer pointing to Base object

basePtr = &b;

basePtr->display();

// Base pointer pointing to Derived1 object

basePtr = &d1;

basePtr->display();

// Base pointer pointing to Derived2 object

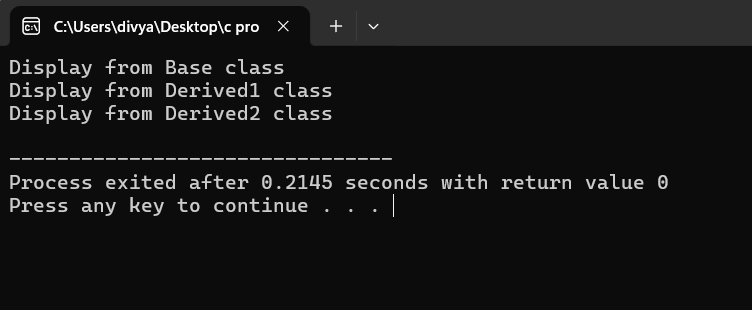
basePtr = &d2;

basePtr->display();

return 0;

}

**OUTPUT:**



RESULT: The program was successfully compiled and executed. Thus, the concept of **virtual functions** was successfully implemented and verified.

**EXPERIMENT 7**

**AIM:**

To write a C++ program to implement pure virtual functions and calculate the area of different shapes using an abstract class.

**DESCRIPTION:**

In C++, a pure virtual function is a function that is declared in a base class but has no definition in the base class. It is specified using = 0 in the declaration.

A class that contains at least one pure virtual function is called an abstract class, and it cannot be instantiated directly.

This approach is useful when defining a common interface for all derived classes, where each derived class provides its own implementation of the pure virtual function.

🔹 Syntax:

class Base {

public:

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

};

**PROGRAM:**

// Pure Virtual Function

#include <iostream>

#include <cmath>

using namespace std;

// Abstract Base Class

class Shape {

public: // Pure virtual function

virtual void area() = 0; };

// Derived Class: Circle

class Circle : public Shape {

float radius;

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

void area() {

cout << "Area of Circle = " << 3.14159 \* radius \* radius << endl;

} };

// Derived Class: Rectangle

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;

} };

// Derived Class: Triangle

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; // Base class pointer

Circle c(5);

Rectangle r(4, 6);

Triangle t(3, 7);

s = &c;

s->area();

s = &r;

s->area();

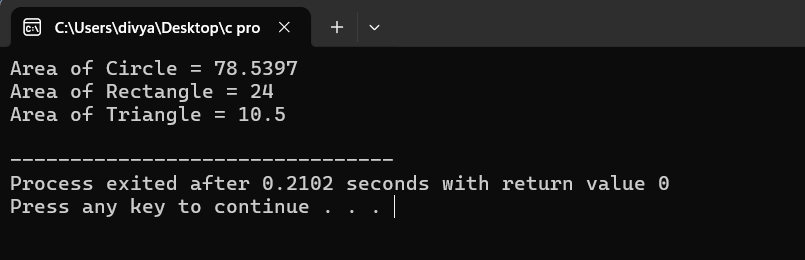
s = &t;

s->area();

return 0;

}

**OUTPUT:**



**RESULT:**

The program was successfully compiled and executed.  
It demonstrated the use of a pure virtual function in an abstract base class, and calculated the area of different shapes (Circle, Rectangle, and Triangle) through runtime polymorphism.  
Thus, the concept of abstract classes and pure virtual functions was successfully implemented and verified.