**1. Write a C++ program to create a base class Person with attributes name and age. Derive a class Student that adds rollNo. Use constructors to initialize all attributes. Create objects of both classes and display their detail.**

#include <iostream>

#include <string>

using namespace std;

class Person {

protected:

string name;

int age;

public:

Person(string n, int a) : name(n), age(a) {}

void display() const {

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

}

};

class Student : public Person {

private:

int rollNo;

public:

Student(string n, int a, int r) : Person(n, a), rollNo(r) {}

void display() const {

cout << "Student Details:" << endl;

Person::display();

cout << "Roll No: " << rollNo << endl;

}

};

int main() {

string name;

int age, rollNo;

// Input for Person object

cout << "Creating Person object:" << endl;

cout << "Enter name: ";

getline(cin, name);

cout << "Enter age: ";

cin >> age;

cin.ignore();

Person person(name, age);

cout << "\nPerson Details:" << endl;

person.display();

cout << endl;

// Input for Student object

cout << "Creating Student object:" << endl;

cout << "Enter name: ";

getline(cin, name);

cout << "Enter age: ";

cin >> age;

cout << "Enter roll number: ";

cin >> rollNo;

cin.ignore();

Student student(name, age, rollNo);

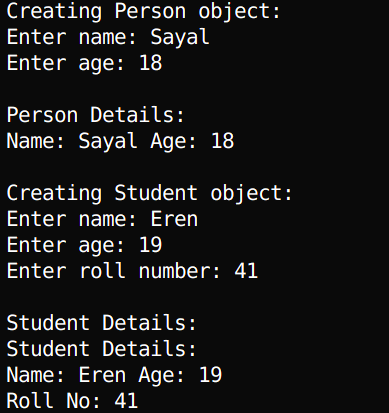
cout << "\nStudent Details:" << endl;

student.display();

return 0;

}

**OUTPUT**



**2. Implement a C++ program with a base class Account having a protected attribute balance. Derive a class SavingsAccount that adds an attribute interestRate and a function addInterest() to modify balance. Use user input to initialize attributes and show how the protected balance is accessed in the derived class but not outside**

#include <iostream>

using namespace std;

class Account {

protected:

double balance;

public:

Account(double b) : balance(b) {}

void display() const {

cout << "Balance: ₨" << balance << endl;

}

};

class SavingsAccount : public Account {

private:

double interestRate;

public:

SavingsAccount(double b, double ir) : Account(b), interestRate(ir) {}

void addInterest() {

double interest = balance \* (interestRate / 100);

balance += interest; // Accessing protected balance

cout << "Interest of ₨" << interest << " added." << endl;

}

void display() const {

cout << "Savings Account Details:" << endl;

Account::display();

cout << "Interest Rate: " << interestRate << "%" << endl;

}

};

int main() {

double balance, interestRate;

cout << "Creating Account object:" << endl;

cout << "Enter balance: ₨";

cin >> balance;

Account account(balance);

cout << "\nAccount Details:" << endl;

account.display();

cout << endl;

cout << "Creating SavingsAccount object:" << endl;

cout << "Enter balance: ₨";

cin >> balance;

cout << "Enter interest rate (%): ";

cin >> interestRate;

SavingsAccount savings(balance, interestRate);

cout << "\nSavings Account Details (Before Interest):" << endl;

savings.display();

cout << endl;

savings.addInterest();

cout << "Savings Account Details (After Interest):" << endl;

savings.display();

cout << endl;

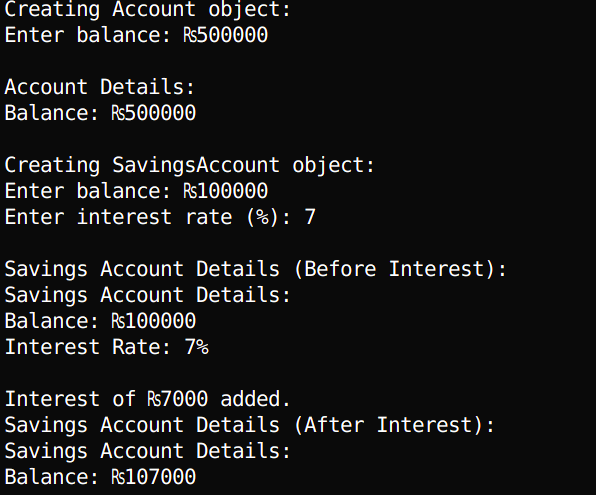
// Compilation error if uncommented: demonstrates protected access

// cout << savings.balance; // Error: balance is protected

return 0;

}

**OUTPUT**



**3. Write a C++ program with a base class Shape having a function draw(). Declare a derived class Circle with an attribute radius initialized via user input. Create a Circle object and call draw() to display a message including radius, demonstrating proper derived class declaration.**

#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw() const {

cout << "Drawing a generic shape." << endl;

}

};

class Circle : public Shape {

private:

double radius;

public:

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

void draw() const override {

cout << "Drawing a Circle with radius: " << radius << endl;

}

};

int main() {

double radius;

cout << "Creating Circle object:" << endl;

cout << "Enter radius: ";

cin >> radius;

Circle circle(radius);

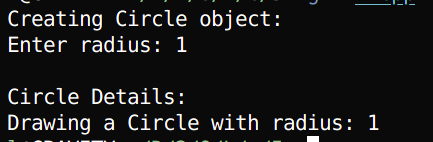
cout << "\nCircle Details:" << endl;

circle.draw();

return 0;

}

**OUTPUT**



**4. Create a C++ program with a base class Vehicle having a function move(). Derive a class Car that overrides move() to indicate driving. Use a base class pointer to call move() on a Car object initialized with user input for attributes like brand. Show that Car is a Vehicle.**

#include <iostream>

#include <string>

using namespace std;

class Vehicle {

protected:

string brand;

public:

Vehicle(string b) : brand(b) {}

virtual void move() const {

cout << brand << " is moving." << endl;

}

void display() const {

cout << "Brand: " << brand << endl;

}

};

class Car : public Vehicle {

public:

Car(string b) : Vehicle(b) {}

void move() const override {

cout << brand << " is driving on the road." << endl;

}

};

int main() {

string brand;

cout << "Creating Car object:" << endl;

cout << "Enter brand: ";

getline(cin, brand);

Car car(brand);

cout << "\nCar Details:" << endl;

car.display();

cout << endl;

cout << "Using base class pointer to demonstrate Car is a Vehicle:" << endl;

Vehicle\* vehiclePtr = &car;

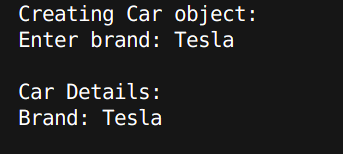
vehiclePtr->m

ove();

return 0;

}

**OUTPUT**



**5. Implement a C++ program with a class Engine having an attribute horsepower. Create a class Car that contains an Engine object (composition) and an attribute model. Initialize all attributes with user input and display details to show that Car has an Engine**

#include <iostream>

#include <string>

using namespace std;

class Engine {

private:

int horsepower;

public:

Engine(int hp) : horsepower(hp) {}

void display() const {

cout << "Engine Horsepower: " << horsepower << endl;

}

};

class Car {

private:

Engine engine; // Composition: Car has an Engine

string model;

public:

Car(int hp, string m) : engine(hp), model(m) {}

void display() const {

cout << "Car Details:" << endl;

cout << "Model: " << model << endl;

engine.display();

}

};

int main() {

string model;

int horsepower;

cout << "Creating Car object:" << endl;

cout << "Enter car model: ";

getline(cin, model);

cout << "Enter engine horsepower: ";

cin >> horsepower;

cin.ignore(); // Clear input buffer

Car car(horsepower, model);

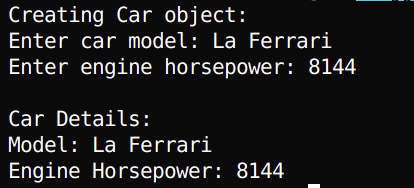
cout << "\n";

car.display();

return 0;

}

**OUTPUT**



**6. Write a C++ program with a base class Base having public, protected, and private attributes (e.g., pubVar, protVar, privVar). Derive three classes using public, protected, and private inheritance, respectively. Demonstrate with user-initialized objects how each inheritance type affects access to base** class members.

#include <iostream>

#include <string>

using namespace std;

class Base {

public:

int pubVar;

protected:

int protVar;

private:

int privVar;

public:

Base(int pub, int prot, int priv) : pubVar(pub), protVar(prot), privVar(priv) {}

void display() const {

cout << "Base: Public Var = " << pubVar << ", Protected Var = " << protVar << ", Private Var = " <<

privVar << endl;

}

};

class PublicDerived : public Base {

public:

PublicDerived(int pub, int prot, int priv) : Base(pub, prot, priv) {}

void display() const {

cout << "PublicDerived: Public Var = " << pubVar << ", Protected Var = " << protVar << endl;

// privVar is inaccessible here

}

};

class ProtectedDerived : protected Base {

public:

ProtectedDerived(int pub, int prot, int priv) : Base(pub, prot, priv) {}

void display() const {

cout << "ProtectedDerived: Public Var = " << pubVar << ", Protected Var = " << protVar << endl;

// privVar is inaccessible here

}

};

class PrivateDerived : private Base {

public:

PrivateDerived(int pub, int prot, int priv) : Base(pub, prot, priv) {}

void display() const {

cout << "PrivateDerived: Public Var = " << pubVar << ", Protected Var = " << protVar << endl;

// privVar is inaccessible here

}

};

int main() {

int pub, prot, priv;

// Input for Base object

cout << "Creating Base object:" << endl;

cout << "Enter public variable: ";

cin >> pub;

cout << "Enter protected variable: ";

cin >> prot;

cout << "Enter private variable: ";

cin >> priv;

Base base(pub, prot, priv);

cout << "\nBase Object Details:" << endl;

base.display();

cout << "Accessing pubVar directly: " << base.pubVar << endl;

// base.protVar and base.privVar are inaccessible here

cout << endl;

// Input for PublicDerived object

cout << "Creating PublicDerived object:" << endl;

cout << "Enter public variable: ";

cin >> pub;

cout << "Enter protected variable: ";

cin >> prot;

cout << "Enter private variable: ";

cin >> priv;

PublicDerived pubDerived(pub, prot, priv);

cout << "\nPublicDerived Object Details:" << endl;

pubDerived.display();

cout << "Accessing pubVar directly: " << pubDerived.pubVar << endl;

// pubDerived.protVar and pubDerived.privVar are inaccessible

cout << endl;

// Input for ProtectedDerived object

cout << "Creating ProtectedDerived object:" << endl;

cout << "Enter public variable: ";

cin >> pub;

cout << "Enter protected variable: ";

cin >> prot;

cout << "Enter private variable: ";

cin >> priv;

ProtectedDerived protDerived(pub, prot, priv);

cout << "\nProtectedDerived Object Details:" << endl;

protDerived.display();

// protDerived.pubVar and protDerived.protVar are inaccessible

cout << endl;

// Input for PrivateDerived object

cout << "Creating PrivateDerived object:" << endl;

cout << "Enter public variable: ";

cin >> pub;

cout << "Enter protected variable: ";

cin >> prot;

cout << "Enter private variable: ";

cin >> priv;

PrivateDerived privDerived(pub, prot, priv);

cout << "\nPrivateDerived Object Details:" << endl;

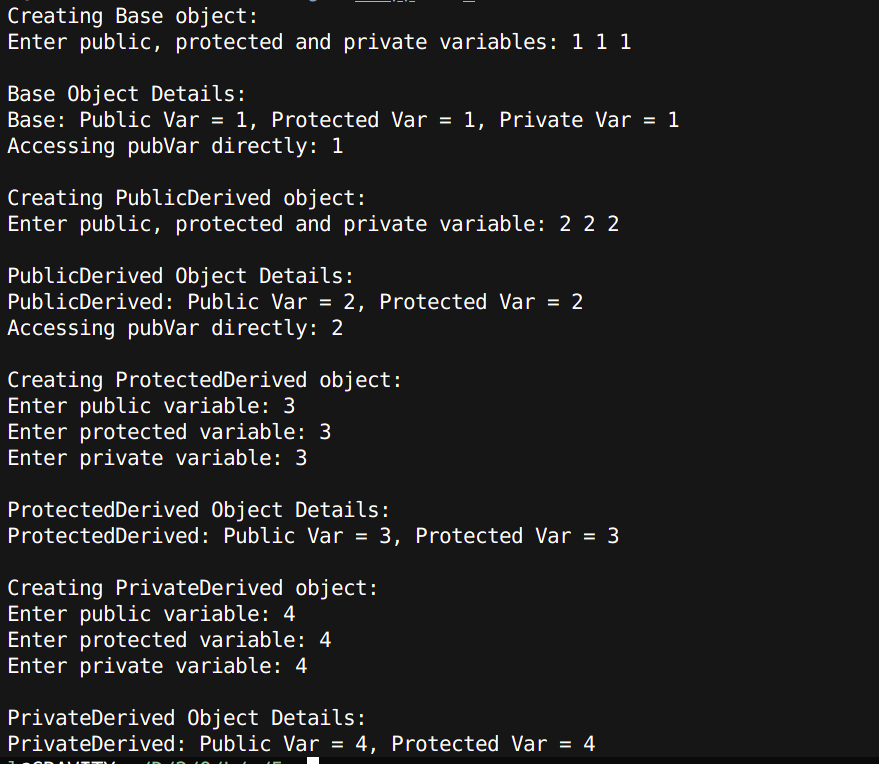
privDerived.display();

// privDerived.pubVar and privDerived.protVar are inaccessible

return 0;

}

**OUTPUT**



**7. Create a C++ program with a base class Animal having a virtual function sound(). Derive classes Dog and Cat that override sound() to print specific sounds. Use a base class pointer array to call sound() on Dog and Cat objects created with user input, showing runtime polymorphism.**

#include <iostream>

#include <string>

using namespace std;

class Animal {

protected:

string name;

public:

Animal(string n) : name(n) {}

virtual void sound() const {

cout << name << " makes a generic animal sound." << endl;

}

void display() const {

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

}

};

class Dog : public Animal {

public:

Dog(string n) : Animal(n) {}

void sound() const override {

cout << name << " says: Woof!" << endl;

}

};

class Cat : public Animal {

public:

Cat(string n) : Animal(n) {}

void sound() const override {

cout << name << " says: Meow!" << endl;

}

};

int main() {

string name;

// Create array of base class pointers

Animal\* animals[2];

// Input for Dog object

cout << "Creating Dog object:" << endl;

cout << "Enter dog name: ";

getline(cin, name);

animals[0] = new Dog(name);

// Input for Cat object

cout << "\nCreating Cat object:" << endl;

cout << "Enter cat name: ";

getline(cin, name);

animals[1] = new Cat(name);

// Display details and demonstrate polymorphism

cout << "\nAnimal Details and Sounds:" << endl;

for (int i = 0; i < 2; i++) {

animals[i]->display();

animals[i]->sound();

cout << endl;

}

// Clean up dynamically allocated memory

for (int i = 0; i < 2; i++) {

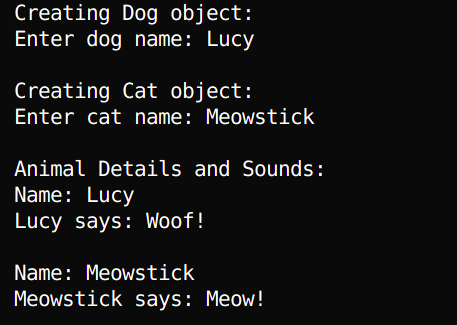
delete animals[i];

}

return 0;

}

**OUTPUT**



**8. Write a C++ program with two base classes Battery and Screen, each with a function showStatus(). Derive a class Smartphone that inherits from both. Resolve ambiguity when calling showStatus() using the scope resolution operator. Initialize attributes with user input and display details.**

**#include <iostream>**

#include <string>

using namespace std;

class Battery {

protected:

int capacity;

public:

Battery(int cap) : capacity(cap) {}

void showStatus() const {

cout << "Battery Status: " << capacity << " mAh" << endl;

}

};

class Screen {

protected:

double size;

public:

Screen(double s) : size(s) {}

void showStatus() const {

cout << "Screen Status: " << size << " inches" << endl;

}

};

class Smartphone : public Battery, public Screen {

private:

string model;

public:

Smartphone(int cap, double s, string m) : Battery(cap), Screen(s), model(m) {}

void display() const {

cout << "Smartphone Details:" << endl;

cout << "Model: " << model << endl;

Battery::showStatus(); // Resolve ambiguity

Screen::showStatus(); // Resolve ambiguity

}

};

int main() {

string model;

int capacity;

double size;

cout << "Creating Smartphone object:" << endl;

cout << "Enter model: ";

getline(cin, model);

cout << "Enter battery capacity (mAh): ";

cin >> capacity;

cout << "Enter screen size (inches): ";

cin >> size;

cin.ignore(); // Clear input buffer

Smartphone phone(capacity, size, model);

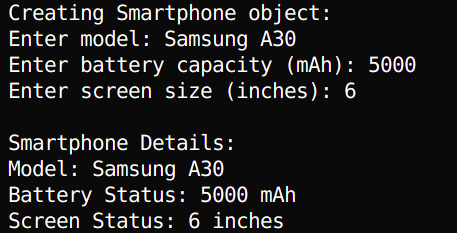
cout << "\n";

phone.display();

return 0;

}

**OUTPUT**



**9. Implement a C++ program with a base class Person having a parameterized constructor for name and age. Derive a class Employee with an additional attribute employeeID. Use user input to initialize all attributes and show the order of constructor invocation when creating an Employee object.**

#include <iostream>

#include <string>

using namespace std;

class Person {

protected:

string name;

int age;

public:

Person(string n, int a) : name(n), age(a) {

cout << "Person constructor called: Name = " << name << ", Age = " << age << endl;

}

void display() const {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

class Employee : public Person {

private:

string employeeID;

public:

Employee(string n, int a, string id) : Person(n, a), employeeID(id) {

cout << "Employee constructor called: EmployeeID = " << employeeID << endl;

}

void display() const {

cout << "Employee Details:" << endl;

Person::display();

cout << "Employee ID: " << employeeID << endl;

}

};

int main() {

string name, employeeID;

int age;

cout << "Creating Employee object:" << endl;

cout << "Enter name: ";

getline(cin, name);

cout << "Enter age: ";

cin >> age;

cin.ignore(); // Clear input buffer

cout << "Enter employee ID: ";

getline(cin, employeeID);

cout << "\nConstructor Invocation Order:" << endl;

Employee employee(name, age, employeeID);

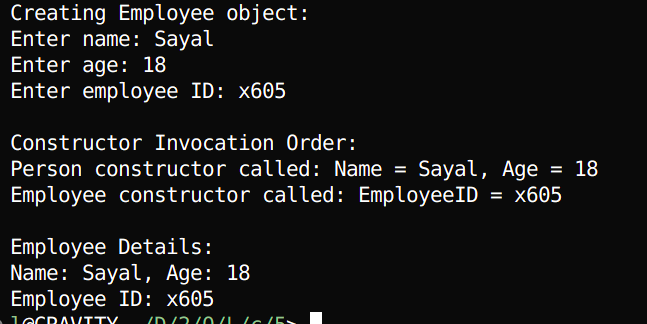
cout << "\n";

employee.display();

return 0;

}

**OUTPUT**



**10. Write a C++ program with a base class Shape and a derived class Rectangle, both with destructors that print messages. Make the base class destructor virtual. Create a Rectangle object through a base class pointer using user input for attributes, and delete it to show proper destructor invocation. Compare with a non-virtual destructor case.**

#include <iostream>

using namespace std;

class Shape {

protected:

double width;

public:

Shape(double w) : width(w) {

cout << "Shape constructor called: Width = " << width << endl;

}

virtual ~Shape() {

cout << "Shape destructor called" << endl;

}

void display() const {

cout << "Shape Width: " << width << endl;

}

};

class Rectangle : public Shape {

private:

double height;

public:

Rectangle(double w, double h) : Shape(w), height(h) {

cout << "Rectangle constructor called: Height = " << height << endl;

}

~Rectangle() {

cout << "Rectangle destructor called" << endl;

}

void display() const {

cout << "Rectangle Details:" << endl;

Shape::display();

cout << "Height: " << height << endl;

}

};

/\*

// Non-virtual destructor case for comparison (uncomment to test)

// Replace the virtual destructor in Shape with this:

class Shape {

protected:

double width;

public:

Shape(double w) : width(w) {

cout << "Shape constructor called: Width = " << width << endl;

}

~Shape() { // Non-virtual destructor

cout << "Shape destructor called" << endl;

}

void display() const {

cout << "Shape Width: " << width << endl;

}

};

\*/

int main() {

double width, height;

cout << "Creating Rectangle object via base class pointer:" << endl;

cout << "Enter width: ";

cin >> width;

cout << "Enter height: ";

cin >> height;

Shape\* shapePtr = new Rectangle(width, height);

cout << "\n";

shapePtr->display();

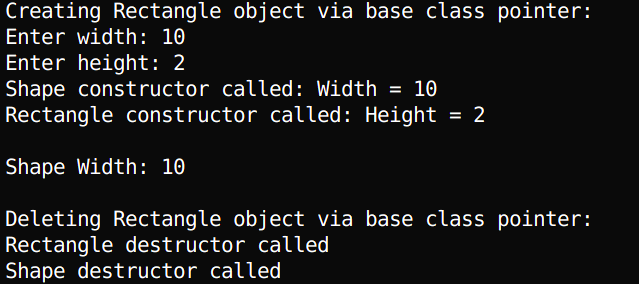
cout << "\nDeleting Rectangle object via base class pointer:" << endl;

delete shapePtr;

return 0;

}

**OUTPUT**



**11. Create a C++ program with a base class A having an attribute value. Derive classes B and C from A, and derive class D from both B and C. Use virtual inheritance to avoid duplication of A’s members. Initialize value with user input and display it from D to show ambiguity resolution.**

#include <iostream>

using namespace std;

class A {

protected:

int value;

public:

void setValue(int v) {

value = v;

}

void showValue() {

cout << "Value from class A: " << value << endl;

}

};

class B : virtual public A {

// Virtually inherits A

};

class C : virtual public A {

// Virtually inherits A

};

class D : public B, public C {

public:

void display() {

showValue(); // No ambiguity due to virtual inheritance

}

};

int main() {

D obj;

int input;

cout << "Enter a value: ";

cin >> input;

obj.setValue(input); // Only one copy of A's data is present

obj.display(); // Unambiguous access to A's method

return 0;

}

**OUTPUT**



# DISCUSSION

This lab titled "Inheritance" explored various forms and applications of inheritance in C++. We examined single, multilevel, multiple, and virtual inheritance, along with the concepts of constructor chaining, access specifiers, and function overriding. In single and multilevel inheritance, we observed how derived classes extend base class functionality, while in multiple inheritance, ambiguity was resolved using the scope resolution operator. We studied public, protected, and private inheritance to understand how member access is affected in derived classes. The lab also emphasized runtime polymorphism using base class pointers and virtual functions, demonstrating how dynamic binding allows for flexible and extensible code. Virtual destructors were introduced to ensure proper destruction of derived objects through base class pointers. Finally, virtual inheritance was used to solve the diamond problem, ensuring only one instance of the base class existed in the inheritance hierarchy.

# CONCLUSION

The lab effectively demonstrated the power and flexibility of inheritance in C++, a key pillar of object-oriented programming. Through different types of inheritance and practical examples, we understood how code reusability, modularity, and polymorphism are achieved. The lab not only clarified how inheritance works at a technical level but also showcased best practices like using virtual functions and destructors to manage complex class hierarchies. Overall, this lab provided a solid foundation for implementing inheritance in real-world C++ applications.