# 1) Q: Create a class BankAccount with private attributes accountNumber and balance. Provide public methods deposit() and withdraw() to modify the balance while ensuring encapsulation

#include <iostream>using namespace std; class BankAccount {

private:

int accountNumber; double balance;

public:

BankAccount(int accNum, double initialBalance) { accountNumber = accNum;

balance = (initialBalance >= 0) ? initialBalance : 0; } void deposit(double amount) {

if (amount > 0) {

balance += amount;

cout << "Deposited: $" << amount << endl;

} else {

cout << "Invalid deposit amount!" << endl; } } void withdraw(double amount) {

if (amount > 0 && amount <= balance) { balance -= amount;

cout << "Withdrawn: $" << amount << endl;

} else {

cout << "Invalid withdrawal amount or insufficient funds!" << endl;

} }

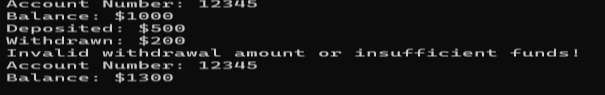
void displayBalance() const {

cout << "Account Number: " << accountNumber << "\nBalance: $" << balance

<< endl; }};int main() {

BankAccount acc(12345, 1000.0); acc.displayBalance(); acc.deposit(500);

acc.withdraw(200); acc.withdraw(1500); acc.displayBalance(); return 0;}



### Data Hiding with Getters & Setters

#include <iostream>using namespace std; class Student {

private:

string name;

int marks;public:

void setName(string studentName) { name = studentName;

}void setMarks(int studentMarks) {

if (studentMarks >= 0 && studentMarks <= 100) { marks = studentMarks; } else {

cout << "Invalid marks! Please enter a value between 0 and 100.\n"}} string getName() {

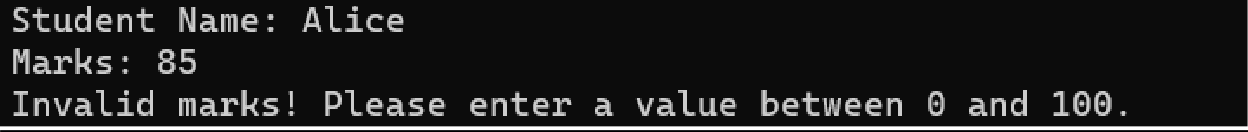
return name; } int getMarks() { return marks;

} void displayStudent() {

cout << "Student Name: " << name << "\nMarks: " << marks << endl;

}};int main() ; Student s1; s1.setName("Alice");s1.setMarks(85); s1.displayStudent(); s1.setMarks(120); return 0;}

***O/P:-***



# Function Overloading

#include <iostream> using namespace std; class MathOperations {

public:

int multiply(int a, int b) { return a \* b;

} int multiply(int a, int b, int c) { return a \* b \* c;

} float multiply(float a, int b) { return a \* b;

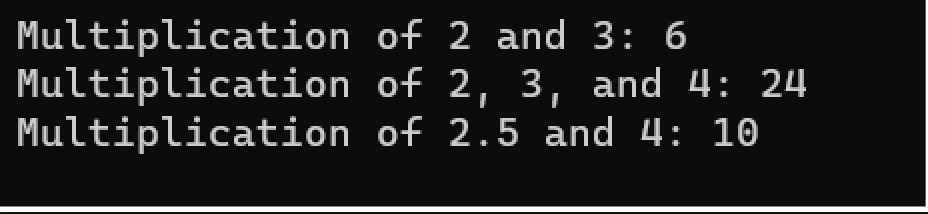
}};int main() { MathOperations mathOp;

cout << "Multiplication of 2 and 3: " << mathOp.multiply(2, 3) << endl;

cout << "Multiplication of 2, 3, and 4: " << mathOp.multiply(2, 3, 4) << endl;

cout << "Multiplication of 2.5 and 4: " << mathOp.multiply(2.5f, 4) << endl; return 0;}

## O/P:-

******

1. ***Operator Overloading (Unary Operator)***

#include <iostream> using namespace std; class MathOperations { public:

int multiply(int a, int b) { return a \* b;

} int multiply(int a, int b, int c) { return a \* b \* c;

} float multiply(float a, int b) {

return a \* b;

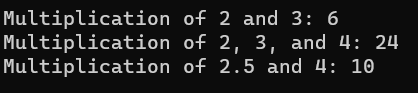
}};int main() { MathOperations mathOp;

cout << "Multiplication of 2 and 3: " << mathOp.multiply(2, 3) << endl;

cout << "Multiplication of 2, 3, and 4: " << mathOp.multiply(2, 3, 4) << endl;

cout << "Multiplication of 2.5 and 4: " << mathOp.multiply(2.5f, 4) << endl; return 0;}

## O/P:-

******

### Operator Overloading (Binary Operator)

#include <iostream> using namespace std; class Animal {

public:

void makeSound() {

cout << "Animal sound" << endl;

}};

class Dog : public Animal { public: void makeSound() {

cout << "Bark" << endl;

}};

int main() { Dog dog; dog.makeSound(); return 0;}

## O/P:-

******

### 7. Single Inheritance

#include <iostream>

using namespace std;class Person {protected: string name;

int age; public:

Person(string n, int a) : name(n), age(a) {}}; class Employee : public Person {

protected:

double salary; public:

Employee(string n, int a, double s) : Person(n, a), salary(s) {}}; class Manager : public Employee {

private:

string department; public:

Manager(string n, int a, double s, string d) : Employee(n, a, s), department(d) {} void display() {

cout << "Name: " << name << ", Age: " << age << ", Salary: " << salary << ", Department: " << department << endl;

}};

int main() {Manager mgr("Alice", 30, 75000, "HR");

mgr.display(); // Output: Name: Alice, Age: 30, Salary: 75000, Department: HR return 0;}

## O/P:-



### 9. Multiple Inheritance

#include <iostream> using namespace std; class Vehicle {

public:

void displayType() {

cout << "Vehicle" << endl }}; class Car : public Vehicle {

public:

void displayType() {

cout << "Car" << endl;

}};

class Bike : public Vehicle { public:

void displayType() {

cout << "Bike" << endl;

}};

int main() {

Car car;

Bike bike;

car.displayType(); bike.displayType(); return 0;}

## O/P:-

******

### Hybrid (Virtual) Inheritance

#include <iostream> using namespace std; class Person {public:

void display() { cout << "Person" << endl;}}; class Teacher : virtual public Person {public:

void display() { cout << "Teacher" << endl;}}; class Student : virtual public Person {public:

void display() { cout << "Student" << endl;}};

class TeachingAssistant : public Teacher, public Student { public: void display() {Teacher::display();Student::display(); }}; int main() {TeachingAssistant ta;

ta.display(); return 0;}



### Constructor and Destructor Call Order

#include <iostream> using namespace std; class Parent {

public:

Parent() {

cout << "Parent Constructor" << endl;}

~Parent() {

cout << "Parent Destructor" << endl; }}; class Child : public Parent {

public:

Child() {

cout << "Child Constructor" << endl; }

~Child() {

cout << "Child Destructor" << endl; }}; int main() {

Child child; return 0;

}#include <iostream>

using namespace std;class Parent { public:

Parent() { cout << "Parent Constructor" << endl;}

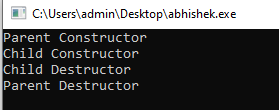
~Parent() {

cout << "Parent Destructor" << endl; }}; class Child : public Parent {

public:Child() {

cout << "Child Constructor" << endl; }

~Child() {cout << "Child Destructor" << endl; }}; int main() {Child child; return 0; }



### Abstract Class and Pure Virtual Functions

#include <iostream> #include <cmath> using namespace std; class Shape {

public:

virtual double area() = 0; };

class Circle : public Shape { private:

double radius; public:

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

return M\_PI \* radius \* radius;}}; class Rectangle : public Shape {

private:

double width; double height;

public:

Rectangle(double w, double h) : width(w), height(h) {} double area() override {

return width \* height; }}; int main() {

Shape\* circle = new Circle(5.0);

Shape\* rectangle = new Rectangle(4.0, 6.0); cout << "Area of Circle: " << circle->area() << endl;

cout << "Area of Rectangle: " << rectangle->area() << endl; delete circle;

delete rectangle; return 0;

}



### Method Overriding (Runtime Polymorphism)

#include <iostream>

using namespace std; class Employee { public:

virtual double calculateSalary() { return 50000; }};

class Manager : public Employee { public:

double calculateSalary() override { return 70000; }};

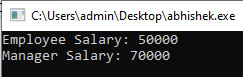
int main() {

Employee\* emp = new Employee(); Employee\* mgr = new Manager();

cout << "Employee Salary: " << emp->calculateSalary() << endl; cout << "Manager Salary: " << mgr->calculateSalary() << endl; delete emp;

delete mgr;return 0;

}



### Virtual Functions and Dynamic Binding

#include <iostream> using namespace std; class Animal {

public:

virtual void speak() {

cout << "Animal speaks" << endl;

}};

class Dog : public Animal { public:

void speak() override {

cout << "Dog barks" << endl; }}; class Cat : public Animal {

public:

void speak() override {

cout << "Cat meows" << endl;

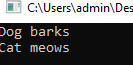
}};

int main() {

Animal\* animals[2]; animals[0] = new Dog(); animals[1] = new Cat();

for (int i = 0; i < 2; i++) { animals[i]->speak(); }

for (int i = 0; i < 2; i++) { delete animals[i]; }return 0;}



### Friend Function

#include <iostream> using namespace std; class Rectangle { private:

double length; double breadth; public:

Rectangle(double l, double b) : length(l), breadth(b) {} friend double calculateArea(Rectangle& rect); };

double calculateArea(Rectangle& rect) {

return rect.length \* rect.breadth; } int main() {

Rectangle rect(5.0, 3.0);

cout << "Area of Rectangle: " << calculateArea(rect) << endl; return 0;}



### Static Members and Methods

#include <iostream> using namespace std;

class Counter {private:static int count;public: Counter() { count++; }

static void displayCount() {

cout << "Number of objects created: " << count << endl; }}; int Counter::count = 0;

int main() {

Counter c1;Counter c2; Counter::displayCount(); return 0;}



### Copy Constructor

#include <iostream> #include <cstring> using namespace std; class Book {

private:

char\* title; public:

Book(const char\* t) {

title = new char[strlen(t) + 1]; strcpy(title, t); }

Book(const Book& b) {

title = new char[strlen(b.title) + 1]; strcpy(title, b.title);}

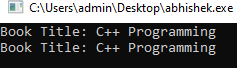
~Book() {

delete[] title; } void display() const {

cout << "Book Title: " << title << endl; }}; int main() {

Book book1("C++ Programming"); Book book2 = book1; book1.display();

book2.display(); return 0;}



### Exception Handling in OOP

#include <iostream> using namespace std; class Division { public:

double divide(int numerator, int denominator) { if (denominator == 0) {

throw runtime\_error("Division by zero is not allowed."); } return static\_cast<double>(numerator) / denominator; }};

int main() {

Division div;

int num, denom;

cout << "Enter numerator: "; cin >> num;

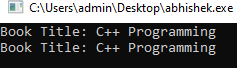
cout << "Enter denominator: "; cin >> denom;

try {

double result = div.divide(num, denom); cout << "Result: " << result << endl;

} catch (const runtime\_error& e) {

cout << "Error: " << e.what() << endl; } return 0;}



### Smart Pointer Implementation (C++)

#include <iostream> using namespace std; template <typename T> class UniquePointer { private:

T\* ptr; public:

UniquePointer(T\* p = nullptr) : ptr(p) {}

~UniquePointer() {

delete ptr; } T& operator\*() {

return \*ptr; } T\* operator->() {

return ptr; }

UniquePointer(const UniquePointer&) = delete; UniquePointer& operator=(const UniquePointer&) = delete;

UniquePointer(UniquePointer&& other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }

UniquePointer& operator=(UniquePointer&& other) noexcept { if (this != &other) {

delete ptr;

ptr = other.ptr;

other.ptr = nullptr; } return \*this; }};

class Test { public:

void display() {

cout << "Test class method called." << endl; }}; int main() {

UniquePointer<Test> uptr(new Test()); uptr->display();

return 0;

}

