Q-1: Implement a base class "Shape" with two derived classes "Circle" and "Rectangle."

Calculate and display the area of a circle and rectangle using single inheritance.

Sample test case:

|  |
| --- |
| Input: Circle circle(5.0)  Output:Area of Circle: 78.5  Input: Rectangle rectangle(4.0, 6.0)  Output: Area of Rectangle: 24 |

Solution:

#include <iostream>

// Abstract Base class Shape

class Shape {

public:

virtual double calculateArea() const = 0;

};

// Derived class Circle

class Circle : public Shape {

private:

double radius;

public:

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

double calculateArea() const override {

return 3.14159 \* radius \* radius;

}

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double calculateArea() const override {

return length \* width;

}

};

int main() {

Shape\* shapes[2];

shapes[0] = new Circle(5.0);

shapes[1] = new Rectangle(4.0, 6.0);

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

std::cout << "Area: " << shapes[i]->calculateArea() << std::endl;

delete shapes[i];

}

return 0;

}

Q-2: Create a base class "Vehicle" with a derived class "Car" and another derived class "SportsCar."Implement the "Vehicle" class to store the vehicle's name and display it.

The "Car" class should store the car's model and display it.The "SportsCar" class should store the top speed of the sports car and display it.

Sample test case:

|  |
| --- |
| Input:  SportsCar myCar("Ferrari", "F430", 320);  myCar.display();  Output:  Vehicle Name: Ferrari  Car Model: F430  Top Speed: 320 km/h |

Solution:

#include <iostream>

#include <string>

// Base class Vehicle

class Vehicle {

protected:

std::string name;

public:

Vehicle(const std::string& n) : name(n) {}

void display() const {

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

}

};

// Derived class Car

class Car : public Vehicle {

protected:

std::string model;

public:

Car(const std::string& n, const std::string& m) : Vehicle(n), model(m) {}

void display() const {

Vehicle::display();

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

}

};

// Derived class SportsCar

class SportsCar : public Car {

private:

int topSpeed;

public:

SportsCar(const std::string& n, const std::string& m, int speed)

: Car(n, m), topSpeed(speed) {}

void display() const {

Car::display();

std::cout << "Top Speed: " << topSpeed << " km/h" << std::endl;

}

};

int main() {

SportsCar myCar("Ferrari", "F430", 320);

myCar.display();

return 0;

}

Q-3: Implement a base class "Person" with two derived classes "Employee" and "Student."

Create another class "Faculty" that is derived from both "Employee" and "Student."

Display the details of a faculty member, including their name, employee ID, and student ID.

Sample test case:

|  |
| --- |
| Input:  Faculty faculty("John Doe", 12345, 98765)  Output:  Name: John Doe  Employee ID: 12345  Name: John Doe  Student ID: 98765 |
|  |

Solution:

#include <iostream>

#include <string>

// Base class Person

class Person {

protected:

std::string name;

public:

Person(const std::string& n) : name(n) {}

void display() const {

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

}

};

// Derived class Employee

class Employee : public Person {

protected:

int employeeID;

public:

Employee(const std::string& n, int id) : Person(n), employeeID(id) {}

void display() const {

Person::display();

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

}

};

// Derived class Student

class Student : public Person {

protected:

int studentID;

public:

Student(const std::string& n, int id) : Person(n), studentID(id) {}

void display() const {

Person::display();

std::cout << "Student ID: " << studentID << std::endl;

}

};

// Derived class Faculty

class Faculty : public Employee, public Student {

public:

Faculty(const std::string& n, int empID, int stuID)

: Employee(n, empID), Student(n, stuID) {}

void display() const {

Employee::display();

Student::display();

}

};

int main() {

Faculty faculty("John Doe", 12345, 98765);

faculty.display();

return 0;

}

Q-4: Implement a base class "Animal" with two derived classes "Mammal" and "Bird."

Add a function "sound()" to the "Animal" class that displays "Unknown sound." Override the "sound()" function in both "Mammal" and "Bird" to display their respective sounds.

Sample test case:

|  |
| --- |
| Input:  Animal.sound()  Mammal.sound()  Bird.sound()  Output:  Unknown sound  Mammal sound: Roar  Bird sound: Chirp |

Output:

#include <iostream>

#include <string>

// Base class Animal

class Animal {

public:

virtual void sound() const {

std::cout << "Unknown sound" << std::endl;

}

};

// Derived class Mammal

class Mammal : public Animal {

public:

void sound() const override {

std::cout << "Mammal sound: Roar" << std::endl;

}

};

// Derived class Bird

class Bird : public Animal {

public:

void sound() const override {

std::cout << "Bird sound: Chirp" << std::endl;

}

};

int main() {

Animal\* animals[3];

animals[0] = new Animal;

animals[1] = new Mammal;

animals[2] = new Bird;

for (int i = 0; i < 3; ++i) {

animals[i]->sound();

delete animals[i];

}

return 0;

}

Q-5: Create a base class "Vehicle" with two derived classes "Car" and "Bike."

Add a virtual function "start()" in the base class and override it in the derived classes to display their respective starting messages.

Sample test case:

|  |
| --- |
| Input: Car.start()  Output: Car starting...  Input: Bike.start()  Output: Bike starting... |

Solution:

#include <iostream>

// Base class Vehicle

class Vehicle {

public:

virtual void start() const {

std::cout << "Vehicle starting..." << std::endl;

}

};

// Derived class Car

class Car : public Vehicle {

public:

void start() const override {

std::cout << "Car starting..." << std::endl;

}

};

// Derived class Bike

class Bike : public Vehicle {

public:

void start() const override {

std::cout << "Bike starting..." << std::endl;

}

};

int main() {

Vehicle\* v1 = new Car;

Vehicle\* v2 = new Bike;

v1->start();

v2->start();

delete v1;

delete v2;

return 0;

}

Q-6: Implement a base class "Shape" with a virtual function "display()" that displays "Shape". Implement two derived classes "Circle" and "Rectangle" that override the "display()" function to display "Circle" and "Rectangle," respectively. Perform upcasting and downcasting with base class pointers.

Sample test case:

|  |
| --- |
| Shape\* shape1 = new Circle;  shape1->display(); // Output: Circle  Shape\* shape2 = new Rectangle;  shape2->display(); // Output: Rectangle |

Solution:

#include <iostream>

// Base class Shape

class Shape {

public:

virtual void display() const {

std::cout << "Shape" << std::endl;

}

};

// Derived class Circle

class Circle : public Shape {

public:

void display() const override {

std::cout << "Circle" << std::endl;

}

};

// Derived class Rectangle

class Rectangle : public Shape {

public:

void display() const override {

std::cout << "Rectangle" << std::endl;

}

};

int main() {

Shape\* shape1 = new Circle;

Shape\* shape2 = new Rectangle;

shape1->display(); // Output: Circle

shape2->display(); // Output: Rectangle

Circle\* circle = dynamic\_cast<Circle\*>(shape1);

if (circle != nullptr) {

circle->display(); // Output: Circle

}

Rectangle\* rectangle = dynamic\_cast<Rectangle\*>(shape2);

if (rectangle != nullptr) {

rectangle->display(); // Output: Rectangle

}

delete shape1;

delete shape2;

return 0;

}

Q-7: Implement a base class Rectangle with attributes length and width and a method to calculate the area. Create a derived class Box from Rectangle with an additional height attribute and a method to calculate the volume of the box.

Sample test case:

|  |
| --- |
| Input:  Enter the length of the box: 5  Enter the width of the box: 3  Enter the height of the box: 2  Output:  Box's Volume: 30 |

Solution:

#include <iostream>

class Rectangle {

protected:

double length, width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double calculateArea() {

return length \* width;

}

};

class Box : public Rectangle {

private:

double height;

public:

Box(double l, double w, double h) : Rectangle(l, w), height(h) {}

double calculateVolume() {

return calculateArea() \* height;

}

};

int main() {

double length, width, height;

std::cout << "Enter the length of the box: ";

std::cin >> length;

std::cout << "Enter the width of the box: ";

std::cin >> width;

std::cout << "Enter the height of the box: ";

std::cin >> height;

Box box(length, width, height);

std::cout << "Box's Volume: " << box.calculateVolume() << std::endl;

return 0;

}

Q-8: Create a base class Shape with a method to calculate the area. Derive three classes Circle, Rectangle, and Triangle from Shape, and implement methods to calculate their respective areas.

Sample test case:

|  |
| --- |
| circle.calculateArea() // Circle's Area: 78.5  rectangle.calculateArea() // Rectangle's Area: 24  triangle.calculateArea() // Triangle's Area: 12 |

Solution:

#include <iostream>

#include <cmath>

#define M\_PI 3.14

class Shape {

public:

virtual double calculateArea() = 0; // Pure virtual method, making Shape an abstract class

};

class Circle : public Shape {

private:

double radius;

public:

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

double calculateArea() override {

return M\_PI \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double length, width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double calculateArea() override {

return length \* width;

}

};

class Triangle : public Shape {

private:

double base, height;

public:

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

double calculateArea() override {

return 0.5 \* base \* height;

}

};

int main() {

Circle circle(5.0);

std::cout << "Circle's Area: " << circle.calculateArea() << std::endl;

Rectangle rectangle(4.0, 6.0);

std::cout << "Rectangle's Area: " << rectangle.calculateArea() << std::endl;

Triangle triangle(3.0, 8.0);

std::cout << "Triangle's Area: " << triangle.calculateArea() << std::endl;

return 0;

}

Q-9: Create a base class Calculator with methods for addition and subtraction.

Derive a class ScientificCalculator from Calculator and add methods for multiplication and division.

Sample test case:

|  |
| --- |
| Input:  ScientificCalculator sc;  a = 10, b = 3  sc.add(a, b)  sc.subtract(a, b)  sc.multiply(a, b)  sc.divide(a, b)  Output:  Addition: 13  Subtraction: 7  Multiplication: 30  Division: 3 |

Solution:

#include <iostream>

class Calculator {

public:

int add(int a, int b) {

return a + b;

}

int subtract(int a, int b) {

return a - b;

}

};

class ScientificCalculator : public Calculator {

public:

int multiply(int a, int b) {

return a \* b;

}

int divide(int a, int b) {

if (b == 0) {

std::cout << "Error: Division by zero!" << std::endl;

return 0;

}

return a / b;

}

};

int main() {

ScientificCalculator sc;

int a = 10, b = 3;

std::cout << "Addition: " << sc.add(a, b) << std::endl;

std::cout << "Subtraction: " << sc.subtract(a, b) << std::endl;

std::cout << "Multiplication: " << sc.multiply(a, b) << std::endl;

std::cout << "Division: " << sc.divide(a, b) << std::endl;

return 0;

}

Q-10: Create a base class NumberSeries with a method to generate the next term in the series. Derive two classes FibonacciSeries and PrimeSeries from NumberSeries to generate the next Fibonacci number and the next prime number, respectively.Sample test case:

|  |
| --- |
| Input: n=10  Output: Fibonacci Series: 1 2 3 5 8 13 21 34 55 89  Prime Series: 3 4 5 7 11 13 17 19 23 29 |

Solution:

#include <iostream> // Include the standard input/output stream library

#include <vector> // Include the vector container

// Base class: NumberSeries

class NumberSeries {

public:

// Virtual function to generate the next term in the series (to be overridden by derived classes)

virtual int generateNextTerm() = 0;

};

// Derived class: FibonacciSeries (inherits from NumberSeries)

class FibonacciSeries : public NumberSeries {

private:

int prev, curr; // Private data members to keep track of previous and current terms

public:

// Constructor to initialize the first two terms of the Fibonacci series

FibonacciSeries() : prev(0), curr(1) {}

// Overridden function to generate the next term in the Fibonacci series

int generateNextTerm() override {

int next = prev + curr;

prev = curr;

curr = next;

return next;

}

};

// Derived class: PrimeSeries (inherits from NumberSeries)

class PrimeSeries : public NumberSeries {

private:

std::vector<bool> isPrime; // Private vector to store whether a number is prime or not

int num; // Private data member to keep track of the current number

public:

// Constructor to initialize the PrimeSeries with the first prime number (2)

PrimeSeries() : num(2) {

isPrime.resize(1000, true); // Resize the vector to initially consider all numbers as prime

isPrime[0] = isPrime[1] = false; // Mark 0 and 1 as non-prime since they are not considered prime

}

// Overridden function to generate the next prime number in the series

int generateNextTerm() override {

while (true) {

++num;

if (num >= isPrime.size())

isPrime.resize(isPrime.size() \* 2, true); // Resize the vector if needed

if (isPrime[num]) {

// If the current number is prime, mark all its multiples as non-prime

for (int i = num \* 2; i < isPrime.size(); i += num) {

isPrime[i] = false;

}

return num; // Return the current prime number

}

}

}

};

int main() {

// Create a FibonacciSeries object and display the first 10 terms of the Fibonacci series

FibonacciSeries fibSeries;

std::cout << "Fibonacci Series: ";

for (int i = 0; i < 10; ++i) {

std::cout << fibSeries.generateNextTerm() << " ";

}

std::cout << std::endl;

// Create a PrimeSeries object and display the first 10 prime numbers

PrimeSeries primeSeries;

std::cout << "Prime Series: ";

for (int i = 0; i < 10; ++i) {

std::cout << primeSeries.generateNextTerm() << " ";

}

std::cout << std::endl;

return 0;

}

Q-11: In a virtual reality game, the players are exploring a fantasy world filled with magical objects. As part of their adventure, they come across enchanted circles and cylinders with unique properties. To help the players understand these objects, you need to create a program that calculate area of circle and volume of the cylinder. Cylinder should inherit properties of Circle.

Sample test case:

|  |
| --- |
| Input: radius: 5, height: 7  Output: Cylinder's Volume: 549.5 |

Solution:

#include <iostream> // Include the standard input/output stream library

#include <cmath> // Include the math library for mathematical functions

#define M\_PI 3.14 // Define the constant value of pi

// Base class: Circle

class Circle {

protected:

double radius; // Protected data member to store the radius of the circle

public:

// Constructor to initialize the radius of the circle

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

// Function to calculate the area of the circle

double calculateArea() {

return M\_PI \* radius \* radius; // Area formula for a circle

}

};

// Derived class: Cylinder (inherits from Circle)

class Cylinder : public Circle {

private:

double height; // Private data member to store the height of the cylinder

public:

// Constructor to initialize the radius and height of the cylinder using the Circle's constructor

Cylinder(double r, double h) : Circle(r), height(h) {}

// Function to calculate the volume of the cylinder

double calculateVolume() {

return calculateArea() \* height; // Volume formula for a cylinder: area of base \* height

}

};

int main() {

double radius, height;

std::cout << "Enter radius: ";

std::cin >> radius;

std::cout << "Enter height: ";

std::cin >> height;

Cylinder cylinder(radius, height); // Create a Cylinder object with the given radius and height

std::cout << "Cylinder's Volume: " << cylinder.calculateVolume() << std::endl;

return 0;

}

Q-12: Write a C++ program by creating three classes: A, B, and C.

Class A is the base class and contains two data members x and y, and a member function getdata() to input values for x and y.

Class B is derived from class A and contains a member function product() that calculates and displays the product of x and y.

Class C is also derived from class A and contains a member function sum() that calculates and displays the sum of x and y.

In the main() function, create objects of classes B and C, input values for x and y for each object using the getdata() function,

and then calculate and display the product for object B and the sum for object C.

Sample test case:

|  |
| --- |
| Input: Enter value of x and y: 4 5  Output: Product= 20  Input: Enter value of x and y: 2 4  Output: Sum= 6 |

Solution:

#include <iostream>

using namespace std;

class A // Base class

{

public:

int x, y; // data members

void getdata() // to input x and y

{

cout << "Enter value of x and y:\n";

cin >> x >> y;

}

};

class B : public A // B is derived from class A

{

public:

void product()

{

cout << "\nProduct= " << x \* y << endl; // Perform product

}

};

class C : public A // C is also derived from class A

{

public:

void sum()

{

cout << "\nSum= " << x + y; // Perform sum

}

};

int main()

{

B obj1; // object of derived class B

C obj2; // object of derived class C

obj1.getdata(); // input x and y for obj1

obj1.product(); // calculate and display the product for obj1

obj2.getdata(); // input x and y for obj2

obj2.sum(); // calculate and display the sum for obj2

return 0;

}

Q-13: Create a base class "Person" with two derived classes "Student" and "Employee."

Implement functions to display details of students and employees by use of multiple inheritance.

Sample test case:

|  |
| --- |
| Student student("John Doe", 12345)  student.displayDetails() // Name: John Doe, Roll Number: 12345  Employee employee("Jane Smith", "E789")  employee.displayDetails() // Name: Jane Smith, Employee ID: E789 |

Solution:

#include <iostream>

#include <string>

using namespace std;

class Person {

protected:

string name;

public:

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

};

class Student : public Person {

private:

int rollNumber;

public:

Student(string n, int roll) : Person(n), rollNumber(roll) {}

void displayDetails() const {

cout << "Name: " << name << ", Roll Number: " << rollNumber << endl;

}

};

class Employee : public Person {

private:

string empId;

public:

Employee(string n, string id) : Person(n), empId(id) {}

void displayDetails() const {

cout << "Name: " << name << ", Employee ID: " << empId << endl;

}

};

int main() {

Student student("John Doe", 12345);

Employee employee("Jane Smith", "E789");

student.displayDetails();

employee.displayDetails();

return 0;

}

Q-14: Create a base class "Animal" and two derived classes "FlyingAnimal" and "SwimmingAnimal." Implement a class "Bird" that inherits from both "FlyingAnimal" and "Animal" by using virtual base classes.

Sample test case:

|  |
| --- |
| Input: Bird parrot("Duck")  Output: Duck can fly. Duck can swim. |

Solution:

#include <iostream>

#include <string>

using namespace std;

class Animal {

protected:

string species;

public:

Animal(string s) : species(s) {}

};

class FlyingAnimal : virtual public Animal {

public:

FlyingAnimal(string s) : Animal(s) {}

virtual void fly() const = 0; // Pure virtual function

};

class SwimmingAnimal : virtual public Animal {

public:

SwimmingAnimal(string s) : Animal(s) {}

virtual void swim() const = 0; // Pure virtual function

};

class Bird : public FlyingAnimal, public SwimmingAnimal {

public:

Bird(string s) : Animal(s), FlyingAnimal(s), SwimmingAnimal(s) {}

void fly() const override {

cout << species << " can fly." << endl;

}

void swim() const override {

cout << species << " can swim." << endl;

}

};

int main() {

Bird parrot("Duck");

parrot.fly();

parrot.swim();

return 0;

}

Q-15: Implement a base class "Base" and two derived classes "A" and "B." Add a static member variable to "Base" and use it in "A" and "B."

Sample test case:

|  |
| --- |
| Base::staticVar // Output:Static Variable Value: 0  A objA1;  A objA2;  B objB1;  B objB2;  Base::staticVar // Output:Static Variable Value: 4 |

Solution:

#include <iostream>

using namespace std;

class Base {

public:

static int staticVar;

Base() {

staticVar++;

}

};

int Base::staticVar = 0;

class A : public Base {

public:

A() : Base() {}

};

class B : public Base {

public:

B() : Base() {}

};

int main() {

A objA1;

A objA2;

B objB1;

B objB2;

cout << "Static Variable Value: " << Base::staticVar << endl;

return 0;

}

Q-16: Create a program that manages animal information, including species and characteristics such as legs for mammals and flying ability for birds. Create a base class Animal Create two derived classed that inherit from Base as class Mammal and class Bird. Print Species and Legs for Mammals and print Species and Can fly for Birds.

Sample test case:

|  |
| --- |
| Input:  Mammal lion("Lion", 4)  Output:  Species: Lion  Legs: 4  Input:  Bird sparrow("Sparrow", true)  Output:  Species: Sparrow  Can Fly: Yes |

Solution:

#include <iostream>

using namespace std;

class Animal {

protected:

string species;

public:

Animal(string sp) : species(sp) {}

void displaySpecies() {

cout << "Species: " << species << endl;

}

};

class Mammal : public Animal {

protected:

int legs;

public:

Mammal(string sp, int lg) : Animal(sp), legs(lg) {}

void displayLegs() {

cout << "Legs: " << legs << endl;

}

};

class Bird : public Animal {

protected:

bool canFly;

public:

Bird(string sp, bool fly) : Animal(sp), canFly(fly) {}

void displayFly() {

cout << "Can Fly: " << (canFly ? "Yes" : "No") << endl;

}

};

int main() {

Mammal lion("Lion", 4);

Bird sparrow("Sparrow", true);

lion.displaySpecies();

lion.displayLegs();

sparrow.displaySpecies();

sparrow.displayFly();

return 0;

}

Q-17: Create a base class "A" with two derived classes "B" and "C". Implement a class "D" that inherits from both "B" and "C," and call base class constructors during multiple inheritance

Sample test case:

|  |
| --- |
| Input: D obj;  Output:  A's Constructor  B's Constructor  C's Constructor  D's Constructor |

Solution:

#include <iostream>

using namespace std;

class A {

public:

A() {

cout << "A's Constructor" << endl;

}

};

class B : virtual public A {

public:

B() {

cout << "B's Constructor" << endl;

}

};

class C : virtual public A {

public:

C() {

cout << "C's Constructor" << endl;

}

};

class D : public B, public C {

public:

D() {

cout << "D's Constructor" << endl;

}

};

int main() {

D obj;

return 0;

}

Q-18: Create a base class "Device" and two derived classes "Keyboard" and "Mouse."

Implement a class "ComboDevice" that inherits from both "Keyboard" and "Mouse" and perform function overriding with virtual base classes.

Sample test case:

|  |
| --- |
| ComboDevice combo;  combo.type();// Output: Combo Device |

Solution:

#include <iostream>

using namespace std;

class Device {

public:

virtual void type() const {

cout << "Generic Device" << endl;

}

};

class Keyboard : virtual public Device {

public:

void type() const override {

cout << "Keyboard Device" << endl;

}

};

class Mouse : virtual public Device {

public:

void type() const override {

cout << "Mouse Device" << endl;

}

};

class ComboDevice : public Keyboard, public Mouse {

public:

void type() const override {

cout << "Combo Device" << endl;

}

};

int main() {

ComboDevice combo;

combo.type();

return 0;

}

Q-19: Create a program that classifies animals into carnivores, herbivores, and omnivores using hybrid inheritance.

Sample test case:

|  |
| --- |
| Omnivore omnivore("Bear");  omnivore.display(); // Output: Species: Bear |

Solution:

#include <iostream>

using namespace std;

class Animal {

protected:

string species;

public:

Animal(string sp) : species(sp) {}

void display() {

cout << "Species: " << species << endl;

}

};

class Carnivore : virtual public Animal {

public:

Carnivore(string sp) : Animal(sp) {}

};

class Herbivore : virtual public Animal {

public:

Herbivore(string sp) : Animal(sp) {}

};

class Omnivore : public Carnivore, public Herbivore {

public:

Omnivore(string sp) : Animal(sp), Carnivore(sp), Herbivore(sp) {}

};

int main() {

Omnivore omnivore("Bear");

omnivore.display();

return 0;

}

Q-20: Create a base class called Vehicle, and derive two classes Car and Bike from it.

Implement a virtual function startEngine() in the Vehicle class. Override the startEngine() function in both Car and Bike classes to start their respective engines.

Sample test case:

|  |
| --- |
| Vehicle\* vehicle1 = new Car();  vehicle1->startEngine(); // Output: Car engine started.  Vehicle\* vehicle2 = new Bike();  vehicle2->startEngine(); // Output: Bike engine started. |

Solution:

#include <iostream>

class Vehicle {

public:

// Virtual function to start the engine of any vehicle

virtual void startEngine() {

std::cout << "Generic vehicle engine started." << std::endl;

}

};

class Car : public Vehicle {

public:

// Override the startEngine() function for Car

void startEngine() {

std::cout << "Car engine started." << std::endl;

}

};

class Bike : public Vehicle {

public:

// Override the startEngine() function for Bike

void startEngine() {

std::cout << "Bike engine started." << std::endl;

}

};

int main() {

Vehicle\* vehicle1 = new Car();

Vehicle\* vehicle2 = new Bike();

vehicle1->startEngine();

vehicle2->startEngine();

delete vehicle1;

delete vehicle2;

return 0;

}

Q-21: You work as a software engineer for a company specializing in CAD (Computer-Aided Design) software. Your team has been assigned a task to create a C++ program that handles 3D shapes. The program should support two types of 3D shapes: spheres and cones. The main goal is to calculate and display the volumes of these shapes using method overriding.

Sample test case:

|  |
| --- |
| Input:  Sphere(radius=5)  Cone(radius=3, height=7)  Output:  Volume of Sphere: 523.333  Volume of Cone: 65.94 |

Solution:

#include <iostream> // Include the standard input/output stream library

#include <cmath> // Include the math library for mathematical functions

#define M\_PI 3.14 // Define the constant value of pi

// Base class for 3D shapes

class Shape3D {

public:

// Virtual function to calculate the volume of the shape (to be overridden by derived classes)

virtual double calculateVolume() const {

return 0.0; // Default implementation returns 0 for the volume

}

};

// Sphere class, derived from Shape3D

class Sphere : public Shape3D {

private:

double radius; // The radius of the sphere

public:

// Constructor to initialize the sphere with a given radius

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

// Overridden function to calculate the volume of the sphere

double calculateVolume() const override {

return (4.0 / 3.0) \* M\_PI \* pow(radius, 3); // Volume formula for a sphere

}

};

// Cone class, derived from Shape3D

class Cone : public Shape3D {

private:

double radius; // The radius of the cone

double height; // The height of the cone

public:

// Constructor to initialize the cone with given radius and height

Cone(double r, double h) : radius(r), height(h) {}

// Overridden function to calculate the volume of the cone

double calculateVolume() const override {

return (1.0 / 3.0) \* M\_PI \* pow(radius, 2) \* height; // Volume formula for a cone

}

};

int main() {

// Creating instances of Sphere and Cone objects

Shape3D\* shape1 = new Sphere(5);

Shape3D\* shape2 = new Cone(3, 6);

// Calculating and displaying the volume of each shape using their respective functions

std::cout << "Volume of Sphere: " << shape1->calculateVolume() << std::endl;

std::cout << "Volume of Cone: " << shape2->calculateVolume() << std::endl;

// Cleaning up the allocated memory for the objects

delete shape1;

delete shape2;

return 0;

}

Q-22: Create a base class called Person, and derive two classes Student and Teacher from it. Implement a virtual function introduce() in the Person class, which prints "I am a person." Override the introduce() function in both Student and Teacher classes to print their roles and names.

Sample test case:

|  |
| --- |
| Person\* person1 = new Student("Alice");  person1->introduce(); // Output: I am a student named Alice.  Person\* person2 = new Teacher("Mr. Smith");  person2->introduce(); // Output: I am a teacher named Mr. Smith. |

Solution:

#include <iostream>

#include <string>

class Person {

public:

// Virtual function to introduce a person

virtual void introduce() const {

std::cout << "I am a person." << std::endl;

}

};

class Student : public Person {

private:

std::string name;

public:

Student(const std::string& n) : name(n) {}

// Override the introduce() function for Student

void introduce() const override {

std::cout << "I am a student named " << name << "." << std::endl;

}

};

class Teacher : public Person {

private:

std::string name;

public:

Teacher(const std::string& n) : name(n) {}

// Override the introduce() function for Teacher

void introduce() const override {

std::cout << "I am a teacher named " << name << "." << std::endl;

}

};

int main() {

Person\* person1 = new Student("Alice");

Person\* person2 = new Teacher("Mr. Smith");

person1->introduce();

person2->introduce();

delete person1;

delete person2;

return 0;

}

Q-23: Create one Base class and derive Derived class from it. Add default and parameterized constructors in both classes. Pass Arguments to constructor and monitor it.

Sample test case:

|  |
| --- |
| Input: Derived d(25, 15)  Output: Param of Base 25  Param of Derived 15 |

Solution:

#include <iostream>

using namespace std;

class Base

{

public:

Base ()

{

cout << "Default of Base" << endl;

}

Base (int x)

{

cout << "Param of Base " << x << endl;

}

};

class Derived : public Base

{

public:

Derived ()

{

cout << "Default of Derived" << endl;

}

Derived (int a)

{

cout << "Param of Derived : " << a << endl;

}

Derived(int x, int a) : Base(x)

{

cout << "Param of Derived " << a;

}

};

int main()

{

Derived d(25, 15);

}

Q-24: Create a program to calculate sine, cosine, and tangent of an angle in radians using hierarchical inheritance.

Sample test case:

|  |
| --- |
| angleInRadians = 3.14 / 4.0  Sine sine(angleInRadians);  Cosine cosine(angleInRadians);  Tangent tangent(angleInRadians);  sine.calculate() //Output: Sine of 0.785 radians: 0.706825  cosine.calculate() // Output: Cosine of 0.785 radians: 0.707388  tangent.calculate() //Output: Tangent of 0.785 radians: 0.999204 |

Solution:

#include <iostream>

#include <cmath>

#define M\_PI 3.14

using namespace std;

class TrigonometricFunction {

protected:

double angle;

public:

TrigonometricFunction(double a) : angle(a) {}

virtual double calculate() = 0;

};

class Sine : public TrigonometricFunction {

public:

Sine(double a) : TrigonometricFunction(a) {}

double calculate() override {

return sin(angle);

}

};

class Cosine : public TrigonometricFunction {

public:

Cosine(double a) : TrigonometricFunction(a) {}

double calculate() override {

return cos(angle);

}

};

class Tangent : public TrigonometricFunction {

public:

Tangent(double a) : TrigonometricFunction(a) {}

double calculate() override {

return tan(angle);

}

};

int main() {

double angleInRadians = M\_PI / 4.0; // 45 degrees in radians

Sine sine(angleInRadians);

Cosine cosine(angleInRadians);

Tangent tangent(angleInRadians);

cout << "Sine of " << angleInRadians << " radians: " << sine.calculate() << endl;

cout << "Cosine of " << angleInRadians << " radians: " << cosine.calculate() << endl;

cout << "Tangent of " << angleInRadians << " radians: " << tangent.calculate() << endl;

return 0;

}

Q-25: Implement a hierarchical inheritance system for educational institutions such as schools, colleges, and universities. Each educational institution has common attributes like name, location, and establishment year, along with specific attributes like the number of students and courses offered.

Sample test case:

|  |
| --- |
| Input:  EducationalInstitution\* institution1 = new School("ABC School", "City-A", 1990, 500);  institution1->displayInfo();  Output:  School Name: ABC School  Location: City-A  Establishment Year: 1990  Number of Students: 500 |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class representing an Educational Institution

class EducationalInstitution {

protected:

string name;

string location;

int establishmentYear;

public:

EducationalInstitution(string \_name, string \_location, int \_establishmentYear)

: name(\_name), location(\_location), establishmentYear(\_establishmentYear) {}

virtual void displayInfo() const = 0;

};

// Derived class representing a School, inheriting from EducationalInstitution

class School : public EducationalInstitution {

private:

int numStudents;

public:

School(string \_name, string \_location, int \_establishmentYear, int \_numStudents)

: EducationalInstitution(\_name, \_location, \_establishmentYear), numStudents(\_numStudents) {}

// Implementing the virtual function to display specific information for Schools.

void displayInfo() const override {

cout << "School Name: " << name << "\nLocation: " << location

<< "\nEstablishment Year: " << establishmentYear

<< "\nNumber of Students: " << numStudents << endl;

}

};

// Derived class representing a College, inheriting from EducationalInstitution

class College : public EducationalInstitution {

private:

int numStudents;

int numCourses;

public:

College(string \_name, string \_location, int \_establishmentYear, int \_numStudents, int \_numCourses)

: EducationalInstitution(\_name, \_location, \_establishmentYear), numStudents(\_numStudents), numCourses(\_numCourses) {}

// Implementing the virtual function to display specific information for Colleges.

void displayInfo() const override {

cout << "College Name: " << name << "\nLocation: " << location

<< "\nEstablishment Year: " << establishmentYear

<< "\nNumber of Students: " << numStudents

<< "\nNumber of Courses Offered: " << numCourses << endl;

}

};

// Derived class representing a University, inheriting from EducationalInstitution

class University : public EducationalInstitution {

private:

int numStudents;

int numDepartments;

public:

University(string \_name, string \_location, int \_establishmentYear, int \_numStudents, int \_numDepartments)

: EducationalInstitution(\_name, \_location, \_establishmentYear), numStudents(\_numStudents), numDepartments(\_numDepartments) {}

// Implementing the virtual function to display specific information for Universities.

void displayInfo() const override {

cout << "University Name: " << name << "\nLocation: " << location

<< "\nEstablishment Year: " << establishmentYear

<< "\nNumber of Students: " << numStudents

<< "\nNumber of Departments: " << numDepartments << endl;

}

};

int main() {

// Creating objects of the derived classes through the base class pointer.

EducationalInstitution\* institution1 = new School("ABC School", "City-A", 1990, 500);

EducationalInstitution\* institution2 = new College("XYZ College", "City-B", 1985, 1200, 40);

EducationalInstitution\* institution3 = new University("PQR University", "City-C", 1950, 8000, 15);

// Calling the displayInfo function, which will be dynamically dispatched based on the object's actual type.

institution1->displayInfo();

institution2->displayInfo();

institution3->displayInfo();

// Cleaning up the allocated memory.

delete institution1;

delete institution2;

delete institution3;

return 0;

}

Q-26: Design a hierarchy of employees, including a base class for Employee and derived classes for Manager, Engineer, and Technician. Each class should have specific attributes like designation and salary.

Sample test case:

|  |
| --- |
| Input:  Employee\* emp1 = new Manager("John Doe", 101, "Senior Manager", 80000);  emp1->displayDetails();  Output:  Name: John Doe  Employee ID: 101  Designation: Senior Manager  Salary: $80000 |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class Employee

class Employee {

protected:

string name;

int employeeID;

public:

Employee(string \_name, int \_employeeID) : name(\_name), employeeID(\_employeeID) {}

virtual void displayDetails() const = 0;

};

// Derived class Manager from Employee

class Manager : public Employee {

private:

string designation;

double salary;

public:

Manager(string \_name, int \_employeeID, string \_designation, double \_salary)

: Employee(\_name, \_employeeID), designation(\_designation), salary(\_salary) {}

// Override the base class function to display Manager's details

void displayDetails() const override {

cout << "Name: " << name << "\nEmployee ID: " << employeeID

<< "\nDesignation: " << designation << "\nSalary: $" << salary << endl;

}

};

// Derived class Engineer from Employee

class Engineer : public Employee {

private:

string designation;

double salary;

public:

Engineer(string \_name, int \_employeeID, string \_designation, double \_salary)

: Employee(\_name, \_employeeID), designation(\_designation), salary(\_salary) {}

// Override the base class function to display Engineer's details

void displayDetails() const override {

cout << "Name: " << name << "\nEmployee ID: " << employeeID

<< "\nDesignation: " << designation << "\nSalary: $" << salary << endl;

}

};

// Derived class Technician from Employee

class Technician : public Employee {

private:

string designation;

double salary;

public:

Technician(string \_name, int \_employeeID, string \_designation, double \_salary)

: Employee(\_name, \_employeeID), designation(\_designation), salary(\_salary) {}

// Override the base class function to display Technician's details

void displayDetails() const override {

cout << "Name: " << name << "\nEmployee ID: " << employeeID

<< "\nDesignation: " << designation << "\nSalary: $" << salary << endl;

}

};

int main() {

// Create instances of derived classes and assign them to base class pointers

Employee\* emp1 = new Manager("John Doe", 101, "Senior Manager", 80000);

Employee\* emp2 = new Engineer("Alice Smith", 202, "Software Engineer", 60000);

Employee\* emp3 = new Technician("Bob Johnson", 303, "Lab Technician", 40000);

// Display details of each employee using the base class pointer

emp1->displayDetails();

emp2->displayDetails();

emp3->displayDetails();

// Free the memory allocated for each employee

delete emp1;

delete emp2;

delete emp3;

return 0;

}

Q-27: Design a hierarchy for electronic devices, including a base class for Device and derived classes for Smartphone, Laptop, and Television. Each device should have specific attributes like screen size and battery life.

Sample test case:

|  |
| --- |
| Input:  Device\* device1 = new Smartphone("Apple", 6.1, 20);  device1->displaySpecs();  Output:  Brand: Apple  Screen Size: 6.1 inches  Battery Life: 20 hours |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class representing a generic electronic Device

class Device {

protected:

string brand;

public:

Device(string \_brand) : brand(\_brand) {}

// Pure virtual function to make this class abstract, ensuring it cannot be instantiated.

virtual void displaySpecs() const = 0;

};

// Derived class representing a Smartphone, inheriting from Device

class Smartphone : public Device {

private:

double screenSize;

int batteryLife;

public:

Smartphone(string \_brand, double \_screenSize, int \_batteryLife)

: Device(\_brand), screenSize(\_screenSize), batteryLife(\_batteryLife) {}

// Implementing the virtual function to provide specific behavior for Smartphones.

void displaySpecs() const override {

cout << "Brand: " << brand << "\nScreen Size: " << screenSize << " inches"

<< "\nBattery Life: " << batteryLife << " hours" << endl;

}

};

// Derived class representing a Laptop, inheriting from Device

class Laptop : public Device {

private:

double screenSize;

int batteryLife;

public:

Laptop(string \_brand, double \_screenSize, int \_batteryLife)

: Device(\_brand), screenSize(\_screenSize), batteryLife(\_batteryLife) {}

// Implementing the virtual function to provide specific behavior for Laptops.

void displaySpecs() const override {

cout << "Brand: " << brand << "\nScreen Size: " << screenSize << " inches"

<< "\nBattery Life: " << batteryLife << " hours" << endl;

}

};

// Derived class representing a Television, inheriting from Device

class Television : public Device {

private:

double screenSize;

public:

Television(string \_brand, double \_screenSize)

: Device(\_brand), screenSize(\_screenSize) {}

// Implementing the virtual function to provide specific behavior for Televisions.

void displaySpecs() const override {

cout << "Brand: " << brand << "\nScreen Size: " << screenSize << " inches" << endl;

}

};

int main() {

// Creating objects of the derived classes through the base class pointer.

Device\* device1 = new Smartphone("Apple", 6.1, 20);

Device\* device2 = new Laptop("Dell", 15.6, 8);

Device\* device3 = new Television("Samsung", 55);

// Calling the displaySpecs function, which will be dynamically dispatched based on the object's actual type.

device1->displaySpecs();

device2->displaySpecs();

device3->displaySpecs();

// Cleaning up the allocated memory.

delete device1;

delete device2;

delete device3;

return 0;

}

Q-28: Design a hierarchy of vehicles, including a base class for Vehicle and derived classes for Car and Bike. Each class should have specific attributes like the number of wheels and fuel type.

Sample test case:

|  |
| --- |
| Input:  Car car(4, "Petrol", 4);  car.displayInfo();  Output:  Car Info:  Number of Wheels: 4  Fuel Type: Petrol  Number of Doors: 4 |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class representing a Vehicle

class Vehicle {

protected:

int numWheels;

string fuelType;

public:

// Constructor for the Vehicle class

Vehicle(int \_numWheels, string \_fuelType) : numWheels(\_numWheels), fuelType(\_fuelType) {}

// Method to display information about the vehicle

void displayInfo() const {

cout << "Number of Wheels: " << numWheels << "\nFuel Type: " << fuelType << endl;

}

};

// Derived class representing a Car, inheriting from Vehicle

class Car : public Vehicle {

private:

int numDoors;

public:

// Constructor for the Car class, which also calls the constructor of the base class (Vehicle)

Car(int \_numWheels, string \_fuelType, int \_numDoors)

: Vehicle(\_numWheels, \_fuelType), numDoors(\_numDoors) {}

// Method to display information about the car, including information from the base class

void displayInfo() const {

// Calling the displayInfo method of the base class (Vehicle)

Vehicle::displayInfo();

cout << "Number of Doors: " << numDoors << endl;

}

};

// Derived class representing a Bike, inheriting from Vehicle

class Bike : public Vehicle {

private:

bool hasGears;

public:

// Constructor for the Bike class, which also calls the constructor of the base class (Vehicle)

Bike(int \_numWheels, string \_fuelType, bool \_hasGears)

: Vehicle(\_numWheels, \_fuelType), hasGears(\_hasGears) {}

// Method to display information about the bike, including information from the base class

void displayInfo() const {

// Calling the displayInfo method of the base class (Vehicle)

Vehicle::displayInfo();

cout << "Has Gears: " << (hasGears ? "Yes" : "No") << endl;

}

};

int main() {

// Creating objects of the derived classes (Car and Bike)

Car car(4, "Petrol", 4);

Bike bike(2, "None", true);

cout << "Car Info:\n";

// Calling the displayInfo method of the Car class

car.displayInfo();

cout << "\nBike Info:\n";

// Calling the displayInfo method of the Bike class

bike.displayInfo();

return 0;

}

Q-29: Create a hierarchy of game characters with a base class Character and derived classes Warrior, Mage, and Archer. Implement a method specialAbility() in the base class and override it in the derived classes to display the special ability of

each character type.

Sample test case:

|  |
| --- |
| Input:  Character\* warrior = new Warrior("Aragorn");  warrior->specialAbility();  Output:  Aragorn is a Warrior. Special Ability: Devastating Sword Strike!  Input:  Character\* mage = new Mage("Gandalf");  mage->specialAbility();  Output:  Gandalf is a Mage. Special Ability: Fireball Attack!  Input:  Character\* archer = new Archer("Legolas");  archer->specialAbility();  Output:  Legolas is an Archer. Special Ability: Precise Arrow Shot! |

Solution:

#include <iostream>

#include <string>

using namespace std;

// Base class Character

class Character {

protected:

string name;

public:

Character(string \_name) : name(\_name) {}

virtual void specialAbility() const = 0;

};

// Derived class Warrior from Character

class Warrior : public Character {

public:

Warrior(string \_name) : Character(\_name) {}

// Override the base class function to display Warrior's special ability

void specialAbility() const override {

cout << name << " is a Warrior. Special Ability: Devastating Sword Strike!" << endl;

}

};

// Derived class Mage from Character

class Mage : public Character {

public:

Mage(string \_name) : Character(\_name) {}

// Override the base class function to display Mage's special ability

void specialAbility() const override {

cout << name << " is a Mage. Special Ability: Fireball Attack!" << endl;

}

};

// Derived class Archer from Character

class Archer : public Character {

public:

Archer(string \_name) : Character(\_name) {}

// Override the base class function to display Archer's special ability

void specialAbility() const override {

cout << name << " is an Archer. Special Ability: Precise Arrow Shot!" << endl;

}

};

int main() {

// Create instances of derived classes

Character\* warrior = new Warrior("Aragorn");

Character\* mage = new Mage("Gandalf");

Character\* archer = new Archer("Legolas");

// Call specialAbility() for each character

warrior->specialAbility();

mage->specialAbility();

archer->specialAbility();

// Free the memory allocated for each character

delete warrior;

delete mage;

delete archer;

return 0;

}

Q-30: Create a media library in C++ with a base class MediaItem and two derived classes Book and Movie. Each media item has a title and a year of release. Implement a virtual function display() in the MediaItem class, and override it in the derived classes to display specific details of a book and a movie.

Sample test case:

|  |
| --- |
| Input:  MediaItem\* item1 = new Book("The Great Gatsby", 1925, "F. Scott Fitzgerald");  item1->display();  Output:  Title: The Great Gatsby  Author: F. Scott Fitzgerald  Year of Release: 1925  Input:  MediaItem\* item2 = new Movie("Inception", 2010, "Christopher Nolan");  item2->display();  Output:  Title: Inception  Director: Christopher Nolan  Year of Release: 2010 |

Soution:

#include <iostream>

#include <string>

using namespace std;

// Base class representing a Media Item

class MediaItem {

protected:

string title;

int year;

public:

// Constructor for the MediaItem class

MediaItem(string \_title, int \_year) : title(\_title), year(\_year) {}

// Virtual method to display information about the media item

virtual void display() const {

cout << "Title: " << title << "\nYear of Release: " << year << endl;

}

};

// Derived class representing a Book, inheriting from MediaItem

class Book : public MediaItem {

private:

string author;

public:

// Constructor for the Book class, which also calls the constructor of the base class (MediaItem)

Book(string \_title, int \_year, string \_author) : MediaItem(\_title, \_year), author(\_author) {}

// Override the display method to include the author's information

void display() const override {

cout << "Title: " << title << "\nAuthor: " << author << "\nYear of Release: " << year << endl;

}

};

// Derived class representing a Movie, inheriting from MediaItem

class Movie : public MediaItem {

private:

string director;

public:

// Constructor for the Movie class, which also calls the constructor of the base class (MediaItem)

Movie(string \_title, int \_year, string \_director) : MediaItem(\_title, \_year), director(\_director) {}

// Override the display method to include the director's information

void display() const override {

cout << "Title: " << title << "\nDirector: " << director << "\nYear of Release: " << year << endl;

}

};

int main() {

// Creating objects of the derived classes (Book and Movie) using base class pointers (MediaItem\*)

MediaItem\* item1 = new Book("The Great Gatsby", 1925, "F. Scott Fitzgerald");

MediaItem\* item2 = new Movie("Inception", 2010, "Christopher Nolan");

// Calling the display method of the Book class (will override the display method in MediaItem)

item1->display();

// Calling the display method of the Movie class (will override the display method in MediaItem)

item2->display();

// Cleaning up the allocated memory.

delete item1;

delete item2;

return 0;

}