**Question 1: Shape Hierarchy with Virtual draw()**

**Create a base class Shape with a pure virtual function draw() that has no implementation.**

**Derive classes like Circle, Square, and Triangle from Shape, each overriding draw() to provide their specific drawing behavior (e.g., using cout for simple output or more advanced graphics libraries).**

**Write a main function that creates an array of pointers to Shape objects. Populate the array with instances of derived classes (polymorphism).**

**Iterate through the array and call draw() on each pointer using a loop. Observe how the correct draw() implementation is invoked based on the object's type at runtime.**

#include <iostream>

#include <vector>

using namespace std;

// Base class Shape with a pure virtual function draw()

class Shape {

public:

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

virtual ~Shape() {} // Virtual destructor for proper cleanup of derived class objects

};

// Derived class Circle

class Circle : public Shape {

public:

void draw() const override {

cout << "Drawing a Circle" << endl;

}

};

// Derived class Square

class Square : public Shape {

public:

void draw() const override {

cout << "Drawing a Square" << endl;

}

};

// Derived class Triangle

class Triangle : public Shape {

public:

void draw() const override {

cout << "Drawing a Triangle" << endl;

}

};

int main() {

// Creating an array of pointers to Shape objects

vector<Shape\*> shapes;

// Populating the array with instances of derived classes

shapes.push\_back(new Circle());

shapes.push\_back(new Square());

shapes.push\_back(new Triangle());

// Iterating through the array and calling draw() on each pointer

for (const auto& shape : shapes) {

shape->draw(); // Polymorphic call

}

// Cleaning up allocated memory

for (auto& shape : shapes) {

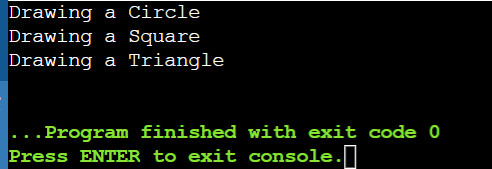
delete shape;

}

return 0;

}

**Output:**

****

**Question 2: Abstract Animal Class with Virtual makeSound()**

**Design an abstract base class Animal with a pure virtual function makeSound() that each derived class must implement differently (e.g., cout for "Meow", "Woof", etc.).**

**Create concrete classes Cat, Dog, and potentially others, inheriting from Animal and overriding makeSound().**

**In main, create a function playAnimalSound that takes an Animal reference as an argument. Inside, call makeSound() on the reference. Demonstrate runtime polymorphism by passing objects of different derived classes to playAnimalSound and observing the correct sound being played.**

#include <iostream>

using namespace std;

class Animal {

public:

virtual void makeSound() const = 0;

virtual ~Animal() {}

};

class Cat : public Animal {

public:

void makeSound() const override {

cout << "Meow" << endl;

}

};

class Dog : public Animal {

public:

void makeSound() const override {

cout << "Woof" << endl;

}

};

void playAnimalSound(const Animal& animal) {

animal.makeSound();

}

int main() {

Cat cat;

Dog dog;

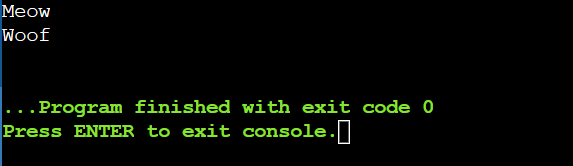
playAnimalSound(cat); // Output: Meow

playAnimalSound(dog); // Output: Woof

return 0;

}

**Output:**

****

**Question 3: Area Calculation with Virtual Destructors**

**Define a base class Shape with a member function area() that returns 0 (since it's a base class). Make Shape abstract using a pure virtual destructor.**

**Derive classes Circle, Square, and Triangle, each overriding area() with their specific area calculation formulas.**

**In main, create an array of pointers to Shape objects. Allocate memory dynamically for each object using new from the derived classes.**

**Iterate through the array and call area() on each pointer. Notice how the appropriate area() implementation is chosen based on the object's type at runtime, even though the array holds Shape pointers.**

**Crucially, remember to delete each object using delete to avoid memory leaks. This demonstrates the importance of virtual destructors in polymorphism scenarios with dynamic memory allocation.**

#include <iostream>

#include <cmath>

using namespace std;

class Shape {

public:

virtual int area() const {

return 0; }

virtual ~Shape() = 0; // Pure virtual destructor

};

Shape::~Shape() {}

class Circle : public Shape {

private:

int radius;

public:

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

int area() const override {

return M\_PI \* radius \* radius; }

};

class Square : public Shape {

private:

int side;

public:

Square(int s) : side(s) {}

int area() const override {

return side \* side; }

};

class Triangle : public Shape {

private:

int base, height;

public:

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

int area() const override {

return 0.5 \* base \* height; }

};

int main() { // Create an array

Shape\* shapes[] = {new Circle(10), new Square(9), new Triangle(3, 6)};

for (Shape\* shape : shapes) { // Iterate through the array and call area() on each pointer

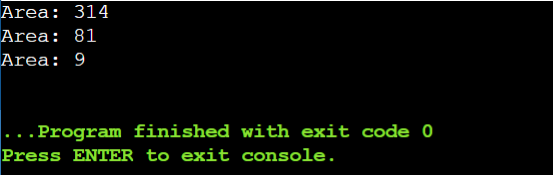
cout << "Area: " << shape->area() << endl;

}

return 0;

}

**Output:**

****

**Question 4: Virtual Destructor and Slicing**

**Create a base class Shape with a member variable color and a virtual destructor.**

**Derive a class Circle from Shape that adds a member variable radius.**

**In main, create a Circle object on the stack and assign it to a Shape reference. Then, delete the reference.**

**Explain why this leads to object slicing (the radius member is not deleted) and the importance of virtual destructors in preventing it. Discuss how virtual destructors ensure the complete destruction of derived class objects when accessed through base class pointers or references.**

# include <iostream>

using namespace std;

class Shape {

protected:

string color;

public:

Shape(const string& c) : color(c) {}

virtual ~Shape() {

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

}

};

class Circle : public Shape {

private:

double radius;

public:

Circle(const string& c, double r) : Shape(c), radius(r) {}

~Circle() override {

cout << "Circle destructor called" << endl;

}

};

int main() {

Circle circle("red", 5.0);

Shape& shapeRef = circle;

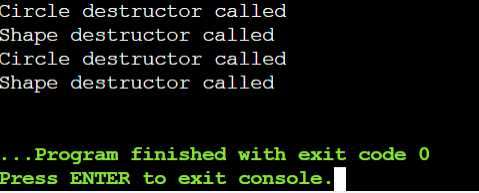
Shape\* shapePtr = new Circle("blue", 10.0);

delete shapePtr; // Correctly calls the Circle destructor due to the virtual destructor in Shape

return 0;

}

**Output:**

****

**Question 5: Runtime Type Information (RTTI)**

**Create base and derived classes with virtual functions.**

**In main, use the typeid operator to obtain runtime type information of objects.**

**Write a function identifyObject that takes a reference to an object and uses typeid to check if it's of a specific derived class type. Based on the type, perform different actions or print messages.**

**Discuss the pros and cons of using RTTI. While it can provide flexibility in certain cases, overuse can sometimes make code less type-safe and harder to maintain. Consider alternative design patterns when possible.**

#include <iostream>

#include <typeinfo>

using namespace std;

class Shape {

public:

virtual void draw() const {

cout << "Drawing Shape" << endl; }

virtual ~Shape() {}

};

class Circle : public Shape {

public:

void draw() const override {

cout << "Drawing Circle" << endl; }

};

class Square : public Shape {

public:

void draw() const override {

cout << "Drawing Square" << endl; }

};

void identifyObject(const Shape& shape) {

if (typeid(shape) == typeid(Circle)) {

cout << "This is a Circle." << endl;

} else if (typeid(shape) == typeid(Square)) {

cout << "This is a Square." << endl;

} else {

cout << "This is an unknown Shape." << endl;

}

shape.draw();

}

int main() {

Circle circle;

Square square;

Shape\* shapes[] = { &circle, &square };

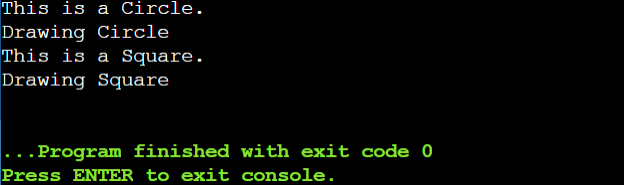
for (Shape\* shape : shapes) {

identifyObject(\*shape); }

return 0;

}

**Output:**

****

**5. RTTI (Run-Time Type Information) is like a detective tool in programming. It lets you figure out the exact type of an object at runtime, even if you don't know it beforehand. This can be really handy for things like:**

**\*Pros:\***

**\* \*Flexibility:\*** You can write code that works with different types of objects without knowing their exact types in advance. This is great for things like polymorphism, where you want to handle different objects in a similar way.

**\* \*Dynamic Behavior:\*** You can change the behavior of your code based on the type of object you're working with. This is useful for things like plugins or customizability.

**\*Cons:\***

**\* \*Less Type-Safe:\*** Since you're not always sure what type of object you're dealing with, there's a higher chance of errors. You might try to use a method that doesn't exist on that particular type, leading to crashes.

**\* \*Harder to Maintain:\*** Code that uses RTTI can be harder to understand and maintain because it's not always clear what types of objects are being used. This can make it difficult to track down bugs and make changes.

**\*Alternatives:\***

**\* \*Design Patterns:\*** There are many design patterns that can help you achieve the same flexibility without relying on RTTI. For example, you can use the Strategy pattern to encapsulate different behaviors in separate classes.

**\* \*Templates:\*** Templates allow you to write code that works with different types without needing to know the exact type at compile time. This can be a good alternative to RTTI for situations where you need to work with different types but don't need to know the exact type at runtime.

**\*In short:\*** RTTI can be useful in certain situations, but it's important to use it sparingly. If you can achieve the same functionality with other techniques, it's often a better idea to do so.