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>

using namespace std;

class Shape {

public:

virtual void draw() const = 0;

virtual ~Shape() {}

};

class Circle : public Shape {

public:

void draw() const override {

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

}

};

class Square : public Shape {

public:

void draw() const override {

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

}

};

class Triangle : public Shape {

public:

void draw() const override {

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

}

};

int main() {

Shape\* shapes[3];

shapes[0] = new Circle();

shapes[1] = new Square();

shapes[2] = new Triangle();

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

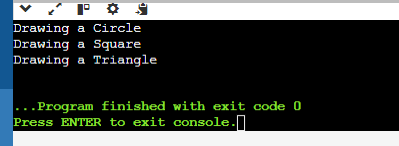
shapes[i]->draw();

}

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;

}

};

class Cow : public Animal {

public:

void makeSound() const override{

cout<< "Humba" << endl;

}

};

void playAnimalSound( const Animal& animal) {

animal.makeSound();

}

int main() {

Cat cat;

Dog dog;

Cow cow;

playAnimalSound(cat);

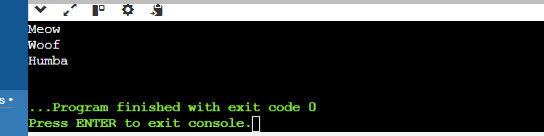
playAnimalSound(dog);

playAnimalSound(cow);

return 0;

}

Output:



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>

using namespace std;

class Shape {

public:

virtual double area() const {

return 0.0;

}

virtual ~Shape() = 0;

};

Shape::~Shape() {}

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return 3.14159 \* radius \* radius;

}

};

class Square : public Shape {

private:

double side;

public:

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

double area() const override {

return side \* side;

}

};

class Triangle : public Shape {

private:

double base, height;

public:

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

double area() const override {

return 0.5 \* base \* height;

}

};

int main() {

Shape\* shapes[3];

shapes[0] = new Circle(7.0);

shapes[1] = new Square(9.0);

shapes[2] = new Triangle(2.0, 8.0);

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

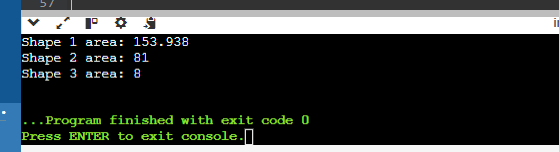
cout << "Shape " << i + 1 << " area: " << shapes[i]->area() << endl;

}

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>

#include <string>

using namespace std;

class Shape {

public:

string color;

Shape(const string& c) : color(c) { cout << "Shape constructor" << endl; }

~Shape() { cout << "Shape destructor" << endl; }

};

class Circle : public Shape {

public:

double radius;

Circle(const string& c, double r) : Shape(c), radius(r){cout << "Circle constructor" <<endl; }

~Circle() { cout << "Circle destructor" << endl; }

};

int main() {

Circle circle("Red", 5.0);

Shape& shapeRef = circle;

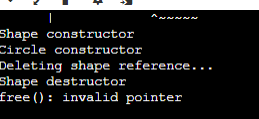
cout << "Deleting shape reference..." << endl;

delete &shapeRef;

return 0;

}

Output:



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 a Shape" << endl;

}

virtual ~Shape() {}

};

class Circle : public Shape {

public:

void draw() const override {

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

}

};

class Square : public Shape {

public:

void draw() const override {

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

}

};

class Triangle : public Shape {

public:

void draw() const override {

cout << "Drawing a Triangle" << 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 if (typeid(shape) == typeid(Triangle)) {

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

} else {

cout << "Unknown Shape." << endl;

}

}

int main() {

Circle circle;

Square square;

Triangle triangle;

identifyObject(circle);

identifyObject(square);

identifyObject(triangle);

return 0;

}

Output:

