Day-9:

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() = 0;

};

class Circle : public Shape {

public:

void draw() override {

cout << "Drawing Circle" << endl;

}

};

class Square : public Shape {

public:

void draw() override {

cout << "Drawing Square" << endl;

}

};

class Triangle : public Shape {

public:

void draw() override {

cout << "Drawing Triangle" << endl;

}

};

int main() {

const int size = 3;

Shape\* shapes[size];

shapes[0] = new Circle();

shapes[1] = new Square();

shapes[2] = new Triangle();

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

shapes[i]->draw();

}

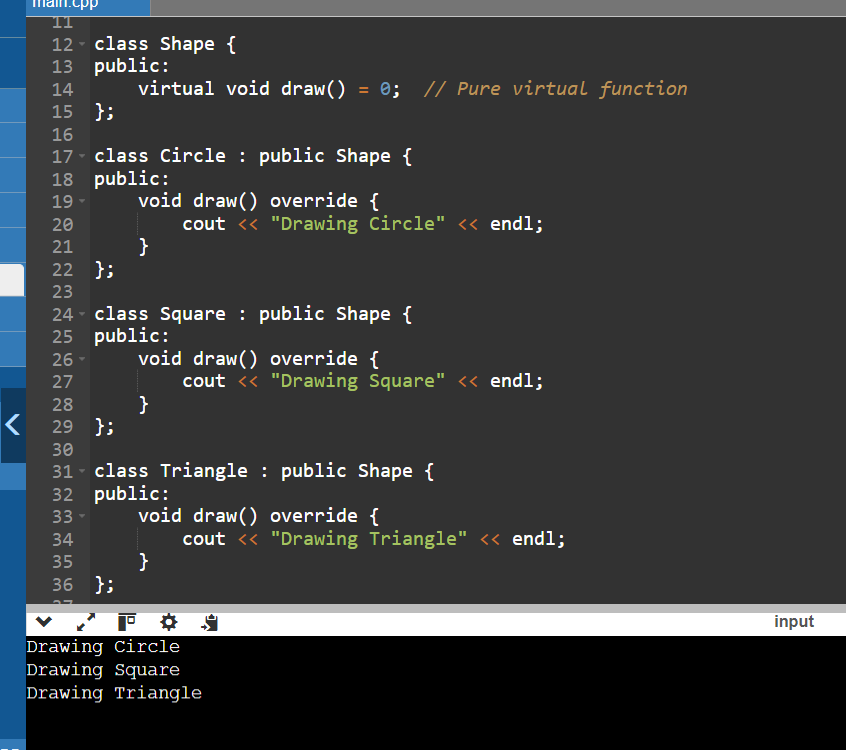
for (int i = 0; i < size; ++i) {

delete shapes[i];

}

return 0;

}

Output:

#include <iostream>

using namespace std;

class Animal {

public:

virtual void makeSound() = 0; // Pure virtual function

};

class Cat : public Animal {

public:

void makeSound() override {

cout << "Meow" << endl;

}

};

class Dog : public Animal {

public:

void makeSound() override {

cout << "Woof" << endl;

}

};

void playAnimalSound(Animal& animal) {

animal.makeSound();

}

int main() {

Cat cat;

Dog dog;

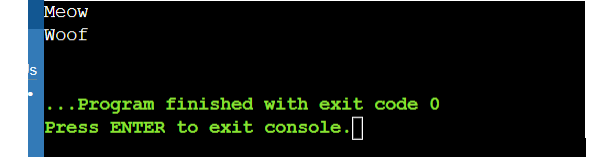
playAnimalSound(cat);

playAnimalSound(dog);

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> // For M\_PI

class Shape {

public:

virtual double area() const = 0;

virtual ~Shape() = 0;

};

Shape::~Shape() {}

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return M\_PI \* 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;

double height;

public:

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

double area() const override {

return 0.5 \* base \* height;

}

};

int main() {

const int size = 3;

Shape\* shapes[size];

shapes[0] = new Circle(3.0);

shapes[1] = new Square(4.0);

shapes[2] = new Triangle(5.0, 2.0);

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

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

}

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

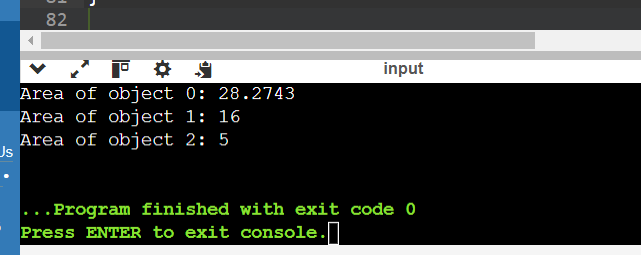
delete shapes[i];

}

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>

class Shape {

private:

std::string color;

public:

Shape(const std::string& col) : color(col) {}

virtual ~Shape() { std::cout << "Shape destructor\n"; }

virtual void printColor() const {

std::cout << "Color: " << color << std::endl;

}

};

class Circle : public Shape {

private:

double radius;

public:

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

void printRadius() const {

std::cout << "Radius: " << radius << std::endl;

}

};

int main() {

Circle c("Red", 5.0);

Shape& shapeRef = c;

delete &shapeRef;

return 0;

}

Output:

Shape destructor

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>

class Shape {

public:

virtual void draw() const {

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

}

};

class Circle : public Shape {

public:

void draw() const override {

std::cout << "Drawing a circle." << std::endl;

}

};

class Square : public Shape {

public:

void draw() const override {

std::cout << "Drawing a square." << std::endl;

}

};

void identifyObject(const Shape& shape) {

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

std::cout << "Identified as a Circle." << std::endl;

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

std::cout << "Identified as a Square." << std::endl;

} else {

std::cout << "Identified as a generic Shape." << std::endl;

}

}

int main() {

Circle circle;

Square square;

Shape genericShape;

std::cout << "Type of circle: " << typeid(circle).name() << std::endl;

std::cout << "Type of square: " << typeid(square).name() << std::endl;

std::cout << "Type of genericShape: " << typeid(genericShape).name() << std::endl;

std::cout << "\nUsing identifyObject function:" << std::endl;

identifyObject(circle);

identifyObject(square);

identifyObject(genericShape);

return 0;

}

Output:

Type of circle: 6Circle

Type of square: 6Square

Type of genericShape: 5Shape

Polymorphism:

Design a class hierarchy for a simple graphic editor with base class Shape and derived classes Circle, Rectangle, and Triangle. Implement a virtual function draw() in the base class and override it in the derived classes. Write a function that takes a Shape\* and calls its draw() method.

#include <iostream>

class Shape {

public:

virtual ~Shape() {} // Virtual destructor

virtual void draw() const {

std::cout << "Drawing a generic Shape\n";

}

};

class Circle : public Shape {

public:

void draw() const override {

std::cout << "Drawing a Circle\n";

}

};

class Rectangle : public Shape {

public:

void draw() const override {

std::cout << "Drawing a Rectangle\n";

}

};

class Triangle : public Shape {

public:

void draw() const override {

std::cout << "Drawing a Triangle\n";

}

};

void drawShape(const Shape\* shape) {

shape->draw();

}

int main() {

Circle circle;

Rectangle rectangle;

Triangle triangle;

drawShape(&circle);

drawShape(&rectangle);

drawShape(&triangle);

return 0;

}

Output:

Drawing a Circle

Drawing a Rectangle

Drawing a Triangle

Static Members:

Create a class Account that has a static data member totalAccounts to keep track of the number of accounts created. Implement necessary constructors and destructors to update totalAccounts. Write a function to display the total number of accounts.

#include <iostream>

class Account {

private:

static int totalAccounts;

public:

Account() {

++totalAccounts;

}

~Account() {

--totalAccounts;

}

static void displayTotalAccounts() {

std::cout << "Total accounts: " << totalAccounts << std::endl;

}

};

int Account::totalAccounts = 0;

int main() {

Account acc1;

Account acc2;

Account::displayTotalAccounts();

Account acc3;

Account::displayTotalAccounts();

return 0;

}

Output:

Total accounts: 2

Total accounts: 3

Friend Functions:

Implement a class Box that has private data members length, breadth, and height. Write a friend function volume() that calculates and returns the volume of the box. Create objects of Box and use the friend function to compute their volumes.

#include <iostream>

class Box {

private:

double length;

double breadth;

double height;

public:

Box(double l, double b, double h)

: length(l), breadth(b), height(h) {}

friend double volume(const Box& box);

};

double volume(const Box& box) {

return box.length \* box.breadth \* box.height;

}

int main() {

Box box1(3.0, 4.0, 5.0);

Box box2(1.5, 2.5, 3.5);

std::cout << "Volume of box1: " << volume(box1) << std::endl;

std::cout << "Volume of box2: " << volume(box2) << std::endl;

return 0;

}

Output:

Volume of box1: 60

Volume of box2: 13.125

Templates:

Write a template class Array that can store an array of any data type. Include member functions to perform operations like adding an element, removing an element, and displaying the array. Demonstrate the functionality with different data types.

#include <iostream>

#include <vector>

template <typename T>

class Array {

private:

std::vector<T> elements;

public:

Array() {}

~Array() {}

void addElement(const T& element) {

elements.push\_back(element);

}

void removeElement(int index) {

if (index >= 0 && index < elements.size()) {

elements.erase(elements.begin() + index);

} else {

std::cout << "Invalid index. Element removal failed.\n";

}

}

void display() const {

std::cout << "Array elements:";

for (const auto& elem : elements) {

std::cout << " " << elem;

}

std::cout << std::endl;

}

};

int main() {

Array<int> intArray;

Array<double> doubleArray;

Array<std::string> stringArray;

intArray.addElement(10);

intArray.addElement(20);

intArray.addElement(30);

doubleArray.addElement(1.5);

doubleArray.addElement(2.5);

doubleArray.addElement(3.5);

stringArray.addElement("Hello");

stringArray.addElement("Vasu");

intArray.display();

doubleArray.display();

stringArray.display();

intArray.removeElement(1);

doubleArray.removeElement(2);

stringArray.removeElement(0);

intArray.display();

doubleArray.display();

stringArray.display();

return 0;

}

Output:

Array elements: 10 20 30

Array elements: 1.5 2.5 3.5

Array elements: Hello Vasu

Array elements: 10 30

Array elements: 1.5 2.5

Array elements: Vasu

Pointers:

Design a class Student with data members name and age. Create an array of Student objects dynamically using pointers. Implement functions to set and display the details of students. Also, write a function to deallocate the memory.

#include <iostream>

#include <string>

class Student {

private:

std::string name;

int age;

public:

Student() : name(""), age(0) {}

Student(const std::string& n, int a) : name(n), age(a)

void setName(const std::string& n) {

name = n;

}

void setAge(int a) {

age = a;

}

std::string getName() const {

return name;

}

int getAge() const {

return age;

}

};

int main() {

int numStudents;

std::cout << "Enter the number of students: ";

std::cin >> numStudents;

std::cin.ignore();

Student\*\* students = new Student\*[numStudents];

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

students[i] = new Student();

std::cout << "Enter details for Student " << i + 1 << ":" << std::endl;

std::string name;

int age;

std::cout << "Name: ";

std::getline(std::cin, name);

students[i]->setName(name);

std::cout << "Age: ";

std::cin >> age;

std::cin.ignore();

students[i]->setAge(age);

}

std::cout << "\nDetails of all students:" << std::endl;

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

std::cout << "Student " << i + 1 << ":" << std::endl;

std::cout << "Name: " << students[i]->getName() << std::endl;

std::cout << "Age: " << students[i]->getAge() << std::endl;

}

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

delete students[i];

}

delete[] students;

return 0;

}

Output:

Enter the number of students: 3

Enter details for Student 1:

Name: vasu

Age: 20

Enter details for Student 2:

Name:ravi

Age: 22

Enter details for Student 3:

Name: siva

Age: 21

Details of all students:

Student 1:

Name: vasu

Age: 20

Student 2:

Name: ravi

Age: 22

Student 3:

Name: siva

Age: 21

Polymorphism with Abstract Classes:

Create an abstract class Animal with a pure virtual function sound(). Derive classes Dog, Cat, and Cow from Animal and override the sound() function in each derived class. Write a program to demonstrate polymorphism using these classes.

#include <iostream>

#include <vector>

class Animal {

public:

virtual void sound() const = 0;

};

class Dog : public Animal {

public:

void sound() const override {

std::cout << "Bark!" << std::endl;

}

};

class Cat : public Animal {

public:

void sound() const override {

std::cout << "Meow!" << std::endl;

}

};

class Cow : public Animal {

public:

void sound() const override {

std::cout << "Moo!" << std::endl;

}

};

int main() {

std::vector<Animal\*> animals;

animals.push\_back(new Dog());

animals.push\_back(new Cat());

animals.push\_back(new Cow());

for (const auto& animal : animals) {

animal->sound();

}

for (auto& animal : animals) {

delete animal;

}

animals.clear();

return 0;

}

Output:

Bark!

Meow!

Moo!

Static Member Functions:

Implement a class Math that has static member functions for basic mathematical operations like addition, subtraction, multiplication, and division. Demonstrate the use of these functions without creating an object of the class.

#include <iostream>

class Math {

public:

static int add(int a, int b) {

return a + b;

}

static int subtract(int a, int b) {

return a - b;

}

static int multiply(int a, int b) {

return a \* b;

}

static double divide(double a, double b) {

if (b == 0) {

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

return 0;

}

return a / b;

}

};

int main() {

int result\_add = Math::add(10, 5);

std::cout << "10 + 5 = " << result\_add << std::endl;

int result\_subtract = Math::subtract(20, 8);

std::cout << "20 - 8 = " << result\_subtract << std::endl;

int result\_multiply = Math::multiply(4, 6);

std::cout << "4 \* 6 = " << result\_multiply << std::endl;

double result\_divide = Math::divide(15.0, 3.0);

std::cout << "15 / 3 = " << result\_divide << std::endl;

return 0;

}

Output:

10 + 5 = 15

20 - 8 = 12

4 \* 6 = 24

15 / 3 = 5

Friend Classes:

Create two classes Alpha and Beta. Make Beta a friend class of Alpha so that it can access private data members of Alpha. Implement functions in Beta to manipulate the private data of Alpha.

#include <iostream>

class Beta;

class Alpha {

private:

int privateData;

public:

Alpha(int data) : privateData(data) {}

friend class Beta;

void displayPrivateData() {

std::cout << "Alpha's private data: " << privateData << std::endl;

}

};

class Beta {

public:

void modifyAlphaPrivateData(Alpha &alpha, int newData) {

alpha.privateData = newData;

std::cout << "Beta modified Alpha's private data to: " << alpha.privateData << std::endl;

}

};

int main() {

Alpha alphaObj(42);

Beta betaObj;

alphaObj.displayPrivateData();

betaObj.modifyAlphaPrivateData(alphaObj, 99);

alphaObj.displayPrivateData();

return 0;

}

Output:

