MOUNIKA (C++ 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>

#include <vector>

using namespace std;

class Shape {

public:

virtual void draw() const = 0;

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;

}

};

class Triangle : public Shape {

public:

void draw() const override {

cout << "Drawing Triangle" << endl;

}

};

int main() {

vector<Shape\*> shapes { new Circle(), new Square(), new Triangle() };

for (Shape\* shape : shapes) {

shape->draw();

}

for (Shape\* shape : shapes) {

delete shape;

}

return 0;

}

OUTPUT:

Drawing Circle

Drawing Square

Drawing Triangle

**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 AnimalSound(const Animal& animal) {

animal.makeSound();

}

int main() {

Cat myCat;

Dog myDog;

AnimalSound(myCat);

AnimalSound(myDog);

return 0;

}

OUTPUT:

Meow!

Woof!

**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>

#include <vector>

using namespace std;

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() {

vector<Shape\*> shapes {

new Circle(3.0),

new Square(4.0),

new Triangle(5.0, 2.0)

};

for (Shape\* shape : shapes) {

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

}

for (Shape\* shape : shapes) {

delete shape;

}

return 0;

}

OUTPUT:  
Area: 28.2743

Area: 16

Area: 5

**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 {

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() {

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

}

};

int main() {

Circle myCircle("Red", 5.0);

Shape& shapeRef = myCircle;

delete &shapeRef;

return 0;

}

OUTPUT:

Circle destructor called.

Shape destructor called.

Segmentation fault

**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>

using namespace std;

class Account {

private:

static int totalAccounts;

int accountNumber;

public:

Account() {

totalAccounts++;

accountNumber = totalAccounts;

cout << "Account " << accountNumber << " created" << endl;

}

~Account() {

totalAccounts--;

cout << "Account " << accountNumber << " deleted" <<endl;

}

static void displayTotalAccounts() {

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

}

};

int Account::totalAccounts = 0;

int main() {

Account acc1;

Account::displayTotalAccounts();

return 0;

}

OUTPUT:  
Account 1 created

Total accounts: 1

Account 1 deleted

**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>

using namespace std;

class Box {

private:

int length;

int breadth;

int height;

public:

Box(int len, int br, int h) : length(len), breadth(br), height(h) {}

friend int volume(const Box& box);

};

int volume(const Box& box) {

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

}

int main() {

Box box1(3, 4, 5);

Box box2(2, 1, 4);

Box box3(2, 3, 2);

cout << "Volume of Box 1: " << volume(box1) << endl;

cout << "Volume of Box 2: " << volume(box2) <<endl;

cout << "Volume of Box 3: " << volume(box3) << endl;

return 0;

}

OUTPUT:

Volume of Box 1: 60

Volume of Box 2: 8

Volume of Box 3: 12

**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>

using namespace std;

class Student {

private:

string name;

int age;

public:

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

void setDetails(const std::string& n, int a) {

name = n;

age = a;

}

void displayDetails() const {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

const int numStudents = 3;

Student\* students = new Student[numStudents]{

Student("A", 20),

Student("B", 21),

Student("C", 19)

};

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

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

students[i].displayDetails();

}

delete[] students;

return 0;

}

OUTPUT:

Student 1: Name: A, Age: 20

Student 2: Name: B, Age: 21

Student 3: Name: C, Age: 19

**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>

using namespace std;

class Animal {

public:

virtual void sound() const = 0;

};

class Dog : public Animal {

public:

void sound() const override {

cout << "Dog: Woof woof!" << endl;

}

};

class Cat : public Animal {

public:

void sound() const override {

cout << "Cat: Meow meow!" << endl;

}

};

class Cow : public Animal {

public:

void sound() const override {

cout << "Cow:amba amba!" << endl;

}

};

int main() {

Dog myDog;

Cat myCat;

Cow myCow;

Animal\* animals[] = {&myDog, &myCat, &myCow};

for (auto animal : animals) {

animal->sound();

}

return 0;

}

OUTPUT:

Dog: Woof woof!

Cat: Meow meow!

Cow:amba amba!

**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>

using namespace std;

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) {

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

return 0;

}

return a / b;

}

};

int main() {

cout << "Addition: " << Math::add(10, 5) << endl;

cout << "Subtraction: " << Math::subtract(10, 5) << endl;

cout << "Multiplication: " << Math::multiply(10, 5) << endl;

cout << "Division: " << Math::divide(10.0, 5.0) << endl;

cout << "Division: " << Math::divide(10.0, 0.0) << endl;

return 0;

}

OUTPUT:

ERROR!

Addition: 15

Subtraction: 5

Multiplication: 50

Division: 2

Division: Error: Division by zero

0

**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>

using namespace std;

class Beta;

class Alpha {

private:

int data;

public:

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

friend class Beta;

};

class Beta {

public:

void modifyAlpha(Alpha& alpha, int newData) {

alpha.data = newData;

}

void displayAlpha(const Alpha& alpha) {

cout << "Alpha's data: " << alpha.data << endl;

}

};

int main() {

Alpha objAlpha(100);

Beta objBeta;

objBeta.displayAlpha(objAlpha);

objBeta.modifyAlpha(objAlpha, 200);

objBeta.displayAlpha(objAlpha);

return 0;

}

OUTPUT:

Alpha's data: 100

Alpha's data: 200

**Class Templates with Multiple Parameters:**

**Write a class template Pair that can store a pair of values of any two data types. Include member functions to set and get the values. Demonstrate the usage of this template with different data types.**

#include <iostream>

using namespace std;

template<typename T1, typename T2>

class Pair {

private:

T1 first;

T2 second;

public:

Pair(const T1& f, const T2& s) : first(f), second(s) {}

void setPair(const T1& f, const T2& s) {

first = f;

second = s;

}

T1 getFirst() const {

return first;

}

T2 getSecond() const {

return second;

}

};

int main() {

Pair<int, int> intPair(10, 20);

cout << "First value: " << intPair.getFirst() << ", Second value: " << intPair.getSecond() << endl;

Pair<std::string, double> strDoublePair("Hello", 3.14);

cout << "First value: " << strDoublePair.getFirst() << ", Second value: " << strDoublePair.getSecond() << endl;

Pair<char, bool> charBoolPair('A', true);

cout << "First value: " << charBoolPair.getFirst() << ", Second value: " << charBoolPair.getSecond() << endl;

return 0;

}

OUTPUT:

First value: 10, Second value: 20

First value: Hello, Second value: 3.14

First value: A, Second value: 1