1. Create a program that simulates a zoo with various animals. Each animal should have a common method called "speak" that makes a sound specific to the animal type.

A: #include <iostream>

using namespace std;

class Animal {

public:

void speak() {

cout << "animal sound!" << endl; }

};

class Dog : public Animal {

public:

void speak() {

cout << "Woof! Woof!" << endl; }

};

class Cat : public Animal {

public:

void speak() {

cout << "Meow! Meow!" << endl; }

};

class Cow : public Animal {

public:

void speak() {

cout << "Moo! Moo!" << endl; }

};

int main() {

Dog d;

Cat c;

Cow C;

d.speak();

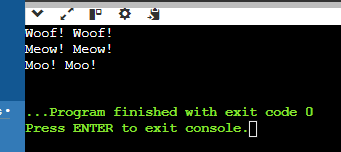
c.speak();

C.speak();

return 0;

}

OUTPUT:



2. Objective: Utilize runtime polymorphism to achieve the following:

Define an abstract base class named Animal with a method speak that doesn't have an implementation (declare it abstract).

Create subclasses for different animals like Lion, Elephant, etc., inheriting from Animal.

Override the speak method in each subclass to define the specific sound of the animal (e.g., Lion roars, Elephant trumpets).

In the main program, create an array of Animal references. Populate this array with objects of different animal subclasses.

Loop through the animal array and call the speak method on each reference. Since the references are of the base class type, runtime polymorphism will determine the actual subclass and invoke the appropriate overridden speak method.

#include <iostream>

using namespace std;

class Animal {

public:

virtual void speak() const = 0;

virtual ~Animal() = default;

};

class Dog : public Animal {

public:

void speak() const override {

cout << "Bark.." << endl;

}

};

class Cat : public Animal {

public:

void speak() const override {

cout << "Meow.." << endl;

}

};

class Lion : public Animal {

public:

void speak() const override {

cout << "Roar.." << endl;

}

};

class Elephant : public Animal {

public:

void speak() const override {

cout << "Trumpet.." << endl;

}

};

int main() {

const int numAnimals = 4;

Animal\* zoo[numAnimals];

zoo[0] = new Dog();

zoo[1] = new Cat();

zoo[2] = new Lion();

zoo[3] = new Elephant();

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

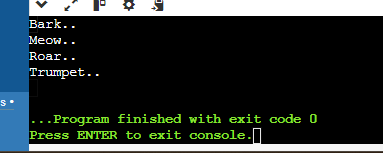
zoo[i]->speak();

}

return 0;

}

OUTPUT:



This exercise will demonstrate runtime polymorphism by:

Highlighting the separation between declared type (reference variable type) and actual type (object type).

Showing how the method call is resolved at runtime based on the actual object.

3. Write a code using Constructors and Destructors.

#include <iostream>

#include <string>

using namespace std;

class Person {

private:

string name;

public:

Person() : name("Unknown") {

cout << "Default constructor called" << endl; // Default constructor

}

Person(const string& name) : name(name) {

cout << "Parameterized constructor called for " << name << endl; // Parameterized constructor

}

Person(const Person& other) : name(other.name) {

cout << "Copy constructor called for " << name << endl; // Copy constructor

}

~Person() {

cout << "Destructor called " << name << endl; // Destructor

}

void display() const {

cout << name << endl;

}

};

int main() {

Person p1; // Using default constructor

p1.display();

Person p2("Harika"); // Using parameterized constructor

p2.display();

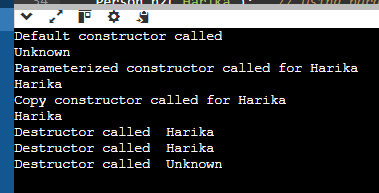
Person p3 = p2; // Using copy constructor

p3.display();

return 0;

}

OUTPUT:



4. You have a TemperatureSensor class that measures temperature in Celsius. You want a separate DisplayTemperature function to print the temperature in Fahrenheit. However, the conversion formula requires accessing the private celsius member. Create a TemperatureSensor class with a private celsius member and a public constructor. Implement a friend function DisplayTemperature that takes a TemperatureSensor object and prints the temperature in Fahrenheit (conversion formula provided). Write a main function to demonstrate how to use the classes.

#include <iostream>

using namespace std;

class TemperatureSensor {

private:

double celsius;

public:

TemperatureSensor(double tempCelsius) : celsius(tempCelsius) {}

friend void DisplayTemperature(const TemperatureSensor& sensor) {

double fahrenheit = sensor.celsius \* 9.0 / 5.0 + 32.0;

cout << "Temperature in Fahrenheit: " << fahrenheit << "°F" << endl;

}

};

int main() {

TemperatureSensor sensor(27.0);

DisplayTemperature(sensor);

return 0;

}

OUTPUT:



5. Friend Class for Stream Insertion: Scenario: You have a Point class with private members for x and y coordinates. You want to define a way to easily print Point objects to output streams like cout. Create a Point class with private x and y members and a public constructor. Design a friend class PointOutputStream that has an overloaded << operator to format and insert Point objects into output streams. In main, demonstrate creating Point objects and printing them using cout.

#include <iostream>

using namespace std;

class Point {

private:

int x, y;

public:

Point(int x, int y) : x(x), y(y) {}

friend ostream& operator<<(ostream& os, const Point& point) {

os << "(" << point.x << ", " << point.y << ")";

return os;

}

};

int main() {

Point p1(3, 4);

Point p2(7, 8);

cout << "Point 1: " << p1 << endl;

cout << "Point 2: " << p2 << endl;

return 0;

}

Output:

