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.

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.

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.

#include <iostream>

using namespace std;

// Abstract base class

class Animal {

public:

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

virtual ~Animal() = default; // Virtual destructor

};

// Subclass for Lion

class Lion : public Animal {

public:

void speak() override {

cout << "Lion: Roar!" << endl;

}

};

// Subclass for Elephant

class Elephant : public Animal {

public:

void speak() override {

cout << "Elephant: Trumpet!" << endl;

}

};

// Subclass for Dog

class Dog : public Animal {

public:

void speak() override {

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

}

};

// Subclass for Cat

class Cat : public Animal {

public:

void speak() override {

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

}

};

int main() {

// Creating instances of different animals

Lion lion;

Elephant elephant;

Dog dog;

Cat cat;

// Array of Animal pointers

Animal\* zoo[4] = { &lion, &elephant, &dog, &cat };

// Loop through the animal array and call the speak method

for (int i = 0; i < 4; ++i) {

zoo[i]->speak(); // Runtime polymorphism

}

return 0;

}

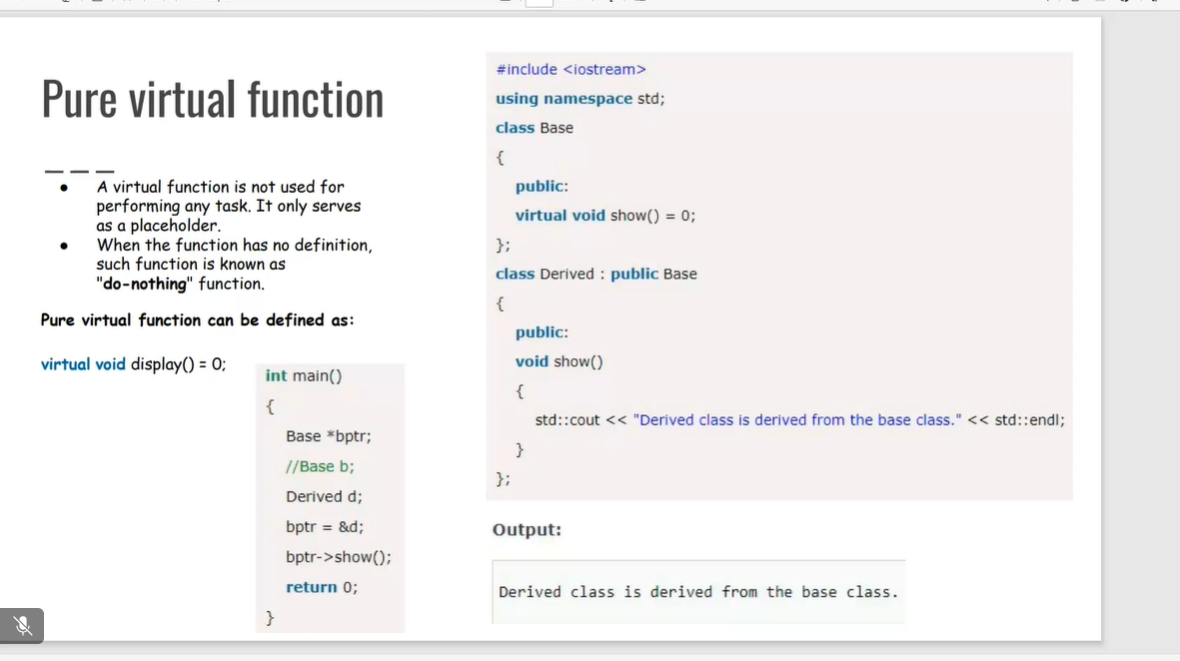
OUTPUT:

Lion: Roar

Elephant: Trumpet

Dog: Bark

Cat: Meow



DESTRUCTORS:

#include <iostream>

#include <cstring>

using namespace std;

class String {

private:

char\* s;

int size;

public:

String(char\* c);

~String();

void print() const;

};

String::String(char\* c) {

size = strlen(c);

s = new char[size + 1];

strcpy(s, c);

}

String::~String() {

delete[] s;

}

void String::print() const {

cout << s << endl;

}

int main() {

char input[] = "Hello World!";

String str(input);

str.print();

return 0;

}

OUTPUT:

Hello World!

**DESTRUCTORS**

**When is destructor called?**

A destructor function is called automatically when the object goes out of scope:

(1) the function ends

(2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

**How are destructors different from a normal member function?**

Destructors have same name as the class preceded by a tilde (~) Destructors don't take any argument and don't return anything

**Can there be more than one destructor in a class?**

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type

**When do we need to write a user-defined destructor?**

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class

contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

**Can a destructor be virtual?**

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.

**VIRTUAL DESTRUCOR:**

#include<iostream>

using namespace std;

class base {

public:

base()

{

cout<<"Constructing base \n"; }

~base()

{ cout<<"Destructing base \n"; }

};

class derived: public base {

public:

derived()

{ cout<<"Constructing derived \n"; }

~derived()

{ cout<<"Destructing derived \n"; }

};

int main(void)

{

derived\*d = new derived();

base\*b = d;

delete b;

getchar();

return 0;

}

OUTPUT:

Constructing base

Constructing derived

Destructing base

**write a simple code where we use all types of constructor and destructor**

#include <iostream>

using namespace std;

class class\_name{

// declaring private class data members

private:

int a,b;

public:

// declaring Constructor

class\_name(int aa, int bb)

{

cout<<"Constructor is called"<<endl;

a = aa;

b = bb;

cout<<"Value of a: "<<a<<endl;

cout<<"Value of b: "<<b<<endl;

cout<<endl;

}

// declaring destructor

~class\_name()

{

cout<<"Destructor is called"<<endl;

cout<<"Value of a: "<<a<<endl;

cout<<"Value of b: "<<b<<endl;

}

};

int main()

{

// creating objects of class using parameterized constructor

class\_name obj(5,6);

return 0;

}

OUTPUT:

Constructor is called

Value of a: 5

Value of b: 6

Destructor is called

Value of a: 5

Value of b: 6

**A Simple and complete C++ program to demonstrate friend class**

#include<iostream>

class A{

private:

int a;

public:

A(){a=0;}

friend class B;

};

class B{

private :

int b;

public:

void showA(A&x)

{

std::cout<<"A::a="<<x.a;

}

};

int main()

{

A a;

B b;

b.showA(a);

return 0;

}

OUPUT:

A::a=0

**A Simple and complete C++ program to demonstrate friend class function of another class**

#include<iostream>  
class B;

Class A{

Public:

Void showB(B&);

};

Class B{

Private:

Int b;

Public:

B(){b=0;}

Friend void A::showB(B&x);

};

Void A::show B(B&x)

{

Std::cout<<”B::b=”<<x.b;

}

Int main()

{

A a;

Bx;

a.showB(X);

Return 0;

}

OUTPUT:

B::b=0

**FRIEND CLASS AND FUNCTION**

Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.

3) Friendship is not inherited

4) The concept of friends is not there in Java.

#include<iostream>

class A {

private:

int a;

public:

A() { a = 0; }

int getA() const { return a; } // Getter method to access private member

};

class B {

private:

int b;

public:

void showA(const A& x) {

std::cout << "A::a = " << x.getA(); // Use getter method to access a

}

};

int main() {

A a;

B b;

b.showA(a);

return 0;

}

OUTPUT:

A::a=0

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 temp{

private:

float c;

public:

temp(float temp):c(temp) {}

friend void printtemp(const temp&);

};

void printtemp(const temp& sensor) {

float f =(sensor.c\*9.0/5.0)+32;

cout << "fahrenheit temperature: " << f << endl;

}

int main() {

temp sensor(67);

printtemp(sensor);

return 0;

}

OUTPUT:

fahrenheit temperature: 152.6