Cplusplus-1-07-24

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:

// Pure virtual function

virtual void speak() const = 0;

// Virtual destructor

virtual ~Animal() {}

};

// Lion class inheriting from Animal

class Lion : public Animal {

public:

// Override the speak method

void speak() const override {

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

}

};

// Elephant class inheriting from Animal

class Elephant : public Animal {

public:

// Override the speak method

void speak() const override {

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

}

};

int main() {

// Array of pointers to Animal objects

Animal\* zoo[2];

// Create Lion and Elephant objects and store their addresses in the array

zoo[0] = new Lion();

zoo[1] = new Elephant();

// Loop through the array and call the speak method

for (int i = 0; i < 2; ++i) {

zoo[i]->speak();

}

// Clean up the dynamically allocated memory

for (int i = 0; i < 2; ++i) {

delete zoo[i];

}

return 0;

}

2. #include <iostream>

using namespace std;

class Base

{

public:

virtual void show() = 0;

};

class Derived : public Base

{

public:

void show()

{

cout<<"Derived class is derived from the base class"<<endl;

}

};

int main()

{

Base \*bptr;

Derived d;

bptr = &d;

bptr->show();

return 0;

}

3. Desstructor:

#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;

}

Virtual destructor :

#include <iostream>

using namespace std;

class base {

public:

base() {

cout << "Constructing base \n";

}

virtual ~base() {

cout << "Destructing base \n";

}

};

class derived : public base {

public:

derived() {

cout << "Constructing derived \n";

}

~derived() {

cout << "Destructing derived \n";

}

};

int main() {

derived \*d = new derived();

base \*b = d;

delete b;

getchar();

return 0;

}

WRITE A CODE WHERE WE USE ALL TYPES OF CONSTRUCTORS AND DESTRUCTORS ?

#include <iostream>

using namespace std;

class Example {

public:

int value;

// Default Constructor

Example() {

value = 0;

cout << "Default Constructor called, value = " << value << endl;

}

// Parameterized Constructor

Example(int val) {

value = val;

cout << "Parameterized Constructor called, value = " << value << endl;

}

// Copy Constructor

Example(const Example &obj) {

value = obj.value;

cout << "Copy Constructor called, value = " << value << endl;

}

// Destructor

~Example() {

cout << "Destructor called, value = " << value << endl;

}

};

int main() {

// Using Default Constructor

Example obj1;

// Using Parameterized Constructor

Example obj2(10);

// Using Copy Constructor

Example obj3 = obj2;

return 0;

}

Friend function:

#include <iostream>

using namespace std;

class Example {

public:

int value;

// Default Constructor

Example() {

value = 0;

cout << "Default Constructor called, value = " << value << endl;

}

// Parameterized Constructor

Example(int val) {

value = val;

cout << "Parameterized Constructor called, value = " << value << endl;

}

// Copy Constructor

Example(const Example &obj) {

value = obj.value;

cout << "Copy Constructor called, value = " << value << endl;

}

// Destructor

~Example() {

cout << "Destructor called, value = " << value << endl;

}

};

int main() {

// Using Default Constructor

Example obj1;

// Using Parameterized Constructor

Example obj2(10);

// Using Copy Constructor

Example obj3 = obj2;

return 0;

}

Another friend class and function :

#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); //friend function

};

void A::showB(B& x)

{

std::cout<<"B::b=" <<x.b;

}

int main()

{

A a;

B x;

a.showB(x);

return 0;

}

2. #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;

}

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 temp) : celsius(temp) {}

friend void DisplayTemperature(const TemperatureSensor& sensor);

};

void DisplayTemperature(const TemperatureSensor& sensor) {

double fahrenheit = (sensor.celsius \* 9.0 / 5.0) + 32;

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

}

int main() {

TemperatureSensor sensor(30.0);

DisplayTemperature(sensor);

return 0;

}