**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) {

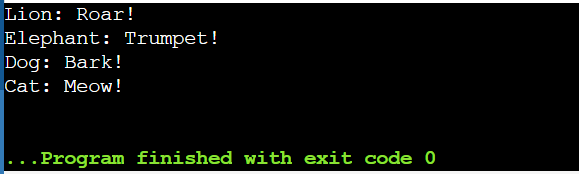
zoo[i]->speak(); // Runtime polymorphism

}

return 0;

}

**Output**

****

**Pure virtual function**

#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;

//Base b;

Derived d;

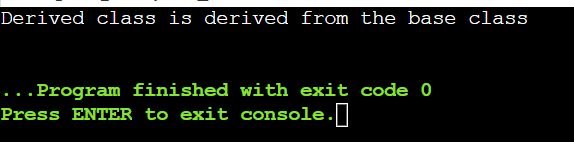
bptr=&d;

bptr->show();

return 0;

}

**Output**

****

**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;

cout << size << endl;

}

int main() {

char input[] = "Hello World!";

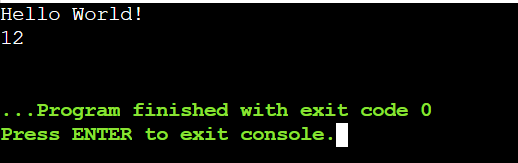
String str(input);

str.print();

return 0;

}

**Ouput**:



**Virtual Destructor**

#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;

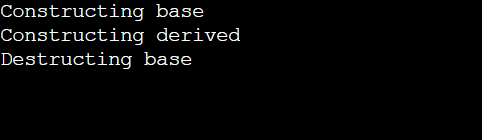
getchar();

delete b; // Properly delete the dynamically allocated object to call destructors

return 0;

}

**Output**:

****

#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(void) {

derived\* d = new derived();

base\* b = d;

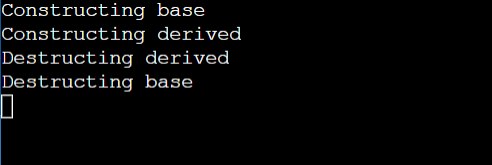
delete b;

getchar();

return 0;

}

**Output:**



**Write a code using all constructors and destructors**

#include <iostream>

#include <string>

using namespace std;

class Person {

private:

string name;

int age;

public:

Person() : name("Mahesh"), age(42)

{

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

}

Person(const string& name) : name(name), age(24)

{

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

}

Person(const Person& other) : name(other.name), age(other.age)

{

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

}

~Person() {

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

}

void display() const {

cout << name << endl;

cout << age << endl;

}

};

int main() {

Person p1; // Using default constructor

p1.display();

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

p2.display();

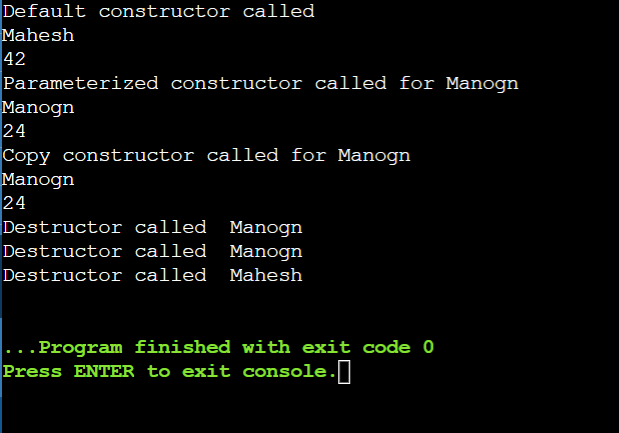
Person p3 = p2; // Using copy constructor

p3.display();

return 0;

}

**Output:**

****

**Friend function**

#include <iostream>

using namespace std;

class A {

private:

int a;

public:

A() {

a=0;

}

friend class B;

};

class B {

private:

int b;

public:

void showA(A&x)

{

//Since B is a friend of A, it can access

//private numbers of access

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

}

};

int main() {

A a;

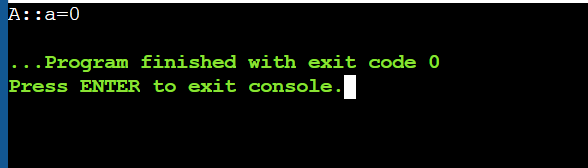
B b;

b.showA(a);

return 0;

}

**Output:**



**Friend Class and Function**

#include<iostream>

using namespace std;

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::showB(B& x){

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

}

int main()

{

A a;

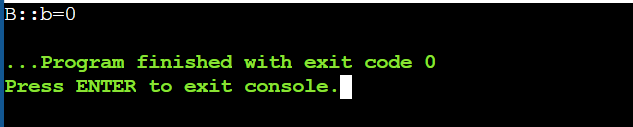
B x;

a.showB(x);

return 0;

}

**Output:**

****

**Temperature**

**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() {

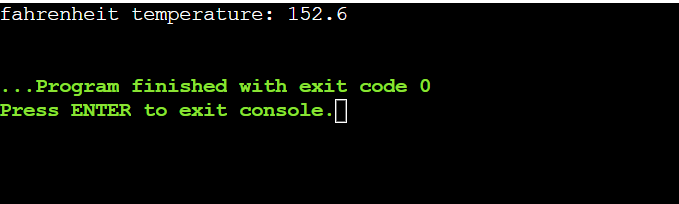
temp sensor(67);

printtemp(sensor);

return 0;

}

**Output:**

****

**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:

// Constructor

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

// Friend function to overload the << operator

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

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

return os;

}

};

int main() {

// Create Point objects

Point p1(3, 4);

Point p2(7, 8);

// Print Point objects using the overloaded << operator

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

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

return 0;

}

**Output:**

