**Runtime polymorphism:**

**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;

class Animal {

public:

virtual void speak() = 0;

virtual ~Animal() = default;

};

class Lion : public Animal {

public:

void speak()override {

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

}

};

class Elephant : public Animal {

public:

void speak()override {

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

}

};

class Dog : public Animal {

public:

void speak()override {

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

}

};

class Cat : public Animal {

public:

void speak() override {

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

}

};

int main() {

Lion lion;

Elephant elephant;

Dog dog;

Cat cat;

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

for (Animal\* animal : zoo) {

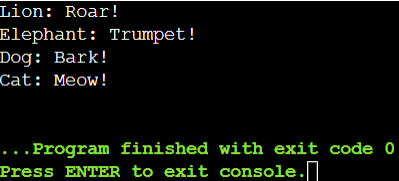
animal->speak();

}

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;

Derived d;

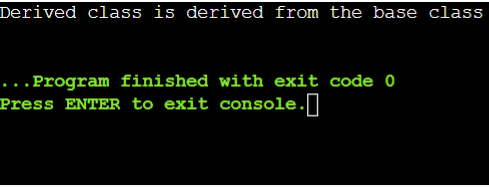
bptr = &d;

bptr->show();

return 0;

}

**Output:**



**Destructors:**

#include<iostream>

#include<cstring>

using namespace std;

class String{

private:

char\* s;

int size;

public:

String(char\*);

~ String();

void print();

};

String::String(char\* c)

{

size = strlen(c);

s = new char[size + 1];

strcpy(s,c);

}

String::~String(){

delete[] s;

}

void String::print(){

cout<<s<<endl;

}

int main(){

char input[]= "Hello World!";

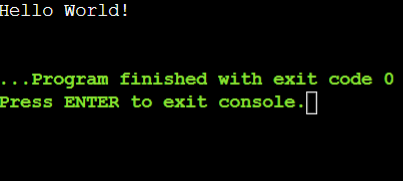
String str(input);

str.print();

return 0;

}

**Output:**



**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;

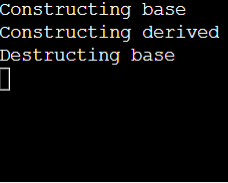
delete b;

getchar();

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;

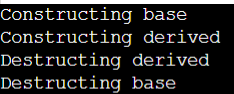
delete b;

getchar();

return 0;

}

**Output:**



**write a code where you use all types of constructor And destructor**

#include <iostream>

using namespace std;

class base{

public:

int x;

// Default constructor

base() {

x = 0;

cout << "Default constructing base\n";

}

// Parameterized constructor

base(int a) {

x = a;

cout << "Parameterized constructing base with value " << x << "\n";

}

// Copy constructor

base(base &a) {

x = a.x;

cout << "Copy constructing base with value " << x << "\n";

}

~base()

{

cout<<"Destructing base \n";

}

};

int main() {

base \*b1 = new base();

delete b1;

base \*b2 = new base(10);

delete b2;

base original(20);

base \*b3 = new base(original);

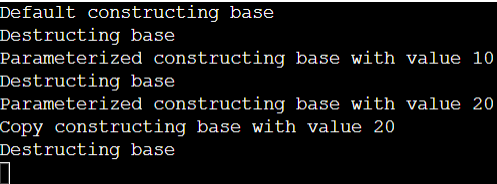
delete b3;

getchar();

return 0;

}

**Output:**



**Friend class and 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)

{

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

}

};

int main()

{

A a;

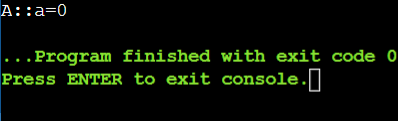
B b;

b.showA(a);

return 0;

}

**Output:**



#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;

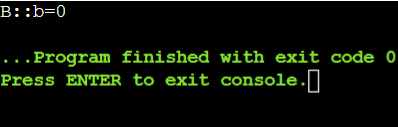
B x;

a.showB(x);

return 0;

}

**Output:**



**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() {

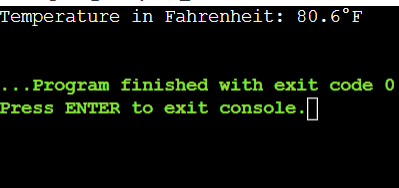
TemperatureSensor sensor(27.0);

DisplayTemperature(sensor);

return 0;

}

Output:



**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:**

