**DAY 6 :**

**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.**

#include <iostream>

using namespace std;

class Animal

{

public:

virtual void speak() const = 0;

virtual ~Animal() {}

};

class Lion : public Animal

{

public:

void speak() const override

{

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

}

};

class Elephant : public Animal

{

public:

void speak() const override

{

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

}

};

int main()

{

Animal\* zoo[2];

zoo[0] = new Lion();

zoo[1] = new Elephant();

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

zoo[i]->speak();

}

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

delete zoo[i];

}

return 0;

}

**OUTPUT :**

Lion: Roar

Elephant: Trumpet

**TASK 2:**

**Virtual Function**

#include <iostream>

using namespace std;

class Base

{

public: virtual void show()=0;

};

class Derived : public Base

{

public:void show()

{

std::cout<<"Derived class is derived from the Base class"<<std::endl;

}

};

int main()

{

Base\* ptr;

Derived d;

ptr = &d;

ptr->show();

return 0;

}

**OUTPUT :**

Derived class is derived from the Base class

**TASK 3 :**

**Destructor**

#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!

**Task 4:**

#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 d;

getchar();

return 0;

}

**Output:**

constructing base

constructing derived

destructing derived

destructing base

**Types of Constructor :**

#include <iostream>

using namespace std;

class Constructors

{

private: int value;

public: Constructors()

{

value = 0;

cout << "Default constructor value: " << value << endl;

}

Constructors(int v)

{

value = v;

cout << "Parameterized constructor value: " << value << endl;

}

Constructors(const Constructors& other)

{

value = other.value;

cout << "Copy constructor value: " << value << endl;

}

~Constructors()

{

cout << "Destructor value: " << value << endl;

}

void show()

{

cout << "Value: " << value << endl;

}

};

int main()

{

Constructors objA;

objA.show();

Constructors objB(12);

objB.show();

Constructors objC = objB;

objC.show();

return 0;

}

**OUTPUT :**

Default constructor value: 0

Value: 0

Parameterized constructor value: 12

Value: 12

Copy constructor value: 12

Value: 12

Destructor value: 12

Destructor value: 12

Destructor value: 0

**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)

{

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

}

};

int main()

{

A a;

B b;

b.showA(a);

return 0;

}

**OUTPUT:**

A::a=0

**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)

{

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

}

int main()

{

A a;

B x;

a.showB(x);

return 0;

}

**OUTPUT:**

**B::b = 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;

}

**OUTPUT :**

**Temperature in Fahrenheit: 86**

**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 to initialize the coordinates

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

// Declare the friend function for the << operator

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

};

// Definition of the overloaded << operator

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

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

return os;

}

int main() {

// Create Point objects

Point p1(3, 4);

Point p2(7, 8);

// Print Point objects using cout

cout << p1 << endl;

cout << p2 << endl;

return 0;

}

**OUTPUT:**

**Point(3, 4)**

**Point(7, 8)**