**DAY – 6 Assignment and LABs**

1. **Assignment-1 Time - 9:05 a.m**

**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() const = 0; // Pure virtual function (abstract method)

};

class Lion : public Animal {

public:

void speak() const override {

cout << "Lion roars" << endl;

}

};

class Elephant : public Animal {

public:

void speak() const override {

cout << "Elephant trumpets"<<endl;

}

};

int main() {

// Array of Animal references

Animal\* animals[2];

// Populate with Lion and Elephant objects (runtime polymorphism)

animals[0] = new Lion;

animals[1] = new Elephant;

// Loop through animals and call speak through the base class reference

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

animals[i]->speak();

}

// Deallocate memory

delete animals[0];

delete animals[1];

return 0;

}

1. **Code 2 Pure Virtual Function**

class Base {

public:

virtual void show() {

cout << "Base class show function" << endl;

}

virtual void display() {

cout << "Base class display function" << endl;

}

};

class Derived : public Base {

public:

void show() override { // Correctly overriding Base::show

cout << "Derived class show function" << endl;

}

// The following line would cause a compile-time error if uncommented,

// because there's no function with this signature in the base class.

// void display(int x) override { }

};

int main() {

Base\* b = new Derived();

b->show(); // Calls Derived's show

b->display(); // Calls Base's display

delete b;

return 0;

}

1. **Code -3 Run Time polymorphism**

class Animal {

public:

virtual void makeSound() const { // Virtual function

cout << "Some generic animal sound" << endl;

}

};

class Dog : public Animal {

public:

void makeSound() const override { // Override base class function

cout << "Woof!" << endl;

}

};

class Cat : public Animal {

public:

void makeSound() const override { // Override base class function

cout << "Meow!" << endl;

}

};

void makeAnimalSound(const Animal& animal) {

animal.makeSound(); // Calls the appropriate makeSound function

}

int main() {

Dog dog;

Cat cat;

Animal\* animals[2];

animals[0] = &dog;

animals[1] = &cat;

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

animals[i]->makeSound(); // Runtime polymorphism in action

}

return 0;

}

1. **Code - 4 Pure Virtual function**

class Base {

public:

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

};

class Derived : public Base {

public:

void show() override { // Override the base class pure virtual function

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

}

};

int main() {

Base\* bptr;

Derived d;

bptr = &d; // bptr points to the Derived object

bptr->show(); // Calls Derived's show function

return 0;

}

1. **Code - 5 Constructor and Destructor using String properties**

#include <cstring> // For strlen, strcpy functions

class String {

private:

char\* s;

int size;

public:

// Constructor

String(char\* c) {

size = strlen(c);

s = new char[size + 1]; // Allocate memory for string and null terminator

strcpy(s, c); // Copy input string into allocated memory

}

// Destructor

~String() {

delete[] s; // Deallocate memory for string

}

// Function to display the string

void display() {

cout << "String: " << s << endl;

}

};

int main() {

// Example usage of String class

String str("Hello, World!");

str.display();

return 0;

}

1. **Code -6 Based on Constructor and Destructor ( Without using Virtual keyword)**

class base {

public:

base() {

cout << "**Constructor of base class\n**";

}

~base() {

cout << "**Destructor of base class\n**";

}

};

class derived : public base {

public:

derived() {

cout << "**Constructor of derived class\n**";

}

~derived() {

cout << "**Destructor of derived class\n**";

}

};

int main(void) {

derived\* d = new derived(); // Creates a derived object 'd'

base\* b = d; // Assigns the derived pointer 'd' to a base pointer 'b'

delete b; // Deletes the base pointer (polymorphic deletion)

//getchar(); // Waits for user input before exiting

return 0;

}

1. **Code -7 Based on Constructor and Destructor ( Using Virtual keyword)**

class base {

public:

base() {

cout << "Constructor of base class\n";

}

virtual ~base() {

cout << "Destructor of base class\n"; // It ensures that when an object is

// deleted through a pointer to a base class,

// the destructor of the derived class is called

// first, followed by the destructor of the base

// class.

}

};

class derived : public base {

public:

derived() {

cout << "Constructor of derived class\n";

}

~derived() {

cout << "Destructor of derived class\n";

}

};

int main(void) {

derived\* d = new derived(); // Creates a derived object 'd'

base\* b = d; // Assigns the derived pointer 'd' to a base pointer 'b'

delete b; // Deletes the base pointer (polymorphic deletion)

getchar(); // Waits for user input before exiting

return 0;

}

1. **Code- 8 Use of Multiple Constructor and Destructor**

class MyClass {

private:

int value;

public:

// Default constructor

MyClass() {

value = 0;

cout << "Default constructor called. Value: " << value << endl;

}

// Parameterized constructor

MyClass(int v) {

value = v;

cout << "Parameterized constructor called. Value: " << value << endl;

}

// Copy constructor

MyClass(const MyClass &obj) {

value = obj.value;

cout << "Copy constructor called. Value: " << value << endl;

}

// Destructor

~MyClass() {

cout << "Destructor called. Value: " << value << endl;

}

// Function to display the value

void display() {

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

}

};

int main() {

// Using default constructor

MyClass obj1;

obj1.display();

// Using parameterized constructor

MyClass obj2(100);

obj2.display();

// Using copy constructor

MyClass obj3 = obj2;

obj3.display();

// Explicitly calling the destructor (usually not done in practice)

obj1.~MyClass();

return 0;

}

1. **Code -9 Friend Function**

class Test {

private:

int value;

public:

// Constructor

Test(int num) : value(num) {}

// Friend function declaration

friend void displayValue(const Test& obj);

};

// Friend function definition

void displayValue(const Test& obj) {

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

}

int main() {

Test obj(42);

// Calling the friend function

displayValue(obj);

return 0;

}

1. **Code – 10(2) Friend Function ( Assignment - 2) to access all private and protected members of Base class .**

class A{

private:

int a;

public:

A( ) { // Default Constructor intilizes 'a' to 20

a = 20;

}

//friend class B;// Friend Class can acces all private and protected members of class A.

class B;

};

class B {

private:

int b;

public:

void showA(A& x)

{

// Since B is friend of A it can access

// private member of access

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

}

};

int main()

{

A a;

B b;

b.showA(a);

return 0;

}

1. **Code - 11 (3) Friend Function ( Assignment - 2)**

class B ; // Forward declaration

class A {

public :

void showB(B&); // Memeber function declaration

};

class B{

private:

int b;

public:

B(){ // Default Constructor

b = 0 ;

}

friend void A::showB(B& x); // Friend function

};

// Definition of the showB function

void A::showB(B& x) {

// Accessing private member of class B

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

}

int main() {

A a; // Create an object of class A

B x; // Create an object of class B

a.showB(x); // Call the showB function to display the private member of B

return 0;

}

1. **Assignment - 3 Time 4:30 p.m**

**/\* Objective - 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.**

class TemperatureSensor {

private:

double celsius;

public:

// Constructor

TemperatureSensor(double temp) : celsius(temp) {}

// Friend function declaration

friend void DisplayTemperature(TemperatureSensor& sensor);

};

void DisplayTemperature(TemperatureSensor& sensor) { // // Friend function definition

double fahrenheit = (sensor.celsius \* 9 / 5) + 32;

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

}

int main() {

// Create a TemperatureSensor object

TemperatureSensor sensor(60); // 60 degrees Celsius

DisplayTemperature(sensor);

return 0;

}

1. **Assignment - 4 Time 5:10 p.m**

**Objective - 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.**

class Point {

private:

int x;

int y;

public:

// Constructor

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

// Friend class declaration

friend class PointOutputStream;

};

class PointOutputStream {

public:

// Overloaded << operator for Point objects

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(-1, 7);

// Print Point objects using overloaded <<

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

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

return 0;

}