DAY 5

Question-14 : Problem Statement: You are designing a simplified version of a social media post management system. In this system, a Post class represents a user's post, which includes attributes such as the post's text content, a dynamically allocated array of reactions (likes, loves, etc.), and the number of reactions. Your goal is to demonstrate the use of various types of constructors within this context:

Default Constructor: Initializes a new post with default values.

Parameterized Constructor: Allows creating a post with specified content and a predefined size for the reactions array.

Copy Constructor: Enables creating a new post as a copy of an existing post.

Move Constructor: Facilitates transferring the ownership of resources from a temporary post object to a new post object efficiently.

sol : #include <iostream>

#include <string>

enum class Reaction {

LIKE,

LOVE,

HAHA,

WOW,

SAD,

ANGRY

};

class Post {

private:

std::string content;

Reaction\* reactions;

int numReactions;

public:

Post() : content(""), reactions(nullptr), numReactions(0) {

std::cout << "Default Constructor called" << std::endl;

}

Post(const std::string& \_content, int \_numReactions) : content(\_content), numReactions(\_numReactions) {

reactions = new Reaction[\_numReactions];

std::cout << "Parameterized Constructor called" << std::endl;

}

Post(const Post& other) : content(other.content), numReactions(other.numReactions) {

reactions = new Reaction[numReactions];

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

reactions[i] = other.reactions[i];

}

std::cout << "Copy Constructor called" << std::endl;

}

Post(Post&& other) noexcept : content(std::move(other.content)), reactions(other.reactions), numReactions(other.numReactions) {

other.reactions = nullptr;

other.numReactions = 0;

std::cout << "Move Constructor called" << std::endl;

}

~Post() {

delete[] reactions;

std::cout << "Destructor called" << std::endl;

}

};

int main() {

Post post1;

Post post2("Hello World!", 5);

Post post3 = post2;

Post post4 = std::move(post3);

return 0;

}

O/P : Default Constructor called

Parameterized Constructor called

Copy Constructor called

Move Constructor called

Destructor called

Destructor called

Destructor called

Destructor called

Question-15 : Problem Statement: You are tasked with developing a part of a vehicle fleet management system for a logistics company. This system should track various types of vehicles, including cars, trucks, and motorcycles. Each vehicle type has different characteristics and behaviors. Utilize C++ class concepts such as inheritance, polymorphism, encapsulation, and abstraction to design this system.

Objective:

Base Class - Vehicle: Create an abstract base class Vehicle that includes common attributes such as make, model, and year, and a pure virtual function displayDetails() to print vehicle details.

Derived Classes - Car, Truck, Motorcycle: Implement these classes to inherit from Vehicle. Each class should override displayDetails() to include specific attributes. For example, Truck might include cargoCapacity, Car could include passengerCapacity, and Motorcycle might have hasSidecar.

Utilize Polymorphism: Demonstrate how a collection of vehicles can be managed and iterated over to display details of each vehicle, regardless of its specific type.

Encapsulation: Ensure that class attributes are appropriately protected and provide public methods to access and modify them safely.

Sol : #include <iostream>

#include <string>

#include <vector>

class Vehicle {

protected:

std::string make;

std::string model;

int year;

public:

Vehicle(const std::string& \_make, const std::string& \_model, int \_year)

: make(\_make), model(\_model), year(\_year) {}

virtual void displayDetails() const = 0;

};

class Car : public Vehicle {

private:

int passengerCapacity;

public:

Car(const std::string& \_make, const std::string& \_model, int \_year, int \_passengerCapacity)

: Vehicle(\_make, \_model, \_year), passengerCapacity(\_passengerCapacity) {}

void displayDetails() const override {

std::cout << "Car: " << make << " " << model << ", Year: " << year << ", Passenger Capacity: " << passengerCapacity << std::endl;

}

};

class Truck : public Vehicle {

private:

int cargoCapacity;

public:

Truck(const std::string& \_make, const std::string& \_model, int \_year, int \_cargoCapacity)

: Vehicle(\_make, \_model, \_year), cargoCapacity(\_cargoCapacity) {}

void displayDetails() const override {

std::cout << "Truck: " << make << " " << model << ", Year: " << year << ", Cargo Capacity: " << cargoCapacity << " tons" << std::endl;

}

};

class Motorcycle : public Vehicle {

private:

bool hasSidecar;

public:

Motorcycle(const std::string& \_make, const std::string& \_model, int \_year, bool \_hasSidecar)

: Vehicle(\_make, \_model, \_year), hasSidecar(\_hasSidecar) {}

void displayDetails() const override {

std::cout << "Motorcycle: " << make << " " << model << ", Year: " << year;

if (hasSidecar)

std::cout << ", Has Sidecar";

else

std::cout << ", No Sidecar";

std::cout << std::endl;

}

};

int main() {

std::vector<Vehicle\*> fleet;

fleet.push\_back(new Car("Toyota", "Camry", 2022, 5));

fleet.push\_back(new Truck("Ford", "F-150", 2021, 2));

fleet.push\_back(new Motorcycle("Harley-Davidson", "Street Glide", 2020, false));

for (const auto& vehicle : fleet) {

vehicle->displayDetails();

}

for (const auto& vehicle : fleet) {

delete vehicle;

}

return 0;

}

O/P : Car: Toyota Camry, Year: 2022, Passenger Capacity: 5

Truck: Ford F-150, Year: 2021, Cargo Capacity: 2 tons

Motorcycle: Harley-Davidson Street Glide, Year: 2020, No Sidecar

Question-16 : 16. Scenario: Design a basic banking application where a base class Account holds common attributes like account number and balance. The derived class SavingsAccount extends Account with an interest rate attribute. Implement a method to calculate interest in SavingsAccount.

Question: How would you design the Account and SavingsAccount classes to demonstrate simple inheritance, ensuring that SavingsAccount inherits from Account and adds its unique functionality?

Sol : #include <iostream>

class Account {

protected:

int accountNumber;

double balance;

public:

Account(int \_accountNumber, double \_balance) : accountNumber(\_accountNumber), balance(\_balance) {}

void deposit(double amount) {

balance += amount;

}

void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

} else {

std::cout << "Insufficient balance" << std::endl;

}

}

void display() const {

std::cout << "Account Number: " << accountNumber << ", Balance: $" << balance << std::endl;

}

};

class SavingsAccount : public Account {

private:

double interestRate;

public:

SavingsAccount(int \_accountNumber, double \_balance, double \_interestRate)

: Account(\_accountNumber, \_balance), interestRate(\_interestRate) {}

double calculateInterest() const {

return balance \* interestRate / 100.0;

}

void display() const {

std::cout << "Account Number: " << accountNumber << ", Balance: $" << balance << ", Interest Rate: " << interestRate << "%" << std::endl;

}

};

int main() {

SavingsAccount savingsAccount(123456, 1000.0, 3.5);

savingsAccount.display();

savingsAccount.deposit(500.0);

savingsAccount.withdraw(200.0);

savingsAccount.display();

double interest = savingsAccount.calculateInterest();

std::cout << "Interest earned: $" << interest << std::endl;

return 0;

}

O/P : Account Number: 123456, Balance: $1000, Interest Rate: 3.5%

Account Number: 123456, Balance: $1300, Interest Rate: 3.5%

Interest earned: $45.5