TASK -1

The Social Media Class Hierarchy:

1. Access Control and Getters:

Create the User class with private members for username and profile picture (string).

Implement public member functions for the constructor and getters (accessor methods) for username and profile picture.

A: #include <iostream>

#include <string>

using namespace std;

class User {

private:

string username;

string profilePicture;

public:

User(const string& un, const string& ppic) // Constructor

: username(un), profilePicture(ppic) {}

string getUsername() const { // Get Username

return username;

}

string getProfilePicture() const { //Get ProfilePicture

return profilePicture;

}

};

int main() {

string un;

string ppic;

cout << "Enter username: "; //Inputs for username

getline(cin, un);

cout << "Enter profile picture filename: "; //Inputs for profile picture

getline(cin, ppic);

User user(un, ppic); //User Object

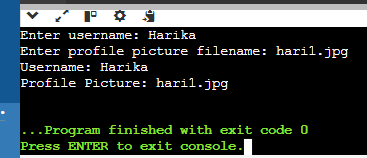
cout << "Username: " << user.getUsername() << endl; //Methods to implement it

cout << "Profile Picture: " << user.getProfilePicture() << endl;

return 0;

}

OUTPUT:



2. Post Class and Display:

Create the derived class Post inheriting from User.

Add private members for post content (string) and timestamp (date/time format of your choice).

Implement a public member function getPostInfo that returns a formatted string containing username, profile picture, post content, and timestamp.

3. Basic Interaction Function:

Define a friend function basicInteract that takes two User objects (or derived class objects) as arguments.

Inside the function, simply print a generic message like "User1 interacts with User2."

A: #include <iostream>

using namespace std;

class User {

private:

string name;

public:

User(const string& name) : name(name) {}

friend void basicInteract(const User& user1, const User& user2);

};

void basicInteract(const User& user1, const User& user2) {

cout << user1.name << " interacts with " << user2.name << endl;

}

int main() {

User user1("harry");

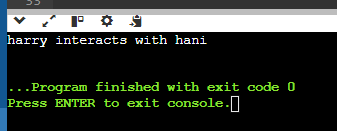
User user2("hani");

basicInteract(user1, user2);

return 0;

}

OUTPUT:



4. Overloaded Interact Functions:

Create overloaded versions of the interact function:

likePost(User& user, Post& post): This function should print a message indicating the user liked the post.

followUser(User& follower, User& followed): This function should print a message indicating the user started following another user.

A: #include <iostream>

#include <string>

using namespace std;

class Post; // Forward declaration

class User {

private:

string name;

public:

User(const string& name) : name(name) {}

friend void interact(User& user, Post& post); //Functions of Friends

friend void interact(User& follower, User& followed);

};

class Post {

private:

string content;

public:

Post(const string& content) : content(content) {}

friend void interact(User& user, Post& post);

};

void interact(User& user, Post& post) { // Overloaded functions implementation

cout << user.name << " liked the post: " << post.content << endl;

}

void interact(User& follower, User& followed) {

cout << follower.name << " started following " << followed.name << endl;

}

int main() {

User user1("Harry");

User user2("Hani");

Post post("Hello World!");

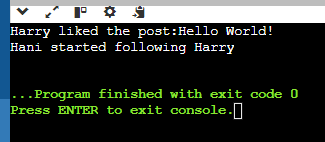
interact(user1, post); // User liking a post

interact(user2, user1); // User2 following User1

return 0;

}

OUTPUT:



5. Refactoring with Encapsulation:

Revisit the class design. Can you modify the code to reduce reliance on friend functions?

Consider adding public member functions or accessor methods within the User class to provide controlled access to relevant data instead of exposing everything through friend functions.

Bonus Challenge:

Implement a way to store and manage friend connections within the class hierarchy. You could explore a separate Friendship class or a boolean flag within User to track friend status. Modify the interact functions to incorporate this information and display more relevant messages based on the relationship between users.

#include <iostream>

#include <string>

using namespace std;

class User {

private:

string username;

bool isFriend;

User\* friendUser;

public:

User(const string& name) : username(name), isFriend(false), friendUser(nullptr) {}

const string& getUsername() const {

return username;

} bool addFriend(User\* user) {

if (user == nullptr || user == this || isFriend) {

return false;

}

if (friendUser == nullptr) {

friendUser = user;

user->friendUser = this; // Bidirectional friendship

isFriend = true;

return true;

}

return false; // Already has a friend

} bool removeFriend() {

if (friendUser != nullptr) {

friendUser->friendUser = nullptr;

friendUser = nullptr;

isFriend = false;

return true;

}

return false;

} void displayFriend() const {

if (friendUser != nullptr) {

cout << "Friend of " << username << ": " << friendUser->getUsername() << endl;

} else {

cout << username << " has no friend." << endl;

}

}

void interact(User\* other) const {

if (other == nullptr) {

cout << "Invalid interaction." << endl;

return;

}

if (other == this) {

cout << "You cannot interact with yourself." << endl;

return;

}

if (friendUser == other) {

cout << "You and " << other->getUsername() << " are friends." << endl;

} else {

cout << "You and " << other->getUsername() << " are not friends." << endl;

}

}

}; int main() {

User harry("Harry");

User mintu("Mintu");

User rita("Rita");

harry.addFriend(&mintu);

mintu.addFriend(&rita);

harry.displayFriend();

mintu.displayFriend();

harry.interact(&mintu);

harry.interact(&rita);

rita.interact(&mintu);

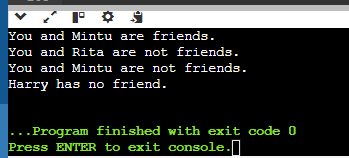
harry.removeFriend();

harry.displayFriend();

return 0;

}

OUTPUT:



------------------------------------------------------------------------------------------------------------------------------

#include<iostream>

using namespace std;

class MyClass {

private:

static int counter;

int count;

public:

MyClass(){

count++;

counter++;

}

static int getCounter() {

return counter;

}

int getCount() {

return count;

}

};

int MyClass::counter = 0;

int main() {

MyClass obj1;

MyClass obj2;

MyClass obj3;

cout<<"Number of objects created:"<<MyClass::getCounter()<<endl;

cout << "objects1 count method: " << obj1.getCount()<<endl;

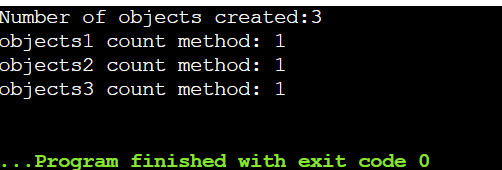
cout << "objects2 count method: " << obj2.getCount()<<endl;

cout << "objects3 count method: " << obj3.getCount()<<endl;

return 0;

}

OUTPUT:



-----------------------------------------------------------------------------------------------------------------------------------------

TASK – 2

* Distance Converter:

Create a class named DistanceConverter. Include the following static methods:

convertMilesToKm(double miles): Converts miles to kilometers (1 mile = 1.60934 kilometers).

convertKmToMiles(double kilometers): Converts kilometers to miles. In your main function, prompt the user for a distance and a unit (miles or kilometers). Use the appropriate static method from the DistanceConverter class to perform the conversion and display the result to the user.

A: #include <iostream>

#include <string>

using namespace std;

double convertMilesToKm(double miles) {

return miles \* 1.60934;

}

double convertKmToMiles(double kilometers) {

return kilometers / 1.60934;

}

int main() {

double distance = 100.0;

string unit = "miles";

if (unit == "miles") {

double km = convertMilesToKm(distance);

cout << distance << " miles is equal to " << km << " kilometers." << endl;

} else if (unit == "kilometers") {

double miles = convertKmToMiles(distance);

cout << distance << " kilometers is equal to " << miles << " miles." << endl;

} else {

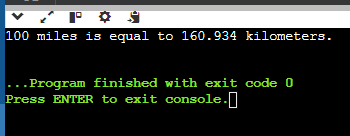
cout << "Invalid unit entered. Please enter miles or kilometers." << endl;

}

return 0;

}

OUTPUT:



* Math Utility Class:

Design a class named MathUtil. Include static methods for basic mathematical operations:

add(int a, int b): Adds two integers.

subtract(int a, int b): Subtracts two integers.

multiply(int a, int b): Multiplies two integers.

divide(int a, int b) (optional): Divides two integers with error handling for division by zero. In your main function, prompt the user for two numbers and an operation (+, -, \*, or /). Use the corresponding static method from the MathUtil class to perform the calculation and display the result.

#include <iostream>

#include <stdexcept>

using namespace std;

class MathUtil {

public:

statIc int add(int a, int b) { // Add two integers

return a + b;

}

static int subtract(int a, int b) { //Subtract two integers

return a - b;

}

static int multiply(int a, int b) { // Static method to multiply two integers

return a \* b;

}

static int divide(int a, int b) {

if (b == 0) {

throw invalid\_argument("Division by zero error");

}

return a / b;

}

};

int main() {

int num1 = 20;

int num2 = 4;

char operation = '/';

try {

switch (operation) {

case '+':

cout << "Result: " << MathUtil::add(num1, num2) << endl;

break;

case '-':

cout << "Result: " << MathUtil::subtract(num1, num2) << endl;

break;

case '\*':

cout << "Result: " << MathUtil::multiply(num1, num2) << endl;

break;

case '/':

cout << "Result: " << MathUtil::divide(num1, num2) << endl;

break;

default:

cout << "Invalid operation entered." << endl;

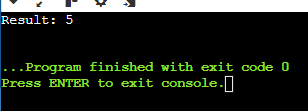
}

} catch (const invalid\_argument& e) { }

return 0;

}

OUTPUT:



* Simple Currency Converter:

Create a class named CurrencyConverter. Define a static variable named exchangeRate (e.g., USD to EUR exchange rate). Implement static methods: convertToEur(double amount): Converts an amount from the base currency (USD) to EUR based on the exchange rate. convertFromEur(double amount): Converts an amount from EUR to the base currency (USD). In your main function, prompt the user for an amount and a conversion direction (USD to EUR or EUR to USD). Use the appropriate static method from the CurrencyConverter class to perform the conversion and display the result.

A: #include <iostream>

using namespace std;

class CurrencyConverter {

public:

static double exchangeRate; // Static variable for the exchange rate

static double convertToEur(double amount) {

return amount \* exchangeRate; // Static method to convert USD to EUR

}

static double convertFromEur(double amount) {

return amount / exchangeRate; // Static method to convert EUR to USD

}

};

double CurrencyConverter::exchangeRate = 0.85; // Example exchange rate: 1 USD = 0.85 EUR

int main() {

double amount;

char direction;

cout << "Enter the amount: ";

cin >> amount;

cout << "Enter the conversion direction (U for USD to EUR, E for EUR to USD): ";

cin >> direction;

if (direction == 'U' || direction == 'u') {

double result = CurrencyConverter::convertToEur(amount);

cout << amount << " USD is " << result << " EUR.\n";

} else if (direction == 'E' || direction == 'e') {

double result = CurrencyConverter::convertFromEur(amount);

cout << amount << " EUR is " << result << " USD.\n";

} else {

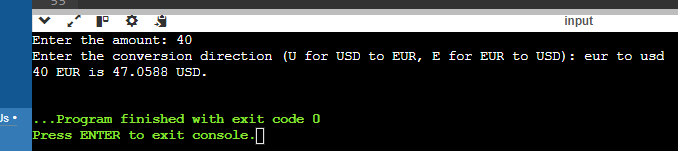
cout << "Invalid conversion direction.\n";

}

return 0;

}

OUTPUT:



-------------------------------------------------------------------------------------------------------------------------------

TASK – 3

* Design a function template named compare that takes two arguments of the same type and returns a boolean value indicating whether the first argument is greater than, less than, or equal to the second argument. How would you adapt this template to work with custom data types?

#include <iostream>

using namespace std;

template <typename T>

bool compare(const T& a, const T& b) {

if (a > b) {

cout << a << " is greater than " << b << endl;

return true;

} else if (a < b) {

cout << a << " is less than " << b << endl;

return false;

} else {

cout << a << " is equal to " << b << endl;

return false;

}

} int main() {

int a = 5, b = 10;

compare(a, b);

int c = 20, d = 15;

compare(c, d);

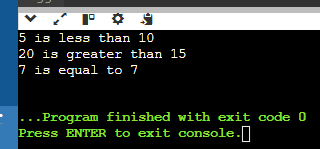
int e = 7, f = 7;

compare(e, f);

return 0;

}

OUTPUT:



* Implement a function template named swap that exchanges the values of two variables of the same type. Discuss the potential limitations of this approach when dealing with complex data structures.

#include <iostream>

using namespace std;

template <typename T>

void mySwap(T& a, T& b) {

T temp = a;

a = b;

b = temp;

} int main() {

int x = 5, y = 10;

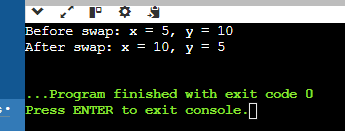
cout << "Before swap: x = " << x << ", y = " << y << endl;

mySwap(x, y);

cout << "After swap: x = " << x << ", y = " << y << endl;

return 0;

}

OUTPUT:  


* Consider a scenario where you need to find the minimum value in an array. Create a function template named findMin that works with any data type for which the comparison operator (<) is defined. Explain how function templates promote code reusability in this case.

#include <iostream>

using namespace std;

template <typename T>

T findMin(const T\* array, int size) {

T minVal = array[0];

for (int i = 1; i < size; ++i) {

if (array[i] < minVal) {

minVal = array[i];

}

}

return minVal;

}

int main() {

int intArray[] = {5, 2, 9, 1, 5, 6};

int intMin = findMin(intArray, 6);

cout << "Minimum value in intArray: " << intMin << endl;

return 0;

}

OUTPUT:  
