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.

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."

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.

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>

#include <vector>

class User {

private:

std::string username;

std::string profilePicture;

public:

// Constructor

User(const std::string& uname, const std::string& ppic)

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

// Accessors

std::string getUsername() const {

return username;

}

std::string getProfilePicture() const {

return profilePicture;

}

};

#include <ctime>

class Post : public User {

private:

std::string content;

std::string timestamp;

public:

// Constructor

Post(const std::string& uname, const std::string& ppic, const std::string& cont, const std::string& time)

: User(uname, ppic), content(cont), timestamp(time) {}

// Accessor

std::string getPostInfo() const {

return "Username: " + getUsername() + "\nProfile Picture: " + getProfilePicture() +

"\nContent: " + content + "\nTimestamp: " + timestamp;

}

};

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

std::cout << user1.getUsername() << " interacts with " << user2.getUsername() << std::endl;

}

void likePost(User& user, Post& post) {

std::cout << user.getUsername() << " liked the post by " << post.getUsername() << std::endl;

}

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

std::cout << follower.getUsername() << " started following " << followed.getUsername() << std::endl;

}

class User {

private:

std::string username;

std::string profilePicture;

public:

// Constructor

User(const std::string& uname, const std::string& ppic)

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

// Accessors

std::string getUsername() const {

return username;

}

std::string getProfilePicture() const {

return profilePicture;

}

// Member functions for interactions

void like(Post& post) const {

std::cout << username << " liked the post by " << post.getUsername() << std::endl;

}

void follow(User& user) const {

std::cout << username << " started following " << user.getUsername() << std::endl;

}

};

BONUS:

class User {

private:

std::string username;

std::string profilePicture;

std::vector<User\*> friends;

public:

// Constructor

User(const std::string& uname, const std::string& ppic)

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

// Accessors

std::string getUsername() const {

return username;

}

std::string getProfilePicture() const {

return profilePicture;

}

// Member functions for interactions

void like(Post& post) const {

std::cout << username << " liked the post by " << post.getUsername() << std::endl;

}

void follow(User& user) {

friends.push\_back(&user);

std::cout << username << " started following " << user.getUsername() << std::endl;

}

bool isFriend(const User& user) const {

for (const auto& friendUser : friends) {

if (friendUser->getUsername() == user.getUsername()) {

return true;

}

}

return false;

}

};

void enhancedInteract(User& user1, User& user2) {

if (user1.isFriend(user2)) {

std::cout << user1.getUsername() << " is friends with " << user2.getUsername() << std::endl;

} else {

std::cout << user1.getUsername() << " is not friends with " << user2.getUsername() << std::endl;

}

}

int main() {

User user1("Alice", "alice.jpg");

User user2("Bob", "bob.jpg");

Post post("Bob", "bob.jpg", "Hello, world!", "2024-07-02 12:00");

basicInteract(user1, user2);

user1.like(post);

user1.follow(user2);

enhancedInteract(user1, user2);

return 0;

}

STATIC MEMBERS:

#include <iostream>

class myclass{

private:

static int counter;

public:

myclass(){

counter++;

}

static int getcount(){

return counter;

}

};

int myclass::counter=0;

int main(){

myclass obj1;

myclass obj2;

myclass obj3;

std::cout<<"number of objects created:"<<myclass::getcount()<<std::endl;

return 0;

}

COUNTER :

#include <iostream>

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;

std::cout<<"number of objects created:"<<myclass::getcounter()<<std::endl;

std::cout<<"onjects1 count method:"<<obj1.getcount()<<std::endl;

std::cout<<"onjects2 count method:"<<obj2.getcount()<<std::endl;

std::cout<<"onjects3 count method:"<<obj3.getcount()<<std::endl;

return 0;

}

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.

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.

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.

#include <iostream> // Includes the iostream library for input and output

class DistanceConverter {

public:

static double convertMilesToKm(double miles) {

return miles \* 1.60934; // Converts miles to kilometers

}

static double convertKmToMiles(double kilometers) {

return kilometers / 1.60934; // Converts kilometers to miles

}

};

int main() {

double distance;

char unit;

std::cout << "Enter distance: ";

std::cin >> distance;

std::cout << "Enter unit (m for miles, k for kilometers): ";

std::cin >> unit;

if (unit == 'm') {

double km = DistanceConverter::convertMilesToKm(distance);

std::cout << distance << " miles is " << km << " kilometers." << std::endl;

} else if (unit == 'k') {

double miles = DistanceConverter::convertKmToMiles(distance);

std::cout << distance << " kilometers is " << miles << " miles." << std::endl;

} else {

std::cout << "Invalid unit!" << std::endl;

}

return 0;

}

#include <iostream> // Includes the iostream library for input and output

class MathUtil {

public:

static int add(int a, int b) {

return a + b; // Adds two integers

}

static int subtract(int a, int b) {

return a - b; // Subtracts two integers

}

static int multiply(int a, int b) {

return a \* b; // Multiplies two integers

}

static int divide(int a, int b) {

if (b == 0) {

std::cerr << "Error: Division by zero!" << std::endl;

return 0; // Error handling for division by zero

}

return a / b; // Divides two integers

}

};

int main() {

int num1, num2;

char operation;

std::cout << "Enter two numbers: ";

std::cin >> num1 >> num2;

std::cout << "Enter operation (+, -, \*, /): ";

std::cin >> operation;

int result;

switch (operation) {

case '+':

result = MathUtil::add(num1, num2);

break;

case '-':

result = MathUtil::subtract(num1, num2);

break;

case '\*':

result = MathUtil::multiply(num1, num2);

break;

case '/':

result = MathUtil::divide(num1, num2);

break;

default:

std::cerr << "Invalid operation!" << std::endl;

return 1;

}

std::cout << "Result: " << result << std::endl;

return 0;

}

#include <iostream> // Includes the iostream library for input and output

class CurrencyConverter {

private:

static constexpr double exchangeRate = 0.85; // Exchange rate from USD to EUR

public:

static double convertToEur(double amount) {

return amount \* exchangeRate; // Converts USD to EUR

}

static double convertFromEur(double amount) {

return amount / exchangeRate; // Converts EUR to USD

}

};

int main() {

double amount;

char direction;

std::cout << "Enter amount: ";

std::cin >> amount;

std::cout << "Enter direction (u for USD to EUR, e for EUR to USD): ";

std::cin >> direction;

if (direction == 'u') {

double eur = CurrencyConverter::convertToEur(amount);

std::cout << amount << " USD is " << eur << " EUR." << std::endl;

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

double usd = CurrencyConverter::convertFromEur(amount);

std::cout << amount << " EUR is " << usd << " USD." << std::endl;

} else {

std::cout << "Invalid direction!" << std::endl;

}

return 0;

}

Funtions templates :

#include <iostream>

using namespace std;

template<class T> T add(T &a, T &b)

{

T result = a+b;

return result;

}

int main()

{

int i = 2;

int j = 3;

float m = 2.3;

float n = 1.2;

cout<< "Addition of i and j is :"<<add(i,j);

cout<<'\n';

cout<< "Addition of m and n is :"<<add(m,n);

return 0;

}

FUNTION TEMPLATES WITH MULTIPLE PARAMETERS:

#include <iostream>

using namespace std;

template<class X, class Y>void fun(X a, Y b)

{

std::cout<<"Value of a is:"<<a<<std::endl;

std::cout<<"Value of b is:"<<b<<std::endl;

}

int main()

{

fun(15,12.3);

return 0;

}

OVERLOADING FUNCTION TEMPLATE :

#include <iostream>

using namespace std;

template<class X> void fun(X a)

{

std::cout<<"Value of a is :" <<a<<std::endl;

}

template<class X,class Y> void fun(X b, Y c)

{

std::cout<<"Value of b is :" <<b<<std::endl;

std::cout<<"Value of c is :" <<c<<std::endl;

}

int main()

{

fun(10);

fun(20,30.5);

return 0;

}

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>

template <typename T>

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

if (a > b) {

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

return true;

} else if (a < b) {

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

return false;

} else {

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

return false;

}

}

int main() {

int a = 5, b = 3;

compare(a, b); // Output: 5 is greater than 3

double x = 4.5, y = 4.5;

compare(x, y); // Output: 4.5 is equal to 4.5

std::string str1 = "apple", str2 = "orange";

compare(str1, str2); // Output: apple is less than orange

return 0;

}

#include <iostream>

class Point {

public:

int x, y;

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

bool operator<(const Point& other) const {

return (x < other.x) || (x == other.x && y < other.y);

}

bool operator>(const Point& other) const {

return (x > other.x) || (x == other.x && y > other.y);

}

bool operator==(const Point& other) const {

return (x == other.x && y == other.y);

}

};

template <typename T>

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

if (a > b) {

std::cout << "First argument is greater than second argument." << std::endl;

return true;

} else if (a < b) {

std::cout << "First argument is less than second argument." << std::endl;

return false;

} else {

std::cout << "Both arguments are equal." << std::endl;

return false;

}

}

int main() {

Point p1(3, 4);

Point p2(2, 5);

compare(p1, p2); // Output: First argument is greater than second argument.

Point p3(1, 2);

Point p4(1, 2);

compare(p3, p4); // Output: Both arguments are equal.

return 0;

}

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>

// Function template to swap two variables of the same type

template <typename T>

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

T temp = a; // Store the value of 'a' in a temporary variable

a = b; // Assign the value of 'b' to 'a'

b = temp; // Assign the value of the temporary variable (original 'a') to 'b'

}

int main() {

int x = 10, y = 20;

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

swap(x, y);

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

double a = 1.5, b = 2.5;

std::cout << "Before swap: a = " << a << ", b = " << b << std::endl;

swap(a, b);

std::cout << "After swap: a = " << a << ", b = " << b << std::endl;

std::string str1 = "Hello", str2 = "World";

std::cout << "Before swap: str1 = " << str1 << ", str2 = " << str2 << std::endl;

swap(str1, str2);

std::cout << "After swap: str1 = " << str1 << ", str2 = " << str2 << std::endl;

return 0;

}

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>

// Function template to find the minimum value in an array

template <typename T>

T findMin(const T\* arr, size\_t size) {

if (size == 0) {

throw std::invalid\_argument("Array size must be greater than 0");

}

T min = arr[0]; // Assume the first element is the minimum

for (size\_t i = 1; i < size; ++i) {

if (arr[i] < min) {

min = arr[i]; // Update min if current element is smaller

}

}

return min; // Return the minimum value found

}

int main() {

int intArr[] = {3, 1, 4, 1, 5, 9};

size\_t intSize = sizeof(intArr) / sizeof(intArr[0]);

std::cout << "Minimum in intArr: " << findMin(intArr, intSize) << std::endl;

double doubleArr[] = {3.1, 4.1, 5.9, 2.6};

size\_t doubleSize = sizeof(doubleArr) / sizeof(doubleArr[0]);

std::cout << "Minimum in doubleArr: " << findMin(doubleArr, doubleSize) << std::endl;

std::string strArr[] = {"apple", "orange", "banana", "pear"};

size\_t strSize = sizeof(strArr) / sizeof(strArr[0]);

std::cout << "Minimum in strArr: " << findMin(strArr, strSize) << std::endl;

return 0;

}