**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.**

#include <iostream>

#include <string>

#include <vector>

#include <ctime>

// Forward declaration of the Post class

class Post;

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) {}

// Accessor methods

std::string getUsername() const

{

return username;

}

std::string getProfilePicture() const

{

return profilePicture;

}

// Add a friend

void addFriend(User\* user)

{

friends.push\_back(user);

user->friends.push\_back(this); // Mutual friendship

}

// Check if a user is a friend

bool isFriend(User\* user) const

{

for (const auto& friendPtr : friends)

{

if (friendPtr == user)

{

return true;

}

}

return false;

}

// Interaction methods

void interactWith(User& other)

{

if (isFriend(&other))

{

std::cout << username << " interacts with friend " << other.getUsername() << std::endl;

}

else

{

std::cout << username << " interacts with " << other.getUsername() << std::endl;

}

}

void likePost(Post& post);

void follow(User& other)

{

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

}

};

// Post class definition

class Post : public User

{

private:

std::string postContent;

std::time\_t timestamp;

public:

// Constructor

Post(const std::string& uname, const std::string& ppic, const std::string& content, std::time\_t time)

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

// Method to get post information

std::string getPostInfo() const

{

char timeBuffer[26];

ctime\_r(&timestamp, timeBuffer); // Convert timestamp to string

std::string info = "Username: " + getUsername() + "\n";

info += "Profile Picture: " + getProfilePicture() + "\n";

info += "Content: " + postContent + "\n";

info += "Timestamp: " + std::string(timeBuffer);

return info;

}

};

// Define likePost outside the User class after Post class is defined

void User::likePost(Post& post)

{

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

}

// Main function to demonstrate functionality

int main()

{

std::time\_t now = std::time(nullptr);

User user1("Suresh", "suresh.jpg");

User user2("Ramesh", "ramesh.jpg");

Post post("Suresh", "suresh.jpg", "Hello World!", now);

user1.interactWith(user2);

user1.likePost(post);

user1.follow(user2);

user1.addFriend(&user2);

user1.interactWith(user2); // Now they are friends

std::cout << post.getPostInfo() << std::endl;

return 0;

}

**OUTPUT:**

**Suresh interacts with Ramesh**

**Suresh liked the post by Suresh**

**Suresh started following Ramesh**

**Suresh interacts with friend Ramesh**

**Username: Suresh**

**Profile Picture: suresh.jpg**

**Content: Hello World!**

**Timestamp: Tue Jul 2 04:48:17 2024**

**TASK 2:**

#include <iostream>

using namespace std;

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;

}

OUTPUT:

**Number of objects created: 3**

**TASK 3:**

#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;

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

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

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

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

return 0;

}

**OUTPUT:**

**Number of objects created: 3**

**objects1 count method: 1**

**objects2 count method: 4198561**

**objects3 count method: 1**

**TASK 4:**

#include <iostream>

using namespace std;

class DistanceConverter {

public:

static double convertMilesToKm(double miles) {

return miles \* 1.60934;

}

static double convertKmToMiles(double kilometers) {

return kilometers / 1.60934;

}

};

class MathUtil {

public:

static int add(int a, int b) {

return a + b;

}

static int subtract(int a, int b) {

return a - b;

}

static int multiply(int a, int b) {

return a \* b;

}

static double divide(int a, int b) {

if (b == 0) {

cout << "Error: Division by zero!" << endl;

return 0;

}

return static\_cast<double>(a) / b;

}

};

class CurrencyConverter {

public:

static double exchangeRate; // USD to EUR exchange rate

static double convertToEur(double amount) {

return amount \* exchangeRate;

}

static double convertFromEur(double amount) {

return amount / exchangeRate;

}

};

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

int main() {

// Distance Converter

cout << "Distance Converter\n";

cout << "Enter distance: ";

double distance;

cin >> distance;

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

char unit;

cin >> unit;

if (unit == 'm') {

cout << distance << " miles is " << DistanceConverter::convertMilesToKm(distance) << " kilometers." << endl;

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

cout << distance << " kilometers is " << DistanceConverter::convertKmToMiles(distance) << " miles." << endl;

} else {

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

}

// Math Utility

cout << "\nMath Utility\n";

cout << "Enter first number: ";

int num1, num2;

cin >> num1;

cout << "Enter second number: ";

cin >> num2;

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

char operation;

cin >> operation;

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!" << endl;

break;

}

// Currency Converter

cout << "\nCurrency Converter\n";

cout << "Enter amount: ";

double amount;

cin >> amount;

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

cin >> unit;

if (unit == 'u') {

cout << amount << " USD is " << CurrencyConverter::convertToEur(amount) << " EUR." << endl;

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

cout << amount << " EUR is " << CurrencyConverter::convertFromEur(amount) << " USD." << endl;

} else {

cout << "Invalid conversion direction!" << endl;

}

return 0;

}

**OUTPUT:**

**Distance Converter**

**Enter distance: 50**

**Enter unit (m for miles, k for kilometers): k**

**50 kilometers is 31.0686 miles.**

**Math Utility**

**Enter first number: 25**

**Enter second number: 35**

**Enter operation (+, -, \*, /): +**

**Result: 60**

**Currency Converter**

**Enter amount: 500**

**Enter conversion direction (u for USD to EUR, e for EUR to USD): u**

**500 USD is 425 EUR.**

**TASK 5**

#include <iostream>

using namespace std;

template<class X> void fun(X a)

{

cout<<"value of a :"<<a<<endl;

}

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

{

cout<<"value of b :"<<b<<endl;

cout<<"value of c :"<<c<<endl;

}

int main()

{

fun(10);

fun(15,12.3);

return 0;

}

**OUTPUT:**

**value of a :10**

**value of b :15**

**value of c :12.3**

**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 function to compare two values

template <typename T>

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

if (a < b) {

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

return true;

} else if (a > b) {

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

return true;

} else {

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

return false;

}

}

// Example custom data type

class MyClass {

public:

int value;

MyClass(int v) : value(v) {}

// Overloading the < operator for MyClass

bool operator<(const MyClass& other) const {

return value < other.value;

}

// Overloading the > operator for MyClass

bool operator>(const MyClass& other) const {

return value > other.value;

}

// Overloading the == operator for MyClass

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

return value == other.value;

}

// Overloading the << operator for MyClass

friend ostream& operator<<(ostream& os, const MyClass& obj) {

os << obj.value;

return os;

}

};

int main() {

int a = 5, b = 10;

compare(a, b);

MyClass obj1(5), obj2(10);

compare(obj1, obj2);

return 0;

}

**OUTPUT:**

**5 is less than 10**

**5 is less than 10**

**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>

#include <string>

using namespace std;

// Template function to swap two values

template <typename T>

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

T temp = a;

a = b;

b = temp;

}

int main() {

int x = 1, y = 2;

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

customSwap(x, y);

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

double dx = 1.1, dy = 2.2;

cout << "Before swap: dx = " << dx << ", dy = " << dy << endl;

customSwap(dx, dy);

cout << "After swap: dx = " << dx << ", dy = " << dy << endl;

string sx = "Hello", sy = "World";

cout << "Before swap: sx = " << sx << ", sy = " << sy << endl;

customSwap(sx, sy);

cout << "After swap: sx = " << sx << ", sy = " << sy << endl;

return 0;

}

**OUTPUT:**

**Before swap: x = 1, y = 2**

**After swap: x = 2, y = 1**

**Before swap: dx = 1.1, dy = 2.2**

**After swap: dx = 2.2, dy = 1.1**

**Before swap: sx = Hello, sy = World**

**After swap: sx = World, sy = Hello**

**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 function to find the minimum value in an array

template <typename T>

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

if (size <= 0) {

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

}

T min = array[0];

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

if (array[i] < min) {

min = array[i];

}

}

return min;

}

int main() {

int intArray[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};

int intSize = sizeof(intArray) / sizeof(intArray[0]);

cout << "Minimum value in intArray: " << findMin(intArray, intSize) << endl;

double doubleArray[] = {2.718, 3.141, 1.414, 1.732, 0.577, 1.618};

int doubleSize = sizeof(doubleArray) / sizeof(doubleArray[0]);

cout << "Minimum value in doubleArray: " << findMin(doubleArray, doubleSize) << endl;

string strArray[] = {"apple", "orange", "banana", "grape", "kiwi"};

int strSize = sizeof(strArray) / sizeof(strArray[0]);

cout << "Minimum value in strArray: " << findMin(strArray, strSize) << endl;

return 0;

}

**OUTPUT:**

**Minimum value in intArray: 1**

**Minimum value in doubleArray: 0.577**

**Minimum value in strArray: apple**