**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.**

#include <iostream>

#include <string>

using namespace std;

class User {

private:

string username;

string profile\_picture;

public:

// Constructor

User(const string& uname, const string& profile\_pic)

: username(uname), profile\_picture(profile\_pic) {}

// Getter for username

string getUsername() const {

return username;

}

// Getter for profile picture

string getProfilePicture() const {

return profile\_picture;

}

};

int main() {

User obj("Kavya", "profile\_pic.jpg");

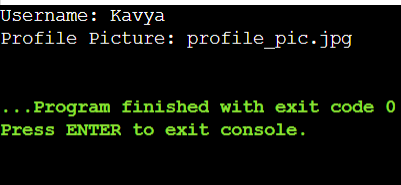
cout << "Username: " << obj.getUsername() << endl;

cout << "Profile Picture: " << obj.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.**

#include <iostream>

#include <string>

#include <ctime>

using namespace std;

class User {

private:

string username;

string profile\_picture;

public:

// Constructor

User(const string& uname, const string& profile\_pic)

: username(uname), profile\_picture(profile\_pic) {}

// Getter for username

string getUsername() const {

return username;

}

// Getter for profile picture

string getProfilePicture() const {

return profile\_picture;

}

};

class Post : public User {

private:

string post\_content;

string timestamp;

public:

// Constructor

Post(const string& uname, const string& profile\_pic, const string& content)

: User(uname, profile\_pic), post\_content(content) {

// Set the current time as the timestamp

time\_t now = time(0);

timestamp = ctime(&now);

}

// Getter for post content

string getPostContent() const {

return post\_content;

}

// Getter for timestamp

string getTimestamp() const {

return timestamp;

}

// Get post info

string getPostInfo() const {

return "Username: " + getUsername() + "\n" +

"Profile Picture: " + getProfilePicture() + "\n" +

"Post Content: " + getPostContent() + "\n" +

"Timestamp: " + getTimestamp();

}

};

int main() {

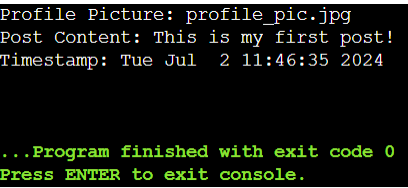
Post obj("Kavya", "profile\_pic.jpg", "This is my first post!");

cout << obj.getPostInfo() << endl;

return 0;

}

**Output:**



**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."**

#include <iostream>

#include <string>

using namespace std;

class User {

private:

string username;

string profile\_picture;

public:

// Constructor

User(const string& uname, const string& profile\_pic)

: username(uname), profile\_picture(profile\_pic) {}

// Getter for username

string getUsername() const {

return username;

}

// Getter for profile picture

string getProfilePicture() const {

return profile\_picture;

}

// Friend function declaration

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

};

class Post : public User {

private:

string post\_content;

string timestamp;

public:

// Constructor

Post(const string& uname, const string& profile\_pic, const string& content)

: User(uname, profile\_pic), post\_content(content) {

// Set the current time as the timestamp

time\_t now = time(0);

timestamp = ctime(&now);

}

// Getter for post content

string getPostContent() const {

return post\_content;

}

// Getter for timestamp

string getTimestamp() const {

return timestamp;

}

// Get post info

string getPostInfo() const {

return "Username: " + getUsername() + "\n" +

"Profile Picture: " + getProfilePicture() + "\n" +

"Post Content: " + getPostContent() + "\n" +

"Timestamp: " + getTimestamp();

}

};

// Friend function definition

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

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

}

int main() {

User obj1("Kavya", "profile\_pic1.jpg");

User obj2("Rohitha", "profile\_pic2.jpg");

basicInteract(obj1, obj2);

Post obj3("Kavya", "profile\_pic1.jpg", "This is my first post!");

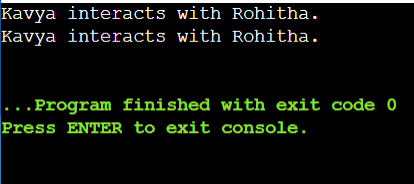
Post obj4("Rohitha", "profile\_pic2.jpg", "This is my second post!");

basicInteract(obj3, obj4);

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.**

#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("Kavya");

User user2("Hari");

Post post("Hello World!");

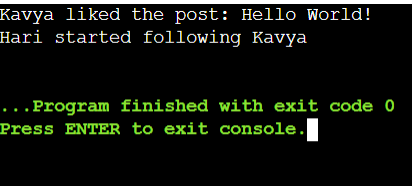
interact(user1, post); // User liking a post

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>

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;

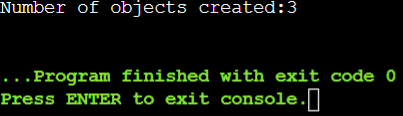
MyClass obj3;

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

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<<"object1 count method:"<<obj1.getCount()<<endl;

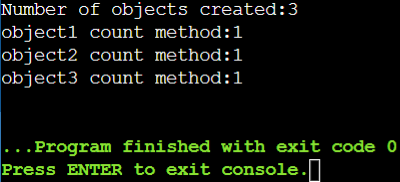
cout<<"object2 count method:"<<obj2.getCount()<<endl;

cout<<"object3 count method:"<<obj3.getCount()<<endl;

return 0;

}

**Output:**



**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.**

#include <iostream>

using namespace std;

class DistanceConverter {

public:

// Static method to convert miles to kilometers

static double convertMilesToKm(double miles) {

return miles \* 1.60934;

}

// Static method to convert kilometers to miles

static double convertKmToMiles(double kilometers) {

return kilometers / 1.60934;

}

};

int main() {

double distance;

string unit;

cout << "Enter distance: ";

cin >> distance;

cout << "Enter unit (miles or kilometers): ";

cin >> unit;

if (unit == "miles") {

double km = DistanceConverter::convertMilesToKm(distance);

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

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

double miles = DistanceConverter::convertKmToMiles(distance);

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

} else {

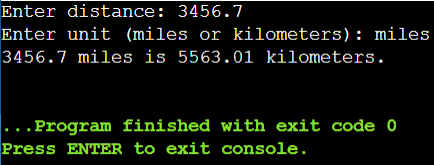
cout << "Invalid unit." << endl;

}

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>

using namespace std;

class MathUtil {

public:

// Static method to add two integers

static int add(int a, int b) {

return a + b;

}

// Static method to subtract two integers

static int subtract(int a, int b) {

return a - b;

}

// Static method to multiply two integers

static int multiply(int a, int b) {

return a \* b;

}

// Static method to divide two integers with error handling for division by zero

static double divide(int a, int b) {

if (b == 0) {

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

return 0; // or handle error as needed

}

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

}

};

int main() {

int num1, num2;

char operation;

cout << "Enter first number: ";

cin >> num1;

cout << "Enter second number: ";

cin >> num2;

cout << "Enter 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;

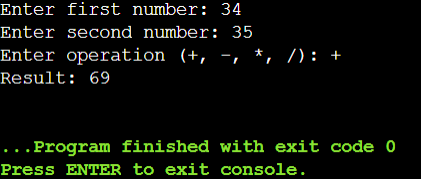
break;

}

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.**

#include <iostream>

using namespace std;

class CurrencyConverter {

public:

// Static method to convert USD to EUR (assuming exchange rate 1 USD = 0.85 EUR)

static double convertToEur(double amount) {

return amount \* 0.85;

}

// Static method to convert EUR to USD

static double convertFromEur(double amount) {

return amount / 0.85;

}

};

int main() {

double amount;

char direction;

// Prompt user for input

cout << "Enter amount: ";

cin >> amount;

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

cin >> direction;

// Perform conversion based on direction

switch (direction) {

case 'u':

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

break;

case 'e':

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

break;

default:

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

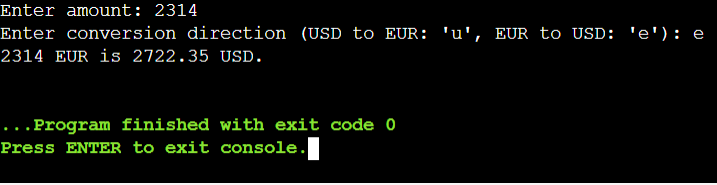
break;

}

return 0;

}

**Output:**



**Function 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);

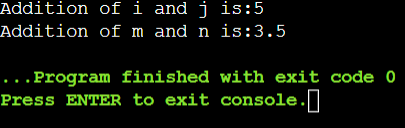
cout<<"\n";

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

return 0;

}

**Output:**



**Function Templates With Multiple Parameters:**

#include <iostream>

using namespace std;

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

{

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

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

}

int main()

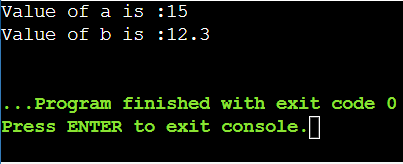
{

fun(15,12.3);

return 0;

}

**Output:**



**Overloading Function Template:**

#include <iostream>

using namespace std;

template<class X>

void fun(X a)

{

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

}

template<class X,class Y>

void fun(X b,Y c)

{

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

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

}

int main()

{

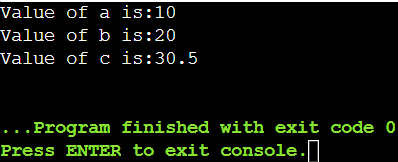
fun(10);

fun(20,30.5);

return 0;

}

**Output:**



**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);

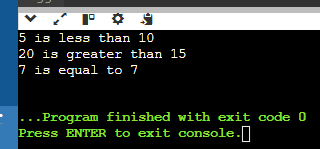
int e = 7, f = 7;

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;

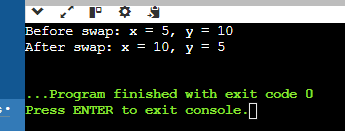
cout << "Before swap: x = " << x << ", y = " << y << endl;

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:**  
