Day-7:

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 <ctime>

using namespace std;

class User {

private:

string username;

string profilePicture;

protected:

// Protected accessor methods for derived classes

string getUsername() { return username; }

string getProfilePicture() { return profilePicture; }

public:

// Constructor

User(string username, string profilePicture) : username(username), profilePicture(profilePicture) {}

// Public accessor methods

string getPublicUsername() { return username; }

string getPublicProfilePicture() { return profilePicture; }

};

class Post : public User {

private:

string postContent;

time\_t timestamp;

public:

// Constructor

Post(string username, string profilePicture, string postContent) : User(username, profilePicture), postContent(postContent) {

timestamp = time(0); // Set timestamp to current time

}

// Get post information

string getPostInfo() {

return "Username: " + getUsername() + ", Profile Picture: " + getProfilePicture() + ", Post Content: " + postContent + ", Timestamp: " + ctime(&timestamp);

}

};

// Basic interaction function (refactored to use public accessor methods)

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

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

}

// Overloaded interact functions

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

cout << "User " << user.getPublicUsername() << " likes Post by " << post.getPublicUsername() << "." << endl;

}

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

cout << "User " << follower.getPublicUsername() << " starts following User " << followed.getPublicUsername() << "." << endl;

}

// Bonus Challenge: Implementing friend connections

class Friendship {

private:

User\* user1;

User\* user2;

public:

Friendship(User& user1, User& user2) : user1(&user1), user2(&user2) {}

void interact() {

if (areFriends()) {

cout << "Friends " << user1->getPublicUsername() << " and " << user2->getPublicUsername() << " interact." << endl;

} else {

cout << "User " << user1->getPublicUsername() << " interacts with User " << user2->getPublicUsername() << "." << endl;

}

}

bool areFriends() {

// Simple implementation: assume friendship is mutual

return true; // Replace with actual friendship logic

}

};

int main() {

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

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

Post post1("Alice", "alice.jpg", "Hello, world!");

basicInteract(user1, user2);

likePost(user2, post1);

followUser(user2, user1);

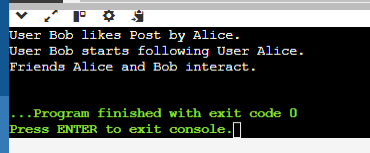
Friendship friendship(user1, user2);

friendship.interact();

return 0;

}

Output:



Static Data Members:

#include <iostream>

class MyClass {

private:

static int counter; // Static variable to keep track of object instances

public:

MyClass() {

counter++;

} // Increment counter for each object creation

static int getCount() { // Static method to access and return the counter

return counter;

}

};

// Initialize static variable outside the class (required)

int MyClass::counter = 0; // Set initial value to e

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

#include <iostream>

class MyClass {

private:

static int counter; // Static variable to keep track of object instances

int count; // Non-static variable to keep track of instance count

public:

MyClass() {

count = 0; // Initialize count to 0 for each instance

counter++; // Increment static counter for each object creation

count++; // Increment instance count for this object

}

static int getCounter() {

return counter; // Return static counter

}

int getCount() {

return count; // Return instance count for this object

}

};

// Initialize static variable outside the class (required)

int MyClass::counter = 0;

int main() {

MyClass obj1;

MyClass obj2;

MyClass obj3;

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

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

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

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

return 0;

}

OUTPUT:

Number of objects created: 3

obj1 count: 1

obj2 count: 1

obj3 count: 1

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:

#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;

// Prompt user for input

cout << "Enter first number: ";

cin >> num1;

cout << "Enter second number: ";

cin >> num2;

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

cin >> operation;

// Perform calculation based on 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;

}

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;

}

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:

#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;

// Prompt user for distance and unit

cout << "Enter distance: ";

cin >> distance;

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

cin >> unit;

// Perform conversion based on the 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;

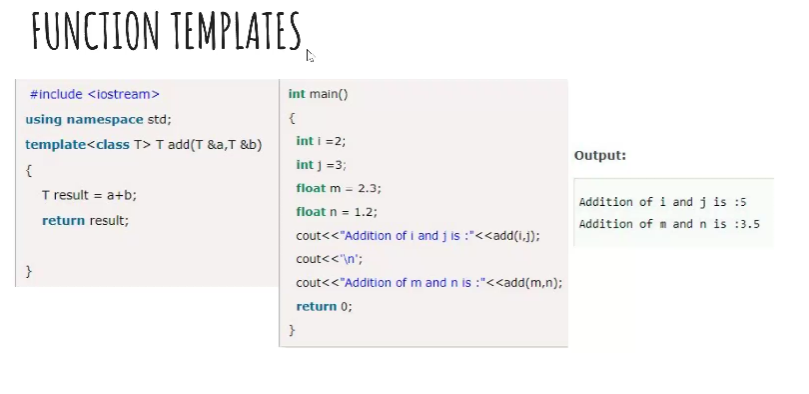
} else {

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

}

return 0;

}



#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<<"Addtion of i and j is"<<add(i,j);

cout<<'\n';

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

return 0;

}

FUNCTION 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);

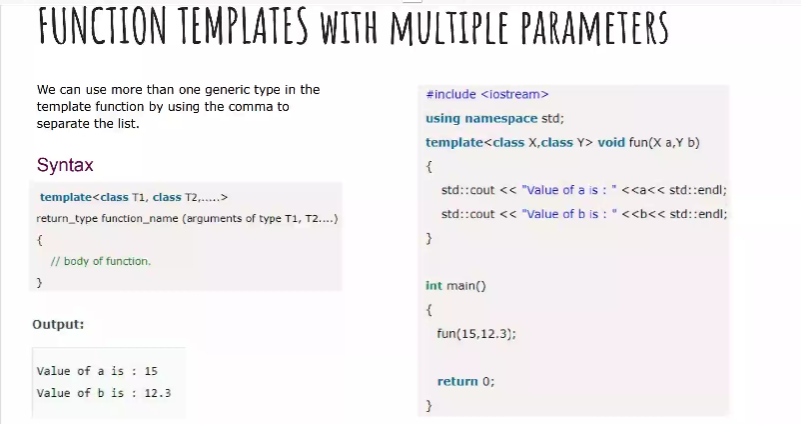
return 0;

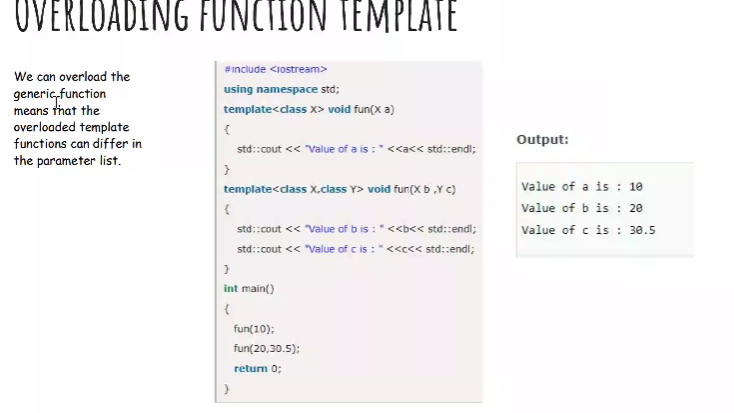
}

OUTPUT:

value of a is:15

value of b is:12.3





#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"<<b<<endl;

cout<<" value "<<c<<endl;

}

int main()

{

fun(10);

fun(20,30.5);

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.

#include <iostream>

#include <string>

using namespace std;

class DistanceConverter {

public:

static double convertMilesToKm(double miles) {

return miles \* 1.60934;

}

static double convertKmToMiles(double kilometers) {

return kilometers / 1.60934;

}

};

int main() {

double distance;

string unit;

distance = 10.0;

unit = "miles";

if (unit == "miles") {

double kilometers = DistanceConverter::convertMilesToKm(distance);

printf("%.2f miles is equal to %.2f kilometers.\n", distance, kilometers);

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

double miles = DistanceConverter::convertKmToMiles(distance);

printf("%.2f kilometers is equal to %.2f miles.\n", distance, miles);

} else {

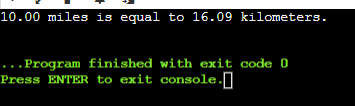
printf("Invalid unit. Please enter miles or kilometers.\n");

}

return 0;

}

Output:



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 <string>

using namespace std;

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) {

throw ("Division by zero is not allowed");

}

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

}

};

int main() {

int num1, num2;

char operation;

cout << "Enter the first number: ";

std::cin >> num1;

cout << "Enter the second number: ";

std::cin >> num2;

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

std::cin >> operation;

try {

if (operation == '+') {

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

} else if (operation == '-') {

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

} else if (operation == '\*') {

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

} else if (operation == '/') {

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

} else {

std::cout << "Invalid operation. Please enter +, -, \*, or /." << std::endl;

}

} catch (const std::runtime\_error& e) {

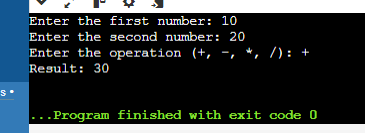
std::cout << "Error: " << e.what() << std::endl;

}

return 0;

}

Output:



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 double exchangeRate; // USD to EUR exchange rate

// Converts an amount from USD to EUR

static double convertToEur(double amount) {

return amount \* exchangeRate;

}

// Converts an amount from EUR to USD

static double convertFromEur(double amount) {

return amount / exchangeRate;

}

};

// Initialize the static variable

double CurrencyConverter::exchangeRate = 0.85; // Example exchange rate

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 convertedAmount = CurrencyConverter::convertToEur(amount);

cout << amount << " USD is " << convertedAmount << " EUR." << endl;

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

double convertedAmount = CurrencyConverter::convertFromEur(amount);

cout << amount << " EUR is " << convertedAmount << " USD." << endl;

} else {

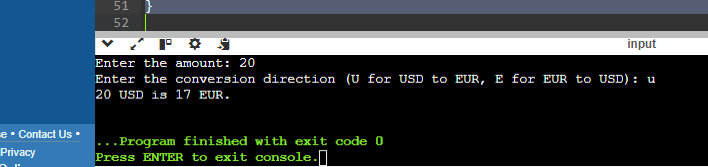
cout << "Invalid conversion direction entered." << endl;

}

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?

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.

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>

template <typename T>

bool compare(T a, T b, char op) {

switch (op) {

case '>':

return a > b;

case '<':

return a < b;

case '=':

return a == b;

default:

throw std::invalid\_argument("Invalid operator");

}

}

template <typename T>

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

T temp = a;

a = b;

b = temp;

}

int main() {

int x = 5;

int y = 10;

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

swap(x, y);

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

return 0;

}

Output:

