Problem Statement:

Write a generic function template named findMinimum in C++ that takes an array of any data type T and its size n as arguments. The function should return the minimum element present in the array.

[08:52, 7/3/2024] +91 79820 39479: 1. Swap Elements:

Problem: Write a function template swap that takes two pointers to variables of any data type T and swaps their values.

Constraints: The function should only modify the values pointed to by the arguments, not the arguments themselves (pass by reference):

#include <iostream>

using namespace std;

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

template <typename T>

T findMinimum(T arr[], int n) {

// Assume the first element is the minimum

T min = arr[0];

// Iterate through the array to find the minimum element

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

if (arr[i] < min) {

min = arr[i];

}

}

// Return the minimum element found

return min;

}

int main() {

int arr[] = {4, 2, 7, 1, 9};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Minimum element is: " << findMinimum(arr, n) << endl;

return 0;

}

#include <iostream>

using namespace std;

// Function template to swap the values of two variables using pointers

template <typename T>

void swap(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;

swap(&x, &y);

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

return 0;

}

2. Find Maximum:

Problem: Similar to findMinimum, create a function template findMaximum that returns the maximum element in an array of any data type T

#include <iostream>

using namespace std;

// Function template to find the maximum element in an array

template <typename T>

T findMaximum(T arr[], int n) {

// Assume the first element is the maximum

T max = arr[0];

// Iterate through the array to find the maximum element

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

if (arr[i] > max) {

max = arr[i];

}

}

// Return the maximum element found

return max;

}

int main() {

int arr[] = {4, 2, 7, 1, 9};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Maximum element is: " << findMaximum(arr, n) << endl;

return 0;

}

GENERIC TEMPLATE:

#include<iostream>

using namespace std;

void fun(double a)

{

cout<<"value of a is:"<<a<<'\n';

}

void fun(int b){

if(b%2==0)

{

cout<<"number is even";

}

else

{

cout<<"number is odd";

}

}

int main(){

fun(4.6);

fun(6);

return 0;

}

CLASS TEMPLATE:

#include<iostream>

using namespace std;

template<class T>

class A

{

public:

T num1=5;

T num2 =6;

void add(){

std::cout<<"addition of num1 and num2:"<<num1+num2<<std::endl;

}

};

int main()

{

A<int>d;

d.add();

return 0;

}

CLASS TEMPLATE USING MULTIPLE PARAMETERS :

#include <iostream>

using namespace std;

template<class T1, class T2>

class A {

T1 a;

T2 b;

public:

A(T1 x, T2 y)

{

a = x;

b = y;

}

void display()

{

cout<<"Value of a and b are:"<<a<<","<<b<<endl;

}

};

int main()

{

A<int,float>d(5,6.5);

d.display();

return 0;

}

Design a generic data processing library using class and function templates in C++. This library should be able to handle various data types (e.g., integers, floats, strings) without code duplication.

Requirements:

Create a class template named DataContainer that can hold elements of any data type specified during instantiation.

Implement member functions for DataContainer:

DataContainer(size\_t size): Constructor to initialize the container with a specific size.

T& operator[](size\_t index): Overloaded subscript operator to access elements.

void printAll(): Prints all elements of the container.

Create a function template named swap that takes two DataContainer objects as arguments and swaps their elements.

Ensure proper memory management using appropriate constructors and destructors.

Coding Practice Questions:

Implement the DataContainer class template:

Define the template parameter to specify the data type.

Use an array or a vector internally to store the elements.

Implement the constructor, subscript operator, and printAll function as described in the requirements.

Implement the swap function template:

Take two DataContainer objects as arguments.

Use a loop or recursion to iterate over corresponding elements and swap their values.

Consider potential edge cases (e.g., containers of different sizes).

Write a main function to demonstrate the library:

Create instances of DataContainer for different data types (e.g., int, float, string).

Populate the containers with sample data.

Call printAll on each container to verify its contents.

Use the swap function to swap elements between containers of the same type.

Print the containers again to confirm the swap.

Enhance the DataContainer class:

Add member functions for:

size(): Returns the current size of the container.

push\_back(const T& value): Appends an element to the back of the container (dynamically resize if necessary).

Modify the constructor to accept an optional initial size (default to 0).

Explore advanced functionalities (optional):

Implement a class template for linked lists or binary search trees, leveraging the DataContainer class.

Create function templates for generic sorting algorithms (e.g., bubble sort, selection sort).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Design a generic data processing library using class and function templates in C++. This library should be able to handle various data types (e.g., integers, floats, strings) without code duplication.

Requirements:

Create a class template named DataContainer that can hold elements of any data type specified during instantiation.

Implement member functions for DataContainer:

DataContainer(size\_t size): Constructor to initialize the container with a specific size.

T& operator[](size\_t index): Overloaded subscript operator to access elements.

void printAll(): Prints all elements of the container.

Create a function template named swap that takes two DataContainer objects as arguments and swaps their elements.

Ensure proper memory management using appropriate constructors and destructors.

#include <iostream>

#include <vector>

template <typename T>

class DataContainer {

private:

std::vector<T> data; // Use a vector to store elements of type T

public:

// Constructor to initialize the container with a specific size

DataContainer(size\_t size) : data(size) {}

// Overloaded subscript operator to access elements

T& operator[](size\_t index) {

return data[index];

}

// Function to print all elements in the container

void printAll() const {

for (const auto& element : data) {

std::cout << element << " ";

}

std::cout << std::endl;

}

};

// Function template to swap elements between two DataContainer objects

template <typename T>

void swap(DataContainer<T>& container1, DataContainer<T>& container2) {

// Check if the sizes are different

if (container1.size() != container2.size()) {

std::cerr << "Containers are of different sizes!" << std::endl;

return;

}

// Swap elements between the two containers

for (size\_t i = 0; i < container1.size(); ++i) {

std::swap(container1[i], container2[i]);

}

}

// Main function to demonstrate the library

int main() {

// Create instances of DataContainer for different data types

DataContainer<int> intContainer(5);

DataContainer<float> floatContainer(3);

DataContainer<std::string> stringContainer(4);

// Populate the containers with sample data

for (int i = 0; i < 5; ++i) {

intContainer[i] = i + 1;

}

floatContainer[0] = 1.1f;

floatContainer[1] = 2.2f;

floatContainer[2] = 3.3f;

stringContainer[0] = "Hello";

stringContainer[1] = "World";

stringContainer[2] = "C++";

stringContainer[3] = "Templates";

// Call printAll on each container to verify its contents

std::cout << "Integer Container: ";

intContainer.printAll();

std::cout << "Float Container: ";

floatContainer.printAll();

std::cout << "String Container: ";

stringContainer.printAll();

// Create another integer container to demonstrate swapping

DataContainer<int> anotherIntContainer(5);

for (int i = 0; i < 5; ++i) {

anotherIntContainer[i] = (i + 1) \* 10;

}

std::cout << "Another Integer Container before swap: ";

anotherIntContainer.printAll();

// Use the swap function to swap elements between intContainer and anotherIntContainer

swap(intContainer, anotherIntContainer);

// Print the containers again to confirm the swap

std::cout << "Integer Container after swap: ";

intContainer.printAll();

std::cout << "Another Integer Container after swap: ";

anotherIntContainer.printAll();

return 0;

}

2. Coding Practice Questions:

Implement the DataContainer class template:

Define the template parameter to specify the data type.

Use an array or a vector internally to store the elements.

Implement the constructor, subscript operator, and printAll function as described in the requirements.

3.Implement the swap function template:

Take two DataContainer objects as arguments.

Use a loop or recursion to iterate over corresponding elements and swap their values.

Consider potential edge cases (e.g., containers of different sizes).

4.Write a main function to demonstrate the library:

Create instances of DataContainer for different data types (e.g., int, float, string).

Populate the containers with sample data.

Call printAll on each container to verify its contents.

Use the swap function to swap elements between containers of the same type.

Print the containers again to confirm the swap.

5.Enhance the DataContainer class:

Add member functions for:

size(): Returns the current size of the container.

push\_back(const T& value): Appends an element to the back of the container (dynamically resize if necessary).

Modify the constructor to accept an optional initial size (default to 0).

6.Explore advanced functionalities (optional):

Implement a class template for linked lists or binary search trees, leveraging the DataContainer class.

Create function templates for generic sorting algorithms (e.g., bubble sort, selection sort).

SMART POINTER:

#include<iostream>

using namespace std;

template <class T>

class Smartpointer{

T \* p;

public:

Smartpointer(T \*ptr = NULL){

p=ptr;

}

~Smartpointer(){

delete(p);

}

T & operator\*(){

return \*p;

}

T\* operator->(){

return p;

}

};

int main(){

Smartpointer<int> p(new int());

\*p = 26;

cout<<"value is:"<<\*p;

return 0;

}

#include <iostream>

#include <cmath>

// Abstract base class

class Shape {

public:

// Pure virtual function

virtual double area() const = 0;

// Virtual destructor

virtual ~Shape() {}

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

double width;

double height;

public:

// Constructor

Rectangle(double w, double h) : width(w), height(h) {}

// Override the pure virtual function

double area() const override {

return width \* height;

}

};

// Derived class Circle

class Circle : public Shape {

private:

double radius;

public:

// Constructor

Circle(double r) : radius(r) {}

// Override the pure virtual function

double area() const override {

return M\_PI \* radius \* radius;

}

};

// Function to print the area of a Shape

void printArea(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

}

int main() {

// Create objects of derived classes

Rectangle rect(10, 5);

Circle circ(7);

// Print the area of the shapes

printArea(rect);

printArea(circ);

return 0;

}

In object-oriented programming with C++, abstract classes are a valuable tool for defining common interfaces and behaviors for a group of related classes. However, directly creating objects from an abstract class is not possible. This problem statement explores how abstract classes are used to enforce a design pattern and promote code reusability. explain me each line of the code since iam very new to c++

#include <iostream>

#include <cmath>

// Abstract base class

class Shape {

public:

// Pure virtual function

virtual double area() const = 0;

// Virtual destructor

virtual ~Shape() {}

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

double width;

double height;

public:

// Constructor

Rectangle(double w, double h) : width(w), height(h) {}

// Override the pure virtual function

double area() const override {

return width \* height;

}

};

// Derived class Circle

class Circle : public Shape {

private:

double radius;

public:

// Constructor

Circle(double r) : radius(r) {}

// Override the pure virtual function

double area() const override {

return M\_PI \* radius \* radius;

}

};

// Function to print the area of a Shape

void printArea(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

}

int main() {

// Create objects of derived classes

Rectangle rect(10, 5);

Circle circ(7);

// Print the area of the shapes

printArea(rect);

printArea(circ);

return 0;

}

OWN SMART POINTER EXAMPLE:

#include <iostream>

#include <memory>

#include <string>

int main() {

// Create a unique\_ptr to a string

std::unique\_ptr<std::string> ptr = std::make\_unique<std::string>("Supriya");

// Use the unique\_ptr to access the string

std::cout << "Name: " << \*ptr << std::endl;

// No need to delete the pointer, it will be automatically cleaned up

return 0;

}