TASK - 1

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.

#include <iostream>

using namespace std;

template <typename T>

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

T min = arr[0];

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

if (arr[i] > min) {

min = arr[i];

}

}

return min;

}

int main() {

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

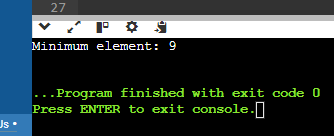
int n = sizeof(arr) / sizeof(arr[0]);

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

return 0;

}

OUTPUT:



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>

#include <string>

using namespace std;

template <typename T>

void swap(T\* a, T\* b) {

T temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

string str1 = "Welcome";

string str2 = "C++";

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

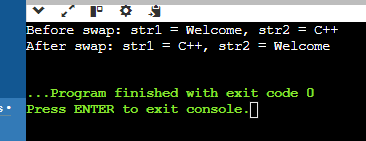
swap(&str1, &str2);

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

return 0;

}

OUTPUT:



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;

template <typename T>

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

T max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

int main() {

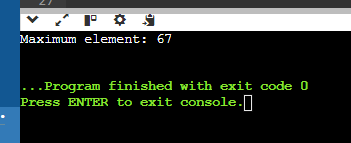
int arr[] = {11, 23, 4, 9, 56, 67, 45};

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

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

return 0;

}

OUTPUT:  


-------------------------------------------------------------------------------------------------------------------------------

TASK – 2

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>

using namespace std;

template <typename T>

class DataContainer {

private:

T\* data;

size\_t size;

public:

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

data = new T[size];

} ~DataContainer() {

delete[] data;

}

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

return data[index];

}

void printAll() const {

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

cout << data[i] << " ";

}

cout << endl;

}

size\_t getSize() const {

return size;

}

void setSize(size\_t newSize) {

size = newSize;

} T\* getData() const {

return data;

}

void setData(T\* newData) {

data = newData;

}

};

template <typename T> // Function template to swap two DataContainer objects

void swap(DataContainer<T>& a, DataContainer<T>& b) {

T\* tempData = a.getData();

size\_t tempSize = a.getSize();

a.setData(b.getData());

a.setSize(b.getSize());

b.setData(tempData);

b.setSize(tempSize);

}

int main() {

DataContainer<int> container1(3);

DataContainer<int> container2(3);

container1[0] = 1;

container1[1] = 2;

container1[2] = 3;

container2[0] = 4;

container2[1] = 5;

container2[2] = 6;

cout << "Container 1 before swap: ";

container1.printAll();

cout << "Container 2 before swap: ";

container2.printAll();

swap(container1, container2);

cout << "Container 1 after swap: ";

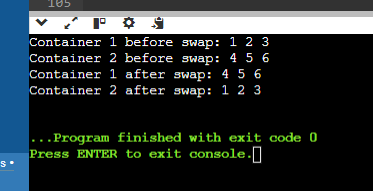
container1.printAll();

cout << "Container 2 after swap: ";

container2.printAll();

return 0;

}



Coding Practice Questions:

1. 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.

A: #include <iostream>

using namespace std;

template<typename T>

class DataContainer {

private:

T\* data; // Raw array for internal storage

size\_t size; // Size of the container

public:

DataContainer(size\_t size) : size(size) { // Constructor to initialize the container

data = new T[size]();

}

T& operator[](size\_t index) { // Subscript operator to access elements by index

return data[index];

}

void printAll() const { // Function to print all elements in the container

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

cout << data[i] << " ";

}

cout << endl;

}

};

int main() {

DataContainer<int> intContainer(5);

for (size\_t i = 0; i < 5; ++i) {

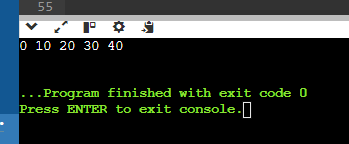
intContainer[i] = i \* 10;

} intContainer.printAll();

return 0;

}

OUTPUT:



1. 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).

A: #include <iostream>

using namespace std;

template<typename T, size\_t N>

struct DataContainer {

T data[N];

DataContainer(initializer\_list<T> init) { // Constructor

size\_t i = 0;

for (const auto& item : init) {

data[i++] = item;

}

}

void print() const { // Function to print the container

for (const auto& elem : data) {

cout << elem << " ";

}

cout << endl;

}

};

template<typename T, size\_t N>

void swap(DataContainer<T, N>& first, DataContainer<T, N>& second) {

for (size\_t i = 0; i < N; ++i) { // Function template to swap two DataContainer

T temp = first.data[i];

first.data[i] = second.data[i];

second.data[i] = temp;

}

} int main() {

DataContainer<int, 3> container1{1, 2, 3};

DataContainer<int, 3> container2{4, 5, 6};

cout << "Before swap:" << endl;

cout << "Container 1: ";

container1.print();

cout << "Container 2: ";

container2.print();

swap(container1, container2); // Swap the containers

cout << "After swap:" << endl;

cout << "Container 1: ";

container1.print();

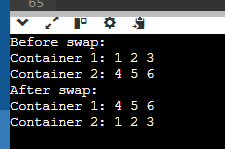
cout << "Container 2: ";

container2.print();

return 0;

}

OUTPUT:



1. 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.

#include <iostream>

#include <string>

using namespace std;

template <typename T> // Template class for DataContainer

class DataContainer {

public:

DataContainer(size\_t size) : size(size), currentIndex(0) {

data = new T[size]; // Constructor

}

void add(const T& item) { // Method to add an item

if (currentIndex < size) {

data[currentIndex++] = item;

} else {

cout << "Container is full!" << endl;

}

}

void printAll() const { // Method to print all items

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

cout << data[i] << " ";

}

cout << endl;

}

void swap(DataContainer<T>& other) {

T\* tempData = data; // Method to swap data

data = other.data;

other.data = tempData;

size\_t tempSize = size;

size = other.size;

other.size = tempSize;

size\_t tempIndex = currentIndex;

currentIndex = other.currentIndex;

other.currentIndex = tempIndex;

}

private:

T\* data;

size\_t size;

size\_t currentIndex;

};

template <typename T> // Swap function

void swap(DataContainer<T>& a, DataContainer<T>& b) {

a.swap(b);

}

int main() {

DataContainer<int> intC1(3);

DataContainer<int> intC2(3);

DataContainer<string> stringC1(3);

DataContainer<string> stringC2(3);

intC1.add(1);

intC1.add(2);

intC1.add(3);

intC2.add(4);

intC2.add(5);

intC2.add(6);

stringC1.add("Hello");

stringC1.add("Friends");

stringC1.add("!");

stringC2.add("Welcome");

stringC2.add("to");

stringC2.add("C++");

cout << "Before swap contents" << endl;

cout << "Integer Container 1: ";

intC1.printAll();

cout << "Integer Container 2: ";

intC2.printAll();

cout << "String Container 1: ";

stringC1.printAll();

cout << "String Container 2: ";

stringC2.printAll();

swap(intC1, intC2);

swap(stringC1, stringC2);

cout << "\nAfter swap contents:" << endl;

cout << "Integer Container 1: ";

intC1.printAll();

cout << "Integer Container 2: ";

intC2.printAll();

cout << "String Container 1: ";

stringC1.printAll();

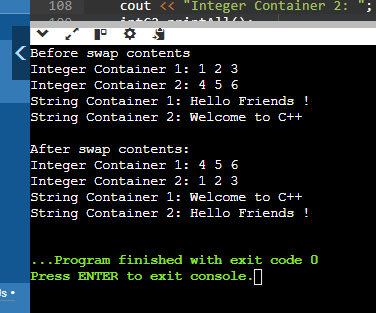
cout << "String Container 2: ";

stringC2.printAll();

return 0;

}

OUTPUT:



1. 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).

A: #include <iostream>

using namespace std;

template <typename T>

class DataContainer {

private:

T\* data;

size\_t current\_size;

size\_t current\_capacity;

void resize(size\_t new\_capacity) {

T\* new\_data = new T[new\_capacity];

for (size\_t i = 0; i < current\_size; ++i) {

new\_data[i] = data[i];

}

delete[] data;

data = new\_data;

current\_capacity = new\_capacity;

}

public:

DataContainer(size\_t initial\_size = 0) // Constructor with optional initial size

: data(new T[initial\_size]), current\_size(0), current\_capacity(initial\_size) {}

size\_t size() const {

return current\_size;

}

void push\_back(const T& value) { // Appends an element to the back of the container

if (current\_size == current\_capacity) {

size\_t new\_capacity = (current\_capacity == 0) ? 1 : current\_capacity \* 2;

resize(new\_capacity);

}

data[current\_size++] = value;

}

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

return data[index];

} };

int main() {

DataContainer<int> container(2);

container.push\_back(1);

container.push\_back(2);

container.push\_back(3);

cout << "Container size: " << container.size() << endl;

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

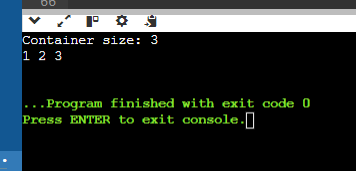
cout << container[i] << " ";

}

cout << endl;

return 0; }

OUTPUT:



TASK – 3

Steps to Implement

1. Define the Abstract Class:

o Create an abstract class Animal with a pure virtual function makeSound().

2. Define Concrete Classes:

o Create a Dog class that inherits from Animal and implements the makeSound() function.

o Create a Cat class that inherits from Animal and implements the makeSound() function.

3. Demonstrate Usage:

o Write a main function to create instances of Dog and Cat.

o Call the makeSound() function on these instances to demonstrate polymorphism.

#include <iostream>

#include <vector>

using namespace std;

// Abstract class Animal

class Animal {

public:

// Pure virtual function

virtual void makeSound() const = 0;

// Virtual destructor

virtual ~Animal() {}

};

// Concrete class Dog

class Dog : public Animal {

public:

void makeSound() const override {

cout << "Woof! Woof!" << endl;

}

};

// Concrete class Cat

class Cat : public Animal {

public:

void makeSound() const override {

cout << "Meow! Meow!" << endl;

}

};

int main() {

// Create instances of Dog and Cat

Animal\* dog = new Dog();

Animal\* cat = new Cat();

// Store the instances in a vector of Animal pointers

vector<Animal\*> animals;

animals.push\_back(dog);

animals.push\_back(cat);

// Call makeSound on each Animal

for (const auto& animal : animals) {

animal->makeSound();

}

// Clean up

for (auto& animal : animals) {

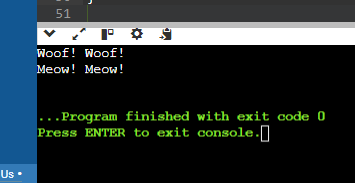
delete animal;

}

return 0;

}

OUTPUT:



2. Use abstract classes and polymorphism in C++ for calculating the areas of various shapes

#include <iostream>

#include <cmath> // for M\_PI

using namespace std;

// Abstract base class Shape

class Shape {

public:

// Pure virtual function to calculate area

virtual double calculateArea() const = 0;

// Virtual destructor

virtual ~Shape() {}

};

// Concrete class Circle

class Circle : public Shape {

private:

double radius;

public:

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

// Override calculateArea() to compute area of circle

double calculateArea() const override {

return M\_PI \* radius \* radius;

}

};

// Concrete class Rectangle

class Rectangle : public Shape {

private:

double width;

double height;

public:

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

// Override calculateArea() to compute area of rectangle

double calculateArea() const override {

return width \* height;

}

};

// Concrete class Triangle

class Triangle : public Shape {

private:

double base;

double height;

public:

Triangle(double b, double h) : base(b), height(h) {}

// Override calculateArea() to compute area of triangle

double calculateArea() const override {

return 0.5 \* base \* height;

}

};

int main() {

// Create instances of Circle, Rectangle, and Triangle

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 7.0);

// Store them in an array of Shape pointers

Shape\* shapes[] = { &circle, &rectangle, &triangle };

// Calculate and print areas using polymorphism

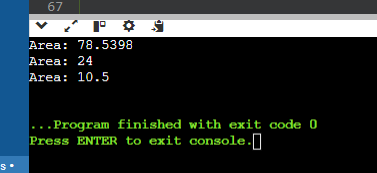
for (auto shape : shapes) {

cout << "Area: " << shape->calculateArea() << endl;

}

return 0;

}



-------------------------------------------------------------------------------------------------------------------------------

**Smart Pointer**

#include <iostream>

using namespace std;

class SmartPtr {

int\* ptr;

public:

// Create an explicit constructor

explicit SmartPtr(int\* p = NULL) { ptr = p; }

// Destructor to deallocate the resource used

~SmartPtr() { delete (ptr); }

// Overloading dereferencing operator

int& operator\*() { return \*ptr; }

};

int main()

{

SmartPtr ptr(new int());

\*ptr = 100;

cout << \*ptr;

return 0;

}

OUTPUT:

