**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, 11, 4, 1, 5, 9, 10, 6, 5 };

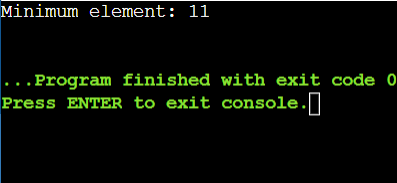
int n = sizeof(arr) / sizeof(arr[0]);

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 = "Kavya";

string str2 = "Hari";

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

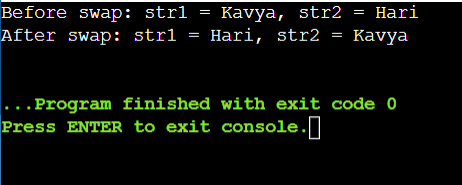
swap(&str1, &str2);

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};

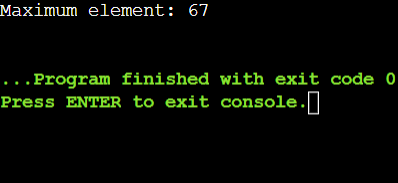
int n = sizeof(arr) / sizeof(arr[0]);

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

return 0;

}

**Output:**



**Restriction of Generic Functions:**

#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()

{

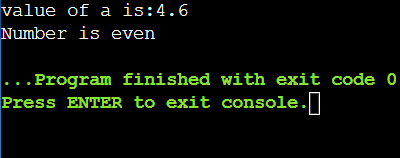
fun(4.6);

fun(6);

return 0;

}

**Output:**



**Class Template:**

#include <iostream>

using namespace std;

template<class T>

class A{

public:

T num1 = 5;

T num2 = 6;

void add()

{

cout<<"Addition of num1 and num2:"<<num1+num2<<endl;

}

};

int main()

{

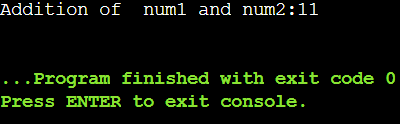
A<int>d;

d.add();

return 0;

}

**Output:**



**Class Template with 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()

{

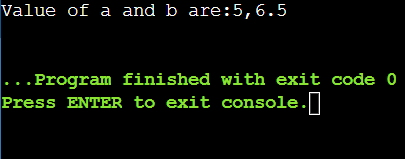
A<int,float>d(5,6.5);

d.display();

return 0;

}

**Output:**



**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.**

**1.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.**

#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]();

}

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

return data[index];

}

void printAll() const {

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;

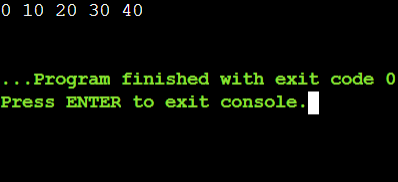
}

intContainer.printAll();

return 0;

}

**Output:**



**2.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).**

#include <iostream>

using namespace std;

template<typename T, size\_t N>

struct DataContainer {

T data[N];

DataContainer(initializer\_list<T> init) {

size\_t i = 0;

for (const auto& item : init) {

data[i++] = item;

}

}

void print() const {

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) {

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);

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

cout << "Container 1: ";

container1.print();

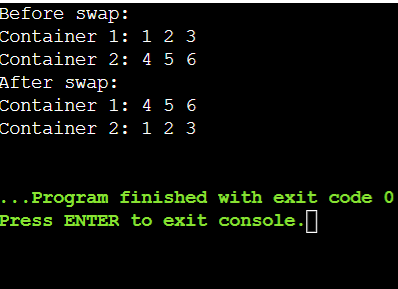
cout << "Container 2: ";

container2.print();

return 0;

}

**Output:**



**3.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>

class DataContainer {

public:

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

data = new T[size];

}

void add(const T& item) {

if (currentIndex < size) {

data[currentIndex++] = item;

} else {

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

}

}

void printAll() const {

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

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

}

cout << endl;

}

void swap(DataContainer<T>& other) {

T\* tempData = 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>

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();

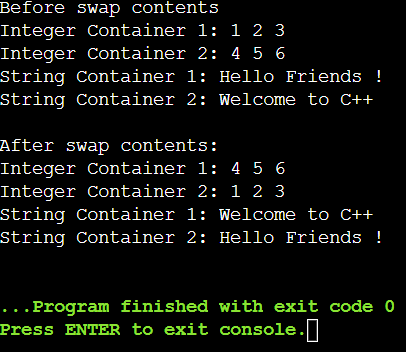
cout << "String Container 2: ";

stringC2.printAll();

return 0;

}

**Output:**



**4.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).**

#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)

: 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) {

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) {

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] << " ";

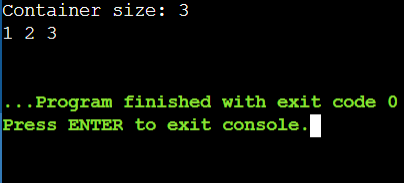
}

cout << endl;

return 0;

}

**Output:**



**5.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 Pointers:**

#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());

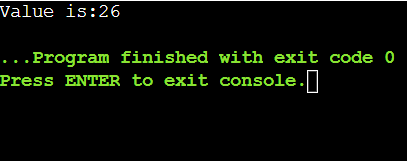
\*p = 26;

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

return 0;

}

**Output:**



**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.**

**Steps to Implement**

1. Define the Abstract Class:

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

2. Define Concrete Classes:

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

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

3. Demonstrate Usage:

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

#include <iostream>

using namespace std;

class Animal {

public:

virtual void makeSound() = 0; // Pure virtual function

virtual ~Animal() {} // Virtual destructor

};

class Dog : public Animal {

public:

void makeSound()override {

cout << "Woof!" << endl;

}

};

class Cat : public Animal {

public:

void makeSound()override {

cout << "Meow!" << endl;

}

};

void playSound(Animal& animal) {

animal.makeSound();

}

int main() {

Dog dog;

Cat cat;

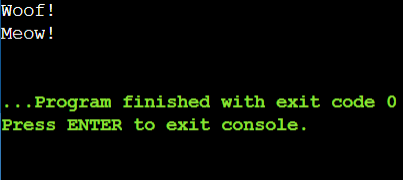
playSound(dog);

playSound(cat);

return 0;

}

**Output:**



**use abstract classes and polymorphism in C++ for calculating the areas of various shapes**

#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

// Abstract class Shape

class Shape {

public:

virtual float area() const = 0; // Pure virtual function

virtual ~Shape() {} // Virtual destructor

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

float width, height;

public:

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

float area() const override {

return width \* height;

}

};

void printShapeAreas(const vector<Shape\*>& shapes) {

for (auto shape : shapes) {

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

}

}

int main() {

Shape\* shapes[] = {

new Circle(7.0f),

new Rectangle(8.0f, 9.0f)

};

printShapeAreas(vector<Shape\*>(shapes, shapes + sizeof(shapes) / sizeof(Shape\*)));

for (auto shape : shapes) {

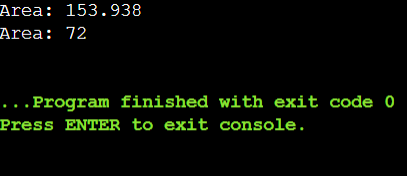
delete shape;

}

return 0;

}

**Output:**



**Smart Pointers:**

#include <iostream>

using namespace std;

class SmartPtr {

int\* ptr;

public:

explicit SmartPtr(int\* p = NULL) {

ptr = p;

}

~SmartPtr() { delete (ptr); }

int& operator\*() {

return \*ptr;

}

};

int main()

{

SmartPtr ptr(new int());

\*ptr = 100;

cout << \*ptr;

return 0;

}

**Output:**

