**Date:03/07/2024**

**Day\_8\_Assignments**

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;

template <typename T>

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

T temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

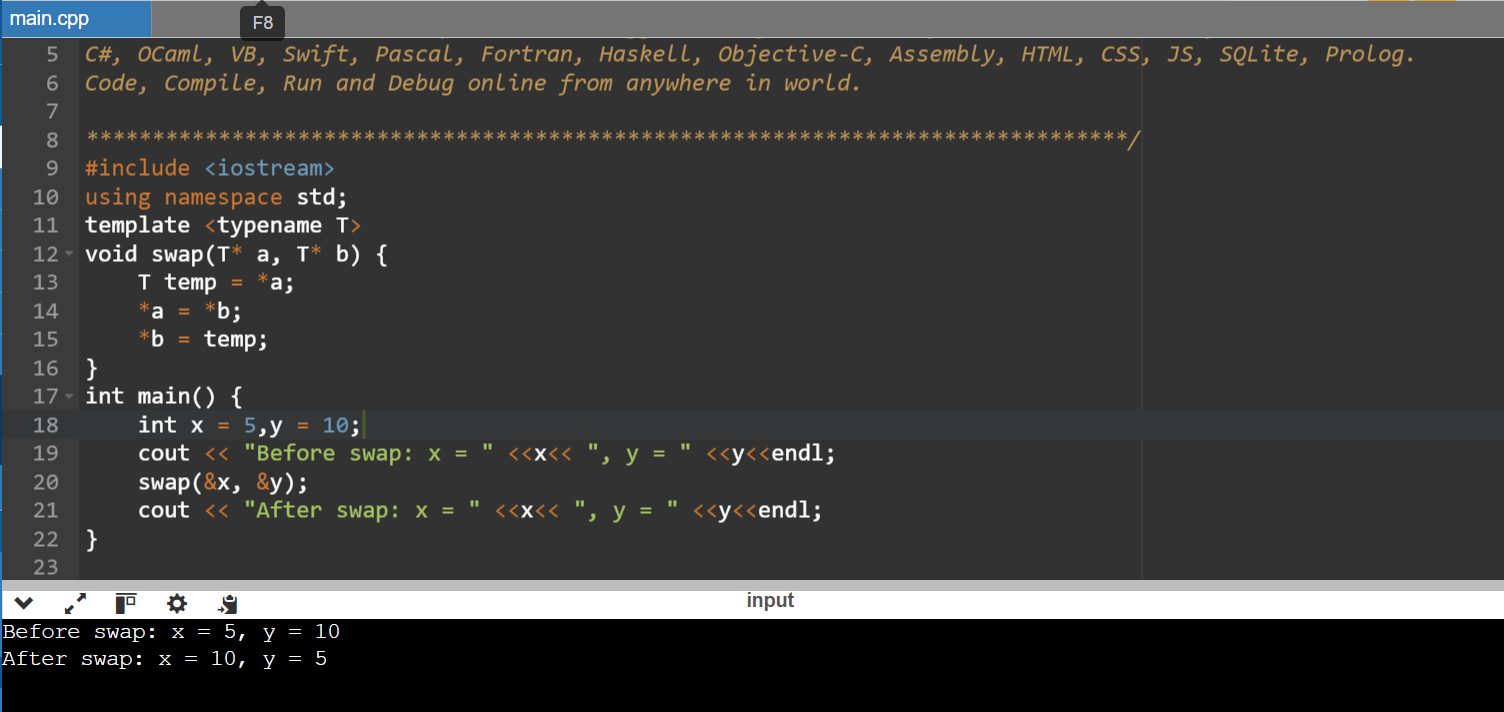
int x = 5,y = 10;

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

swap(&x, &y);

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

}



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 intArray[] = {1, 3, 5, 7, 9};

int intSize = sizeof(intArray) / sizeof(intArray[0]);

cout << "Maximum element in array: " << findMax(intArray, intSize)<<endl;

return 0;

}



3.

#include<iostream>

using namespace std;

void fun(double a)

{

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

}

void fun(int b)

{

if(b%2==0)

{

cout<<"number is even";

}

else

{

cout<<"Number is odd";

}

}

int main()

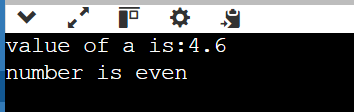
{

fun(4.6);

fun(6);

return 0;

}



4.

#include <iostream>

using namespace std;

template<class T>

class anywhere

{

public:

T num1=5;

T num2=6;

void add()

{

cout<<"Addition of num 1 and num2:"<<num1 + num2<<endl;

}

};

int main()

{

A<int>d;

d.add;

return 0;

}

5.

#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<<"values of a and b are: "<<a<<b<<endl;

}

};

int main()

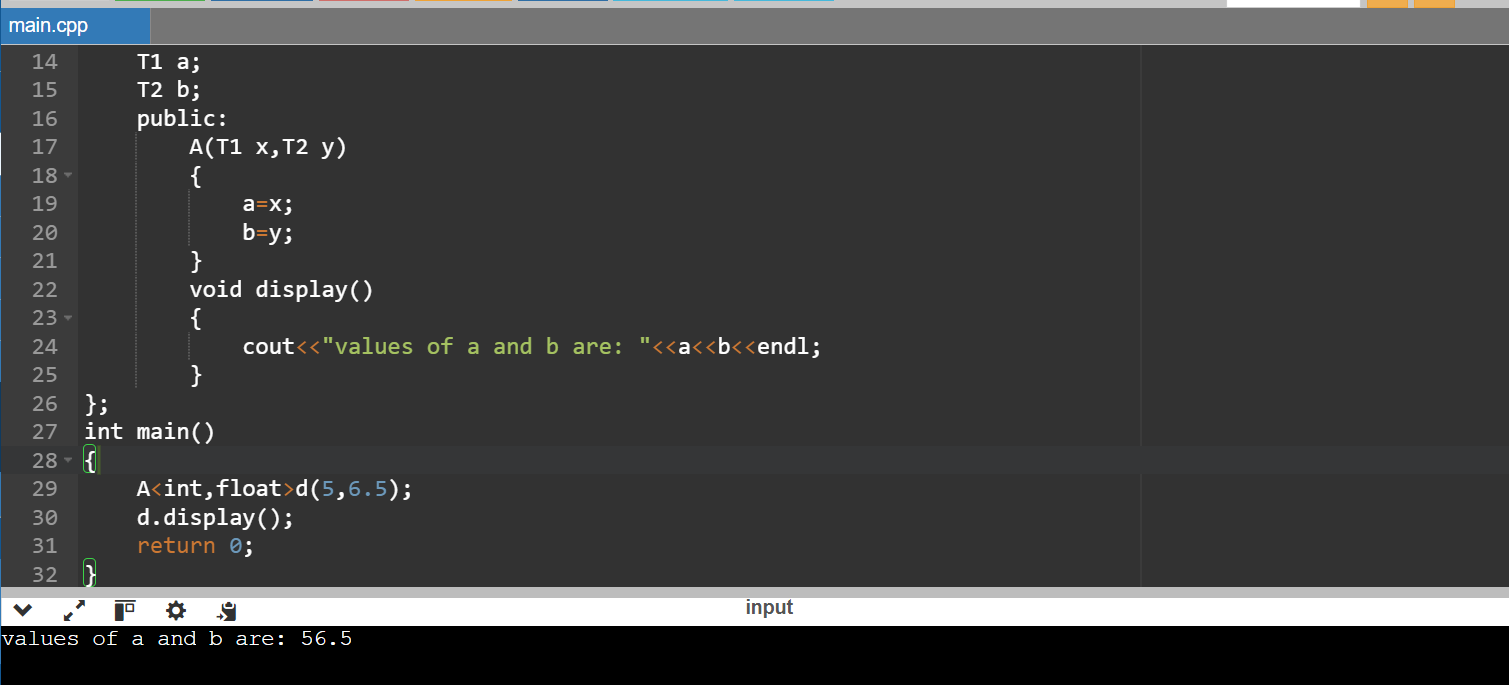
{

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

d.display();

return 0;

}



6.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>

using namespace std;

template <typename T>

class DataContainer {

private:

vector<T> data;

public:

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

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

return data[index];

}

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

return data[index];

}

void printAll() const {

for (const auto& element : data) {

cout << element << " ";

}

cout << endl;

}

template <typename U>

friend void swap(DataContainer<U>& first, DataContainer<U>& second);

};

template <typename T>

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

using std::swap;

swap(first.data, second.data);

}

int main() {

DataContainer<int>Container1(2);

Container1[0] = 10;

Container1[1] = 20;

DataContainer<int> Container2(2);

Container2[0] = 100;

Container2[1] = 200;

cout << "Before swapping:"<< endl;

cout << "Container1: ";

Container1.printAll();

cout << "Container2: ";

Container2.printAll();

swap(Container1,Container2);

cout << "After swapping:" << endl;

cout << "Container1: ";

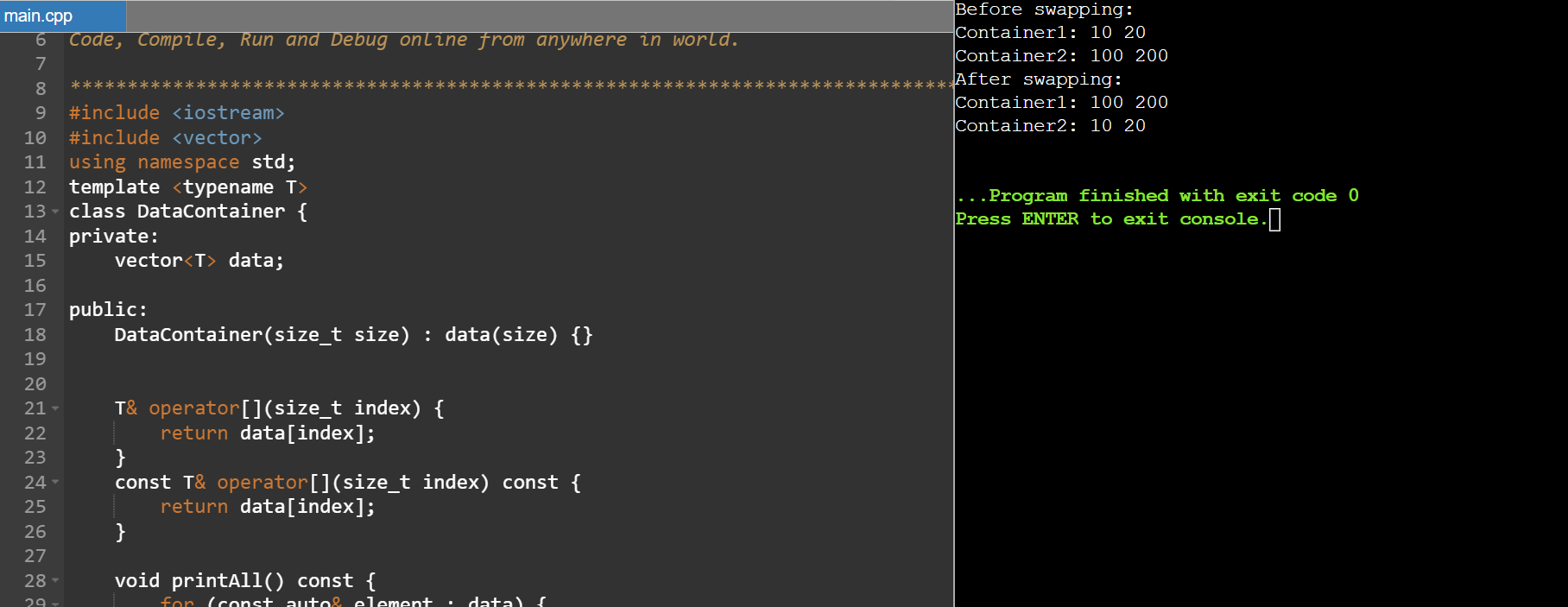
Container1.printAll();

cout << "Container2: ";

Container2.printAll();

return 0;

}



7. 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>

#include <vector>

using namespace std;

template <typename T>

class DataContainer {

private:

vector<T> elements;

public:

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

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

return elements[index];

}

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

return elements[index];

}

void printAll() const {

for (const T& element : elements) {

cout << element << " ";

}

cout << endl;

}

};

int main() {

DataContainer<int> intContainer(5);

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

intContainer[i] = i \* 10;

}

cout << "Integer Container:" << endl;

intContainer.printAll();

DataContainer<double> doubleContainer(3);

doubleContainer[0] = 3.14;

doubleContainer[1] = 1.618;

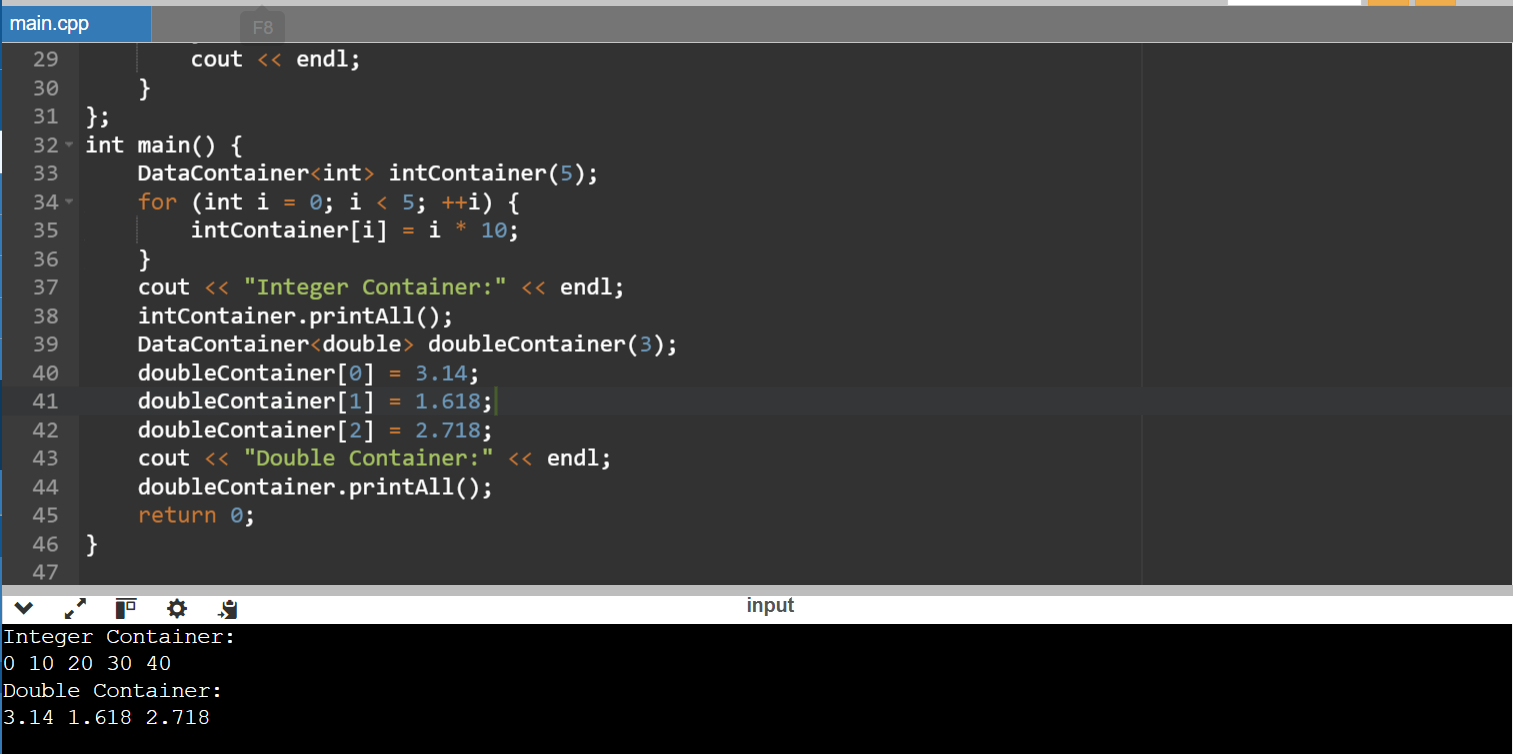
doubleContainer[2] = 2.718;

cout << "Double Container:" << endl;

doubleContainer.printAll();

return 0;

}



8. 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>

#include <vector>

using namespace std;

// Class template for DataContainer

template <typename T>

class DataContainer {

private:

vector<T> elements;

public:

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

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

return elements[index];

}

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

return elements[index];

}

void printAll() const {

for (const T& element : elements) {

cout << element << " ";

}

cout << endl;

}

void swapElements(DataContainer<T>& other) {

size\_t size = min(elements.size(), other.elements.size());

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

swap(elements[i], other.elements[i]);

}

}

};

template <typename T>

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

container1.swapElements(container2);

}

int main() {

DataContainer<int> container1(5);

DataContainer<int> container2(3);

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

container1[i] = i \* 10;

}

for (int i = 0; i < 3; ++i) {

container2[i] = i \* 5;

}

// Printing initial contents of containers

cout << "Before swapping:" << endl;

cout << "Container 1: ";

container1.printAll();

cout << "Container 2: ";

container2.printAll();

// Swapping containers

swapContainers(container1, container2);

// Printing contents after swapping

cout << "After swapping:" << endl;

cout << "Container 1: ";

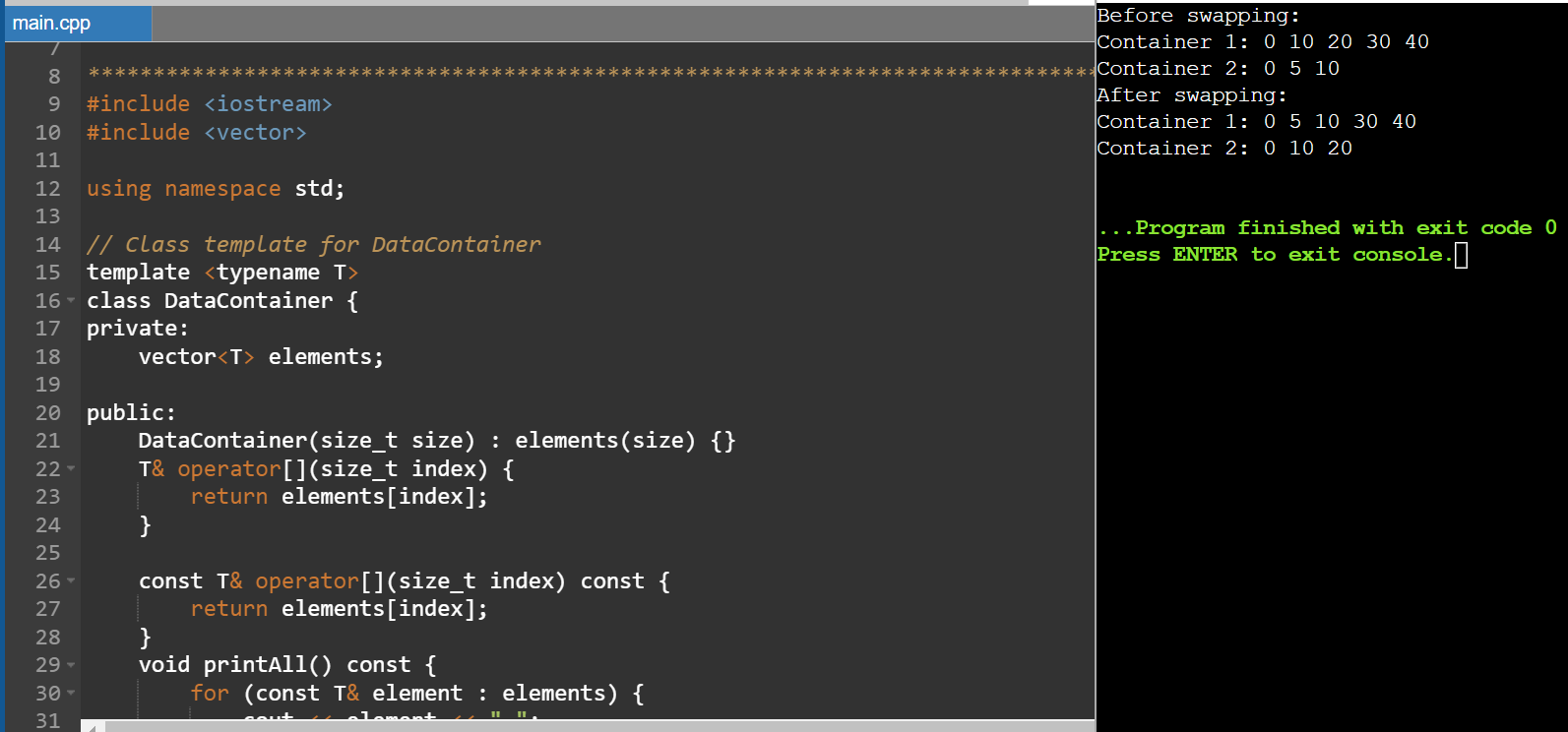
container1.printAll();

cout << "Container 2: ";

container2.printAll();

return 0;

}



9. 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 <vector>

#include <algorithm>

template<typename T>

class DataContainer {

private:

std::vector<T> data;

public:

void addData(const T& value) {

data.push\_back(value);

}

void printAll() const {

std::cout << "DataContainer contents:";

for (const auto& item : data) {

std::cout << " " << item;

}

std::cout << std::endl;

}

void swapData(DataContainer<T>& other) {

std::swap(data, other.data);

}

};

int main() {

// Create instances of DataContainer for different data types

DataContainer<int> intContainer;

DataContainer<float> floatContainer;

DataContainer<std::string> stringContainer;

// Populate the containers with sample data

intContainer.addData(10);

intContainer.addData(20);

floatContainer.addData(3.14f);

floatContainer.addData(1.618f);

stringContainer.addData("Hello");

stringContainer.addData("World");

// Print the contents of each container

intContainer.printAll();

floatContainer.printAll();

stringContainer.printAll();

// Swap elements between containers of the same type

intContainer.swapData(intContainer); // Swap with itself for demonstration

floatContainer.swapData(floatContainer); // Swap with itself for demonstration

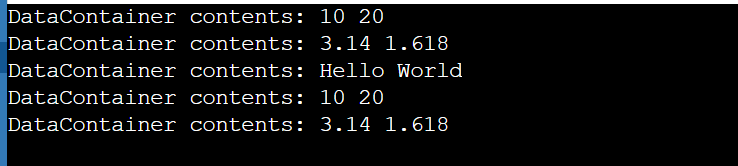
// Print the containers again to confirm the swap

intContainer.printAll();

floatContainer.printAll();

return 0;

}



10. 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):

#include <iostream>

#include <stdexcept>

template <typename T>

class MyContainer {

private:

T\* elements;

int capacity;

int current\_size;

public:

MyContainer(int initial\_size = 0) : capacity(initial\_size), current\_size(0) {

elements = new T[capacity];

}

// Destructor

~MyContainer() {

delete[] elements;

}

int size() const {

return current\_size;

}

void push\_back(const T& value) {

if (current\_size >= capacity) {

// Double the capacity if necessary

int new\_capacity = (capacity == 0) ? 1 : capacity \* 2;

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

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

new\_elements[i] = elements[i];

}

delete[] elements;

elements = new\_elements;

capacity = new\_capacity;

}

elements[current\_size++] = value;

}

};

int main() {

// Example usage

MyContainer<int> container(5); // Initializes with capacity for 5 elements

container.push\_back(10);

container.push\_back(20);

container.push\_back(30);

std::cout << "Current size: " << container.size() << std::endl; // Output: 3

container.push\_back(40);

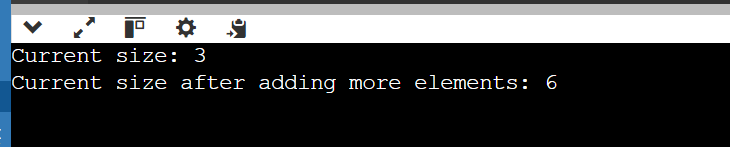
container.push\_back(50);

container.push\_back(60);

std::cout << "Current size after adding more elements: " << container.size() << std::endl; // Output: 6

return 0;

}



11. 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).

#include <iostream>

#include <vector>

template<typename T>

class Node {

public:

T data;

Node<T>\* next;

Node(const T& data) : data(data), next(nullptr) {}

};

template<typename T>

class LinkedList {

private:

Node<T>\* head;

Node<T>\* tail;

public:

LinkedList() : head(nullptr), tail(nullptr) {}

~LinkedList() {

Node<T>\* current = head;

while (current != nullptr) {

Node<T>\* next = current->next;

delete current;

current = next;

}

}

void add(const T& value) {

Node<T>\* newNode = new Node<T>(value);

if (head == nullptr) {

head = tail = newNode;

} else {

tail->next = newNode;

tail = newNode;

}

}

void printAll() const {

std::cout << "LinkedList contents:";

Node<T>\* current = head;

while (current != nullptr) {

std::cout << " " << current->data;

current = current->next;

}

std::cout << std::endl;

}

};

int main() {

LinkedList<int> intList;

LinkedList<float> floatList;

LinkedList<std::string> stringList;

intList.add(10);

intList.add(20);

floatList.add(3.14f);

floatList.add(1.618f);

stringList.add("Hello");

stringList.add("World");

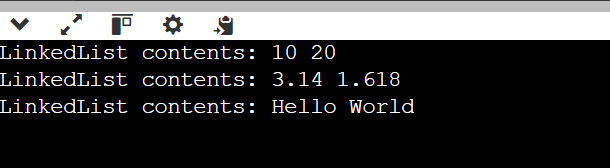
intList.printAll();

floatList.printAll();

stringList.printAll();

return 0;

}



12.

#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()

{

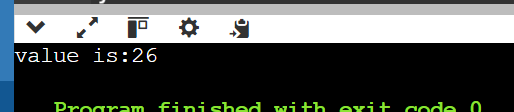
Smartpointer<int>p(new int());

\*p=26;

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

return 0;

}



13.

#include <iostream>

#include <cmath>

// Abstract class Shape

class Shape {

public:

virtual double area() const = 0;

virtual double perimeter() const = 0;

// Virtual destructor

virtual ~Shape() = default;

};

// Circle class derived from Shape

class Circle : public Shape {

private:

double radius;

public:

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

// Implement area for Circle

double area() const override {

return M\_PI \* radius \* radius;

}

// Implement perimeter for Circle

double perimeter() const override {

return 2 \* M\_PI \* radius;

}

};

// Rectangle class derived from Shape

class Rectangle : public Shape {

private:

double width;

double height;

public:

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

// Implement area for Rectangle

double area() const override {

return width \* height;

}

// Implement perimeter for Rectangle

double perimeter() const override {

return 2 \* (width + height);

}

};

// Function to print shape information

void printShapeInfo(const Shape& shape) {

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

std::cout << "Perimeter: " << shape.perimeter() << std::endl;

}

int main() {

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

std::cout << "Circle Info:" << std::endl;

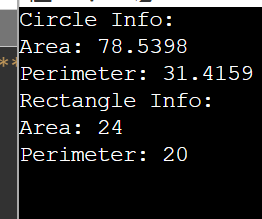
printShapeInfo(circle);

std::cout << "Rectangle Info:" << std::endl;

printShapeInfo(rectangle);

return 0;

}



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

#include <iostream>

#include <cmath>

using namespace std;

// Abstract class Shape

class Shape {

public:

// Pure virtual function for calculating area

virtual double area() const = 0;

// Virtual destructor

virtual ~Shape() = default;

};

// Circle class derived from Shape

class Circle : public Shape {

private:

double radius;

public:

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

// Implement area for Circle

double area() const override {

return M\_PI \* radius \* radius;

}

};

// Rectangle class derived from Shape

class Rectangle : public Shape {

private:

double width;

double height;

public:

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

// Implement area for Rectangle

double area() const override {

return width \* height;

}

};

// Triangle class derived from Shape

class Triangle : public Shape {

private:

double base;

double height;

public:

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

// Implement area for Triangle

double area() const override {

return 0.5 \* base \* height;

}

};

// Function to print shape information

void display(const Shape& shape) {

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

}

int main() {

// Create instances of different shapes

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 4.0);

// Print the area of each shape

cout << "Circle:" <<endl;

display(circle);

cout << "Rectangle:" <<endl;

display(rectangle);

cout << "Triangle:" <<endl;

display(triangle);

return 0;

}

