**MOUNIKA(DAY8)**

**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:

Minimum element: 9

**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;

double m = 1.2, n = 3.4;

char c1 = 'A', c2 = 'B';

// Print initial values

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

cout << "x = " << x << ", y = " << y << endl;

cout << "m = " << m << ", n = " << n << endl;

cout << "c1 = " << c1 << ", c2 = " << c2 << endl;

// Swap values

swap(&x, &y);

swap(&m, &n);

swap(&c1, &c2);

// Print swapped values

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

cout << "x = " << x << ", y = " << y << endl;

cout << "m = " << m << ", n = " << n << endl;

cout << "c1 = " << c1 << ", c2 = " << c2 << endl;

return 0;

}

OUTPUT:

Before swapping:

x = 5, y = 10

m = 1.2, n = 3.4

c1 = A, c2 = B

After swapping:

x = 10, y = 5

m = 3.4, n = 1.2

c1 = B, c2 = A

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

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

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

return 0;

}

OUTPUT:

Maximum element: 9

**RESTRICTIONS OF GENERIC FUNCTION**

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities.

#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:

value of a is :4.6

Number is even

**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:

addition of num1 and num2:11

**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()

{

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

}

};

int main()

{

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

d.display();

return 0;

}

OUTPUT:

values of a and b are:5,6.5

**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.**

#include <iostream>

#include <vector>

using namespace std;

template <typename T>

class DataContainer {

public:

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

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

return data[index];

}

void printAll() const {

for (const auto& element : data) {

cout << element << " ";

}

cout << endl;

}

private:

vector<T> data;

};

int main() {

DataContainer<int> intContainer(5);

DataContainer<double> doubleContainer(3);

DataContainer<string> stringContainer(4);

intContainer[0] = 1;

intContainer[1] = 2;

intContainer[2] = 3;

intContainer[3] = 4;

intContainer[4] = 5;

// Initialize doubleContainer

doubleContainer[0] = 1.1;

doubleContainer[1] = 2.2;

doubleContainer[2] = 3.3;

// Initialize stringContainer

stringContainer[0] = "Hello";

stringContainer[1] = "World";

stringContainer[2] = "C++";

stringContainer[3] = "Templates";

// Print all elements

cout << "intContainer elements: ";

intContainer.printAll();

cout << "doubleContainer elements: ";

doubleContainer.printAll();

cout << "stringContainer elements: ";

stringContainer.printAll();

return 0;

}

OUTPUT:

intContainer elements: 1 2 3 4 5

doubleContainer elements: 1.1 2.2 3.3

stringContainer elements: Hello World C++ Templates

**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:**

#include <iostream>

#include <vector>

#include <stdexcept>

using namespace std;

template <typename T>

class DataContainer {

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) {

if (index >= data.size()) {

throw out\_of\_range("Index out of range");

}

return data[index];

}

// Function to print all elements of the container

void printAll() const {

for (const auto& element : data) {

cout << element << " ";

}

cout << endl;

}

// Function to swap elements with another DataContainer

void swap(DataContainer<T>& other) {

// Swap sizes to handle containers of different sizes

size\_t minSize = min(data.size(), other.data.size());

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

std::swap(data[i], other.data[i]);

}

}

private:

vector<T> data; // Internal vector to store elements

};

// Function template to swap two DataContainer objects

template <typename T>

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

a.swap(b);

}

int main() {

// Create instances of DataContainer for different data types

DataContainer<int> intContainer1(2);

DataContainer<int> intContainer2(2);

DataContainer<double> doubleContainer1(2);

DataContainer<double> doubleContainer2(2);

// Initialize elements in intContainer1

intContainer1[0] = 1;

intContainer1[1] = 2;

// Initialize elements in intContainer2

intContainer2[0] = 6;

intContainer2[1] = 7;

// Initialize elements in doubleContainer1

doubleContainer1[0] = 1.1;

doubleContainer1[1] = 2.2;

// Initialize elements in doubleContainer2

doubleContainer2[0] = 4.4;

doubleContainer2[1] = 5.5;

// Print initial values

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

cout << "intContainer1 elements: "; intContainer1.printAll();

cout << "intContainer2 elements: "; intContainer2.printAll();

cout << "doubleContainer1 elements: "; doubleContainer1.printAll();

cout << "doubleContainer2 elements: "; doubleContainer2.printAll();

// Swap DataContainer objects

swap(intContainer1, intContainer2);

swap(doubleContainer1, doubleContainer2);

// Print swapped values

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

cout << "intContainer1 elements: "; intContainer1.printAll();

cout << "intContainer2 elements: "; intContainer2.printAll();

cout << "doubleContainer1 elements: "; doubleContainer1.printAll();

cout << "doubleContainer2 elements: "; doubleContainer2.printAll();

return 0;

}

OUTPUT:

Before swapping:

intContainer1 elements: 1 2

intContainer2 elements: 6 7

doubleContainer1 elements: 1.1 2.2

doubleContainer2 elements: 4.4 5.5

After swapping:

intContainer1 elements: 6 7

intContainer2 elements: 1 2

doubleContainer1 elements: 4.4 5.5

doubleContainer2 elements: 1.1 2.2

**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: in very simple and short code**

#include <iostream>

#include <vector>

// Define the DataContainer class template

template <typename T>

class DataContainer {

private:

std::vector<T> data;

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 of the container

void printAll() const {

for (const auto& element : data) {

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

}

std::cout << std::endl;

}

// Function to get the size of the container

size\_t size() const {

return data.size();

}

};

// Swap function template

template <typename T>

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

if (a.size() != b.size()) {

throw std::invalid\_argument("Containers must be of the same size to swap.");

}

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

std::swap(a[i], b[i]);

}

}

int main() {

DataContainer<int> intContainer(2);

DataContainer<int> intContainer2(2);

// Initialize the containers

for (size\_t i = 0; i < 2; ++i) {

intContainer[i] = i;

intContainer2[i] = i + 10;

}

// Print original containers

std::cout << "Original intContainer: ";

intContainer.printAll();

std::cout << "Original intContainer2: ";

intContainer2.printAll();

// Swap the contents of the containers

swap(intContainer, intContainer2);

// Print swapped containers

std::cout << "Swapped intContainer: ";

intContainer.printAll();

std::cout << "Swapped intContainer2: ";

intContainer2.printAll();

return 0;

}

OUTPUT:

Original intContainer: 0 1

Original intContainer2: 10 11

Swapped intContainer: 10 11

Swapped intContainer2: 0 1

**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 <stdexcept>

#include <string>

#include <algorithm>

using namespace std;

template <typename T>

class Dc {

private:

vector<T> data;

public:

Dc(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;

}

size\_t size() const {

return data.size();

}

};

template <typename T>

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

if (a.size() != b.size()) {

throw std::invalid\_argument("Containers must be of the same size to swap.");

}

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

std::swap(a[i], b[i]);

}

}

int main() {

Dc<int> intContainer(2);

Dc<int> intContainer2(2);

Dc<float> floatContainer(2);

Dc<std::string> stringContainer(2);

for (size\_t i = 0; i < 2; ++i) {

intContainer[i] = i;

intContainer2[i] = i + 10;

}

floatContainer[0] = 1.1f;

floatContainer[1] = 2.2f;

stringContainer[0] = "Hello";

stringContainer[1] = "world";

std::cout << "intContainer: ";

intContainer.printAll();

cout << "intContainer2: ";

intContainer2.printAll();

cout << "floatContainer: ";

floatContainer.printAll();

cout << "stringContainer: ";

stringContainer.printAll();

swap(intContainer, intContainer2);

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

cout << "intContainer: ";

intContainer.printAll();

cout << "intContainer2: ";

intContainer2.printAll();

return 0;

}

OUTPUT:

intContainer: 0 1

intContainer2: 10 11

floatContainer: 1.1 2.2

stringContainer: Hello world

After swapping:

intContainer: 10 11

intContainer2: 0 1

**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 <vector>

#include <stdexcept>

#include <string>

#include <algorithm>

using namespace std;

template <typename T>

class DataContainer {

private:

vector<T> data;

public:

DataContainer(size\_t size = 0) : data(size) {}

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

return data.at(index);

}

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

return data.at(index);

}

void printAll() const {

for (const auto& element : data) {

cout << element << " ";

}

cout << endl;

}

size\_t size() const {

return data.size();

}

void push\_back(const T& value) {

data.push\_back(value);

}

};

template <typename T>

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

if (a.size() != b.size()) {

throw invalid\_argument("Containers must be of the same size to swap.");

}

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

std::swap(a[i], b[i]);

}

}

int main() {

DataContainer<int> intContainer;

DataContainer<int> intContainer2;

DataContainer<float> floatContainer;

DataContainer<string> stringContainer;

// Populate the containers with sample data

for (size\_t i = 0; i < 2; ++i) {

intContainer.push\_back(i);

intContainer2.push\_back(i + 10);

}

floatContainer.push\_back(1.1f);

floatContainer.push\_back(2.2f);

stringContainer.push\_back("Hello");

stringContainer.push\_back("world");

cout << "intContainer: ";

intContainer.printAll();

cout << "intContainer2: ";

intContainer2.printAll();

cout << "floatContainer: ";

floatContainer.printAll();

cout << "stringContainer: ";

stringContainer.printAll();

swap(intContainer, intContainer2);

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

cout << "intContainer: ";

intContainer.printAll();

cout << "intContainer2: ";

intContainer2.printAll();

return 0;

}

OUTPUT:

intContainer: 0 1

intContainer2: 10 11

floatContainer: 1.1 2.2

stringContainer: Hello world

After swapping:

intContainer: 10 11

intContainer2: 0 1

**CLASS TEMPLATE**

#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:

value is:26

1.Abstract class

2. Interface

An Abstract class is a class that contains at least one pure virtual function.

Interface in C++ is implemented using an abstract class with only pure virtual functions

Rules of Abstract Class

1) As we have seen that any class that has a pure virtual function is an abstract class,

2) We cannot create the instance of abstract class. For example: If I have written this line Animal obj: in the above program, it would have caused compilation error.

3) We can create pointer and reference of base abstract class points to the instance of child class. For example, this is valid:

Animal \*obj = new Dog():

obj->sound();

4) Abstract class can have constructors.

5) If the derived class does not implement the pure virtual function of parent class then the derived class becomes abstract.

**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**

**an abstract class in c++ has at least one pure virtual function by definition. In other words, a function that has no definition. The abstract class's descendants must define the pure virtual function; otherwise, the subclass would become an abstract class in its own right.**

**Abstract classes are used to express broad concepts from which more concrete classes can be derived. An abstract class type object cannot be created. To abstract class types, however, you can use pointers and references. Declare at least one pure virtual member feature when creating an abstract class. The pure specifier (= 0) syntax is used to declare a virtual function.**

**Take a look at the example in virtual functions. The aim of the class is to provide general functionality for shape, but objects of type shape are much too general to be useful. Shape is therefore a suitable candidate for an abstract class:**

**Syntax:**

**Class classname //abstract class**

**{**

**//data members**

**public:**

**//pure virtual function**

**/\* Other members \*/**

};

Abstract classes are a cornerstone of object-oriented programming (OOP), especially in C++, as they help enforce design patterns and promote code reusability. Let's delve into how abstract classes accomplish this and explore a practical example.

Key Concepts

**Abstract Class**: An abstract class cannot be instantiated directly. It is designed to be a base class for other classes. It typically contains one or more pure virtual functions, which must be overridden by derived classes.

**Design Patterns:** Abstract classes are essential in implementing design patterns like Factory Method, Strategy, and Template Method, ensuring a consistent interface across different implementations.

**Code Reusability:** By defining common functionality and interfaces in abstract classes, derived classes can reuse code, reducing duplication and increasing maintainability.

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

#include <iostream>

#include <cmath>

using namespace std;

class Shape {

public:

virtual double calculateArea() const = 0;

virtual ~Shape() {}

};

class Circle : public Shape {

private:

double radius;

public:

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

double calculateArea() const override {

return M\_PI \* radius \* radius;

}

};

class Rectangle : public Shape {

private:

double width;

double height;

public:

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

double calculateArea() const override {

return width \* height;

}

};

class Triangle : public Shape {

private:

double base;

double height;

public:

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

double calculateArea() const override {

return 0.5 \* base \* height;

}

};

int main() {

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 7.0);

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

for (auto shape : shapes) {

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

}

return 0;

}OUTPUT:

Area: 78.5398

Area: 24

Area: 10.5

**SMART POINTERS**

Smart pointer is an abstract data type by using which we can make a normal pointer in such way that it can be used as memory management like file handling, network sockets etc., also it can do many things like automatic destruction, reference counting etc.

Smart pointer in C++, can be implemented as template class, which is overloaded with and operator. auto\_ptr, shared\_ptr, unique\_ptr and weak\_ptr are the forms of smart pointer can be implemented by C++ libraries**.**

#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:100