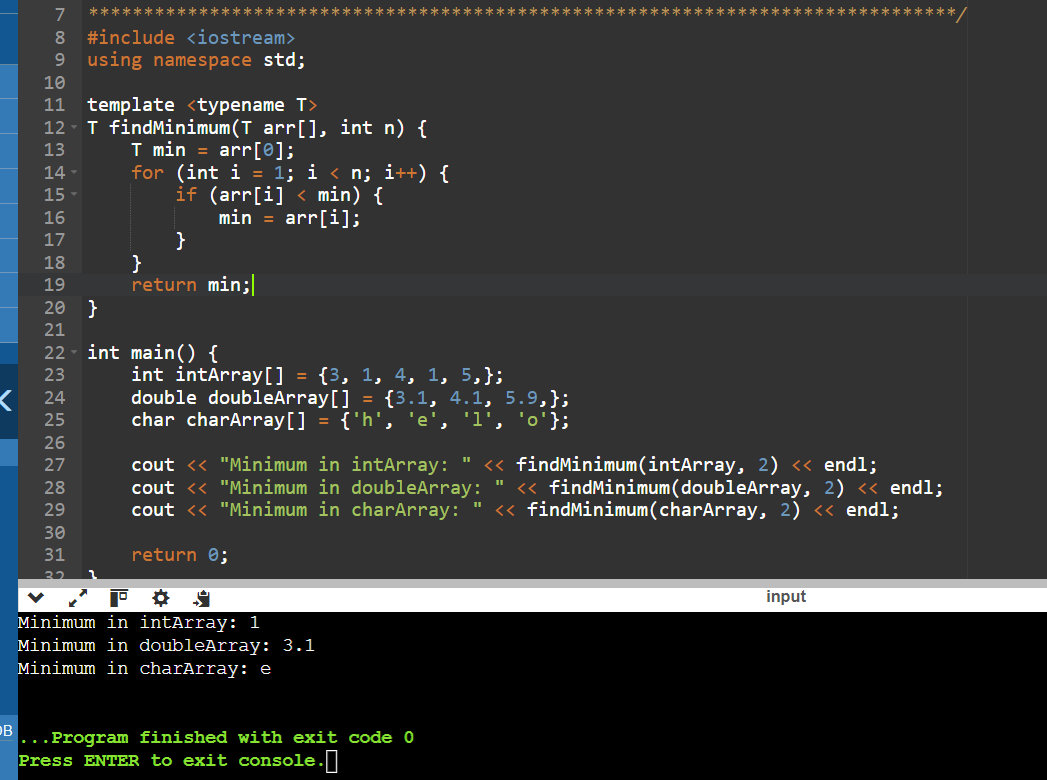
Day-8:

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.



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 < class T>

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

{

T temp;

temp=\*a;

\*a=\*b;

\*b=temp;

}

int main(void)

{

int x,y;

x=100;

y=200;

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

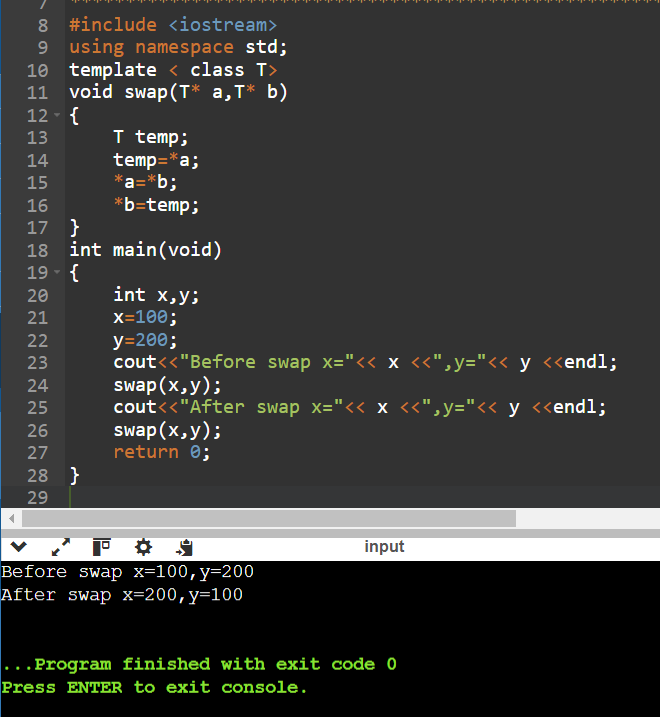
swap(x,y);

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

swap(x,y);

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;

template <typename T>

T findMaximum(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[] = {3, 1, 4, 1, 5, 9};

double doubleArray[] = {3.1, 4.1, 6.9};

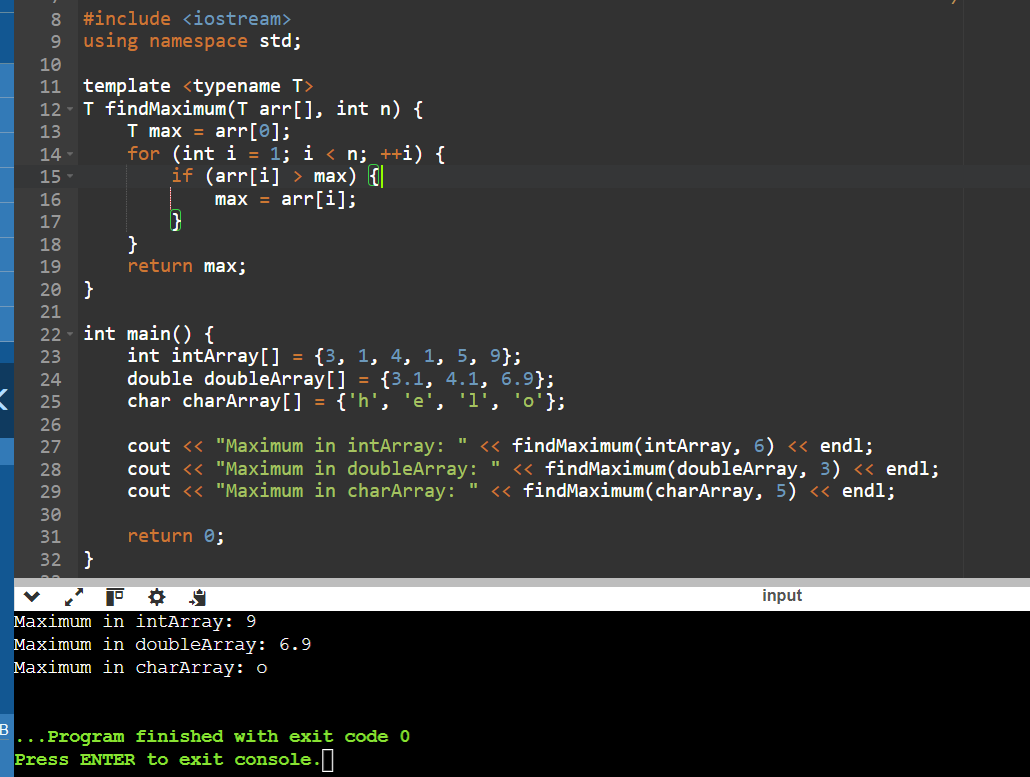
char charArray[] = {'h', 'e', 'l', 'o'};

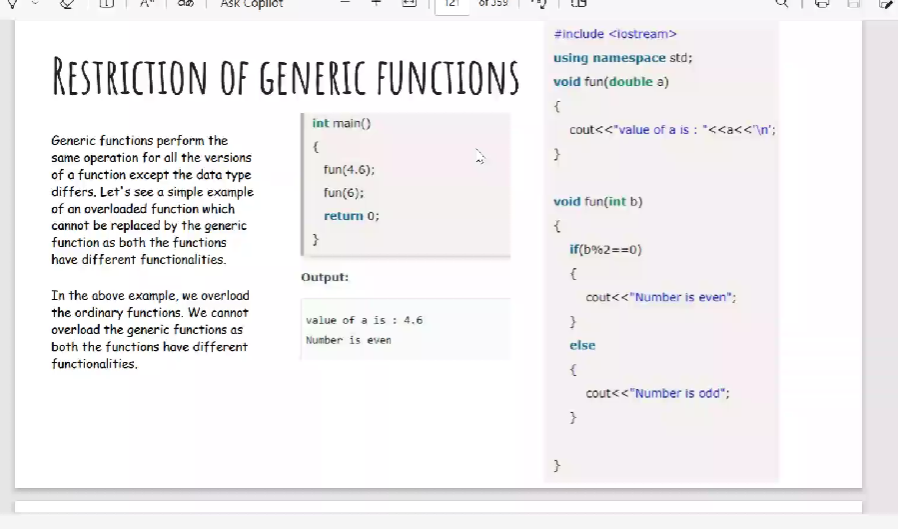
cout << "Maximum in intArray: " << findMaximum(intArray, 5) << endl;

cout << "Maximum in doubleArray: " << findMaximum(doubleArray, 3) << endl;

cout << "Maximum in charArray: " << findMaximum(charArray, 5) << endl;

return 0; }





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 num2:"<<num1+num2<<endl;

}

};

int main()

{

A<int>d;

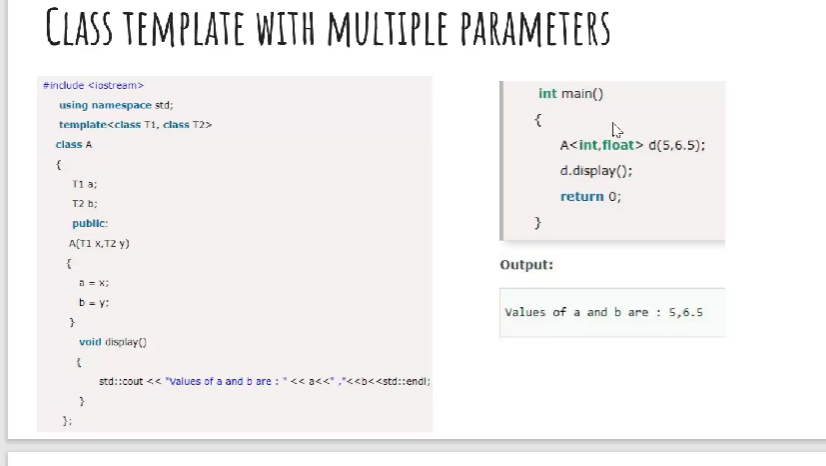
d.add();

return 0;

}

Output:

addition of num1 num2:11



Class Template with Multiple Parameters:

#include <iostream>

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;

}

**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

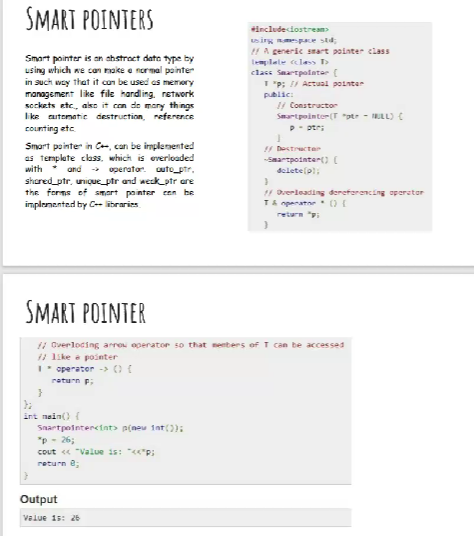
floatContainer: 1.1 2.2

stringContainer: Hello world

After swapping:

intContainer: 10 11

intContainer2: 0 1



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:

value is :26

1.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.

#include <iostream>

#include <cmath> // for M\_PI constant and sqrt function

class Shape {

public:

virtual double area() const = 0;

virtual double perimeter() const = 0;

virtual ~Shape() {}

};

class Circle : public Shape {

private:

double radius;

public:

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

double area() const override {

return M\_PI \* radius \* radius;

}

double perimeter() const override {

return 2 \* M\_PI \* radius;

}

};

class Rectangle : public Shape {

private:

double width, height;

public:

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

double area() const override {

return width \* height;

}

double perimeter() const override {

return 2 \* (width + height);

}

};

class Triangle : public Shape {

private:

double a, b, c;

public:

Triangle(double side1, double side2, double side3) : a(side1), b(side2), c(side3) {}

double area() const override {

double s = (a + b + c) / 2;

return sqrt(s \* (s - a) \* (s - b) \* (s - c));

}

double perimeter() const override {

return a + b + c;

}

};

int main() {

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 4.0, 5.0);

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

for (Shape\* shape : shapes) {

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

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

std::cout << std::endl;

}

return 0; }

Output:

Area: 78.5398

Perimeter: 31.4159

Area: 24

Perimeter: 20

Area: 6

Perimeter: 12