**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 };

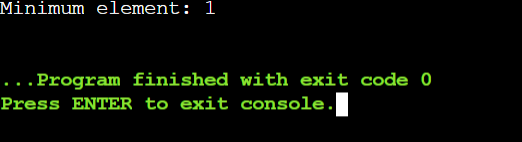
int n = sizeof(arr) / sizeof(arr[0]);

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>

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;

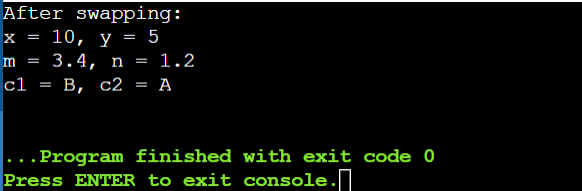
cout << "m = " << m << ", n = " << n << endl;

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

return 0;

}

**Output:**

****

**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 };

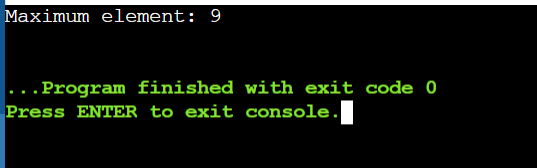
int n = sizeof(arr) / sizeof(arr[0]);

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

return 0;

}

**Output:**

****

**Overload Function Templates:**

#include <iostream>

using namespace std;

void fun(double a)

{

cout <<"value a is : "<<a<<'\n';

}

void fun(int b)

{

if(b%2==0)

{

cout <<"Number is even";

}

else

{

cout <<"Number is odd";

}

}

int main()

{

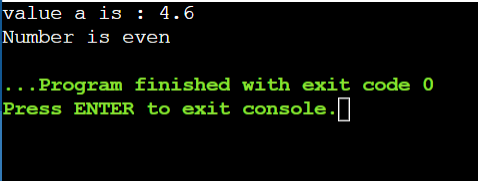
fun(4.6);

fun(6);

return 0;

}

**Output:**

****

**Class Templates:**

#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()

{

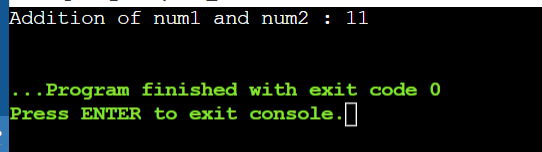
A<int>d;

d.add();

return 0;

}

**Output:**

****

**Class Templates 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 <<"Values of a and b are: "<<a<<","<<b<<endl;

}

};

int main()

{

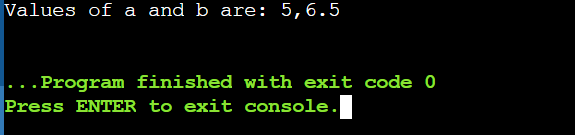
A<int,float>d(5,6.5);

d.display();

return 0;

}

**Output:**

****

**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; // 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) {

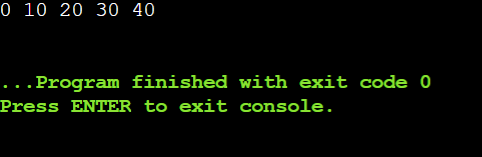
intContainer[i] = i \* 10;

} intContainer.printAll();

return 0;

}

**Output:**

****

**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 Cont {

private:

T\* data;

size\_t size;

public:

// Constructor to initialize the container with a specific size

Cont(size\_t size) : size(size) {

data = new T[size];

}

// Destructor for proper memory management

~Cont() {

delete[] data;

}

// Overloaded subscript operator to access elements

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

return data[index];

}

// Print all elements of the container

void printAll() {

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

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

}

cout << endl;

}

// Function to get the size of the container

size\_t getSize() const {

return size;

}

};

// Function template to swap two Container objects

template <typename T>

void swapConts(Cont<T>& a, Cont<T>& b) {

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

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

}

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

T temp = a[i];

a[i] = b[i];

b[i] = temp;

}

}

int main() {

// Test with int data type

Cont<int> intCont1(5);

Cont<int> intCont2(5);

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

intCont1[i] = i + 1;

intCont2[i] = (i + 1) \* 10;

}

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

cout << "intCont1: ";

intCont1.printAll();

cout << "intCont2: ";

intCont2.printAll();

swapConts(intCont1, intCont2);

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

cout << "intCont1: ";

intCont1.printAll();

cout << "intCont2: ";

intCont2.printAll();

// Test with double data type

Cont<double> doubleCont1(3);

Cont<double> doubleCont2(3);

for (size\_t i = 0; i < 3; ++i) {

doubleCont1[i] = i + 0.1;

doubleCont2[i] = (i + 1) \* 0.5;

}

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

cout << "doubleCont1: ";

doubleCont1.printAll();

cout << "doubleCont2: ";

doubleCont2.printAll();

swapConts(doubleCont1, doubleCont2);

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

cout << "doubleCont1: ";

doubleCont1.printAll();

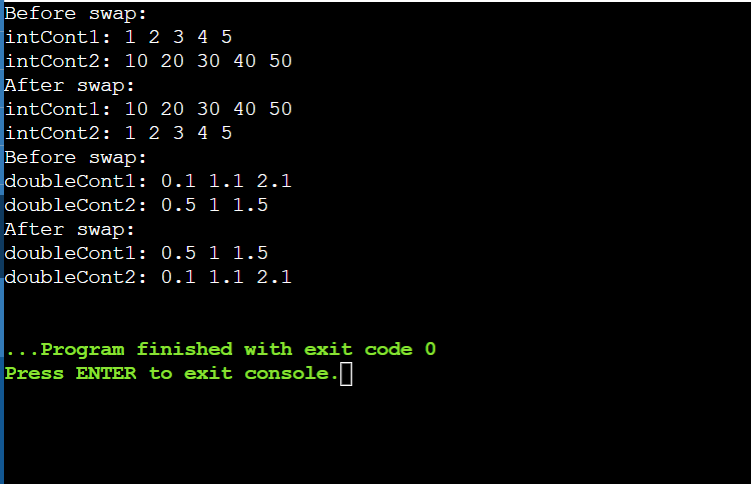
cout << "doubleCont2: ";

doubleCont2.printAll();

return 0;

}

**Output:**



**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) { // 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();

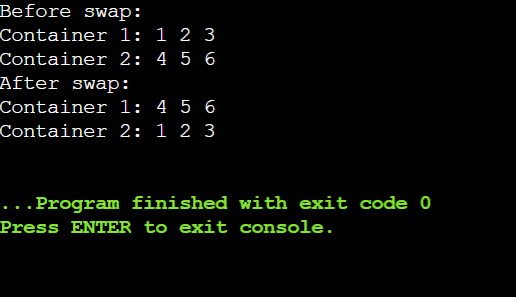
cout << "Container 2: ";

container2.print();

return 0;

}

**Output:**

****

**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 DataCont {

private:

T\* data;

size\_t size;

public:

// Constructor to initialize the container with a specific size

DataCont(size\_t size) : size(size), data(new T[size]) {}

// Destructor for proper memory management

~DataCont() {

delete[] data;

}

// Overloaded subscript operator to access elements

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

return data[index];

}

// Print all elements of the container

void printAll() const {

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

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

}

cout << endl;

}

// Get the size of the container

size\_t getSize() const {

return size;

}

// Swap elements between two containers

friend void swapConts(DataCont& a, DataCont& b) {

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

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

}

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

T temp = a.data[i];

a.data[i] = b.data[i];

b.data[i] = temp;

}

}

};

int main() {

// Create instances of DataContainer for int, float, and string types

DataCont<int> intCont1(5), intCont2(5);

DataCont<float> floatCont1(4), floatCont2(4);

DataCont<string> stringCont1(3), stringCont2(3);

// Populate containers with sample data

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

intCont1[i] = i + 1;

intCont2[i] = (i + 1) \* 10;

}

for (size\_t i = 0; i < 4; ++i) {

floatCont1[i] = i + 0.1f;

floatCont2[i] = (i + 1) \* 0.5f;

}

stringCont1[0] = "Today"; stringCont1[1] = "Training is"; stringCont1[2] = "C++";

stringCont2[0] = "Overload"; stringCont2[1] = "Function"; stringCont2[2] = "Templates";

// Print all containers before swap

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

intCont1.printAll(); intCont2.printAll();

floatCont1.printAll(); floatCont2.printAll();

stringCont1.printAll(); stringCont2.printAll();

// Swap elements between containers of the same type

swapConts(intCont1, intCont2);

swapConts(floatCont1, floatCont2);

swapConts(stringCont1, stringCont2);

// Print all containers after swap

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

intCont1.printAll(); intCont2.printAll();

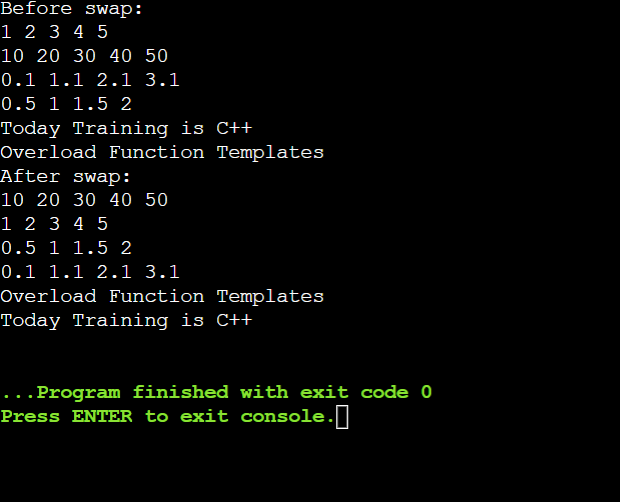
floatCont1.printAll(); floatCont2.printAll();

stringCont1.printAll(); stringCont2.printAll();

return 0;

}

**Output:**

****

**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>

#include <string>

#include <stdexcept>

using namespace std;

template <typename T>

class DataCont {

private:

T\* data;

size\_t size;

size\_t capacity;

// Function to resize the container dynamically

void resize(size\_t newCapacity) {

T\* newData = new T[newCapacity];

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

newData[i] = data[i];

}

delete[] data;

data = newData;

capacity = newCapacity;

}

public:

// Constructor to initialize the container with an optional size

DataCont(size\_t initSize = 0) : size(initSize), capacity(initSize), data(new T[initSize]) {}

// Destructor for proper memory management

~DataCont() {

delete[] data;

}

// Overloaded subscript operator to access elements

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

if (index >= size) {

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

}

return data[index];

}

// Print all elements of the container

void printAll() const {

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

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

}

cout << endl;

}

// Get the current size of the container

size\_t getSize() const {

return size;

}

// Appends an element to the back of the container

void push\_back(const T& value) {

if (size == capacity) {

resize(capacity == 0 ? 1 : capacity \* 2);

}

data[size++] = value;

}

// Swap elements between two containers

friend void swapConts(DataCont& a, DataCont& b) {

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

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

}

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

T temp = a.data[i];

a.data[i] = b.data[i];

b.data[i] = temp;

}

}

};

int main() {

// Create instances of DataContainer for int, float, and string types

DataCont<int> intCont1, intCont2;

DataCont<float> floatCont1, floatCont2;

DataCont<string> stringCont1, stringCont2;

// Populate containers with sample data using push\_back

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

intCont1.push\_back(i + 1);

intCont2.push\_back((i + 1) \* 10);

}

for (int i = 0; i < 4; ++i) {

floatCont1.push\_back(i + 0.1f);

floatCont2.push\_back((i + 1) \* 0.5f);

}

stringCont1.push\_back("Hello");

stringCont1.push\_back("World");

stringCont1.push\_back("C++");

stringCont2.push\_back("Data");

stringCont2.push\_back("Processing");

stringCont2.push\_back("Library");

// Print all containers before swap

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

cout << "intCont1: "; intCont1.printAll();

cout << "intCont2: "; intCont2.printAll();

cout << "floatCont1: "; floatCont1.printAll();

cout << "floatCont2: "; floatCont2.printAll();

cout << "stringCont1: "; stringCont1.printAll();

cout << "stringCont2: "; stringCont2.printAll();

// Swap elements between containers of the same type

swapConts(intCont1, intCont2);

swapConts(floatCont1, floatCont2);

swapConts(stringCont1, stringCont2);

// Print all containers after swap

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

cout << "intCont1: "; intCont1.printAll();

cout << "intCont2: "; intCont2.printAll();

cout << "floatCont1: "; floatCont1.printAll();

cout << "floatCont2: "; floatCont2.printAll();

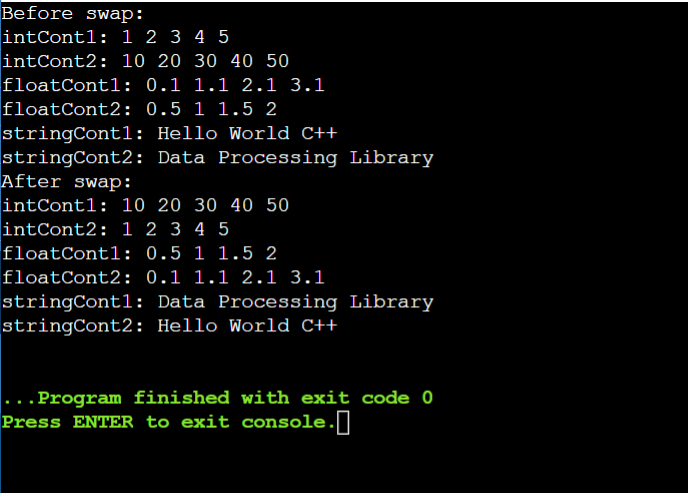
cout << "stringCont1: "; stringCont1.printAll();

cout << "stringCont2: "; stringCont2.printAll();

return 0;

}

**Output:**

****

**Smart Pointers:**

#include <iostream>

using namespace std;

//A generic smart pointer class

template <class T>

class Smartpointer {

T \*p;// Actual pointer

public:

// Constructor

Smartpointer(T \*ptr = NULL) {

p = ptr;

}

// Destructor

~Smartpointer() {

delete(p);

}

// Overloading dereferencing operator

T & operator \* () {

return \*p;

}

T \* operator -> () {

return p;

}

};

int main() {

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

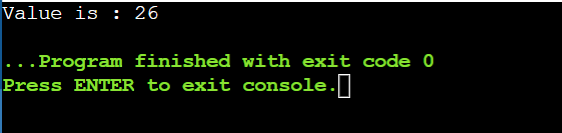
\*p = 26;

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.
   * 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) {

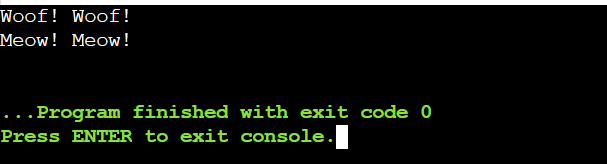
delete animal;

}

return 0;

}

**Output:**

****

**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) {

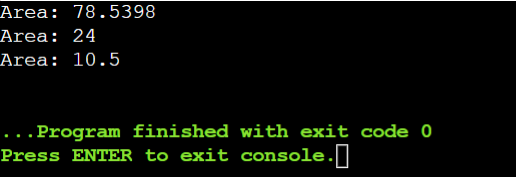
cout << "Area: " << shape->calculateArea() << endl;

}

return 0;

}

**Output:**

****

**Example for 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());

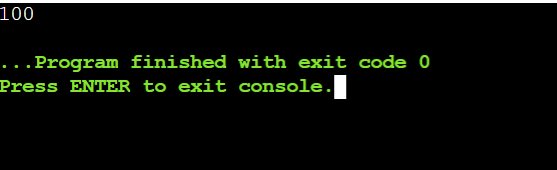
\*ptr = 100;

cout << \*ptr;

return 0;

}

**Output:**

****