**Polymorphism:**

**Design a class hierarchy for a simple graphic editor with base class Shape and derived classes Circle, Rectangle, and Triangle. Implement a virtual function draw() in the base class and override it in the derived classes. Write a function that takes a Shape\* and calls its draw() method.**

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

virtual ~Shape() {} // Virtual destructor

virtual void draw() = 0; // Pure virtual function for drawing

};

// Derived class Circle

class Circle : public Shape {

public:

void draw() override {

cout << "Draw Circle" << endl;

}

};

// Derived class Rectangle

class Rectangle : public Shape {

public:

void draw() override {

cout << "Draw Rectangle" << endl;

}

};

// Derived class Triangle

class Triangle : public Shape {

public:

void draw() override {

cout << "Draw Triangle" << endl;

}

};

// Function to call draw() method of a Shape pointer

void displayShape(Shape\* shape) {

shape->draw();

}

int main() {

// Create objects of derived classes

Circle circle;

Rectangle rectangle;

Triangle triangle;

// Call displayShape() with different Shape objects

displayShape(&circle); // Output: Drawing Circle

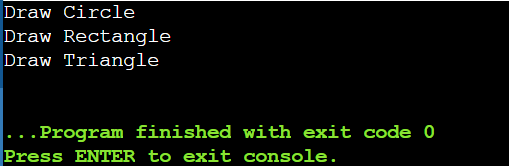
displayShape(&rectangle); // Output: Drawing Rectangle

displayShape(&triangle); // Output: Drawing Triangle

return 0;

}

**Output:**

****

**Static Members:**

**Create a class Account that has a static data member totalAccounts to keep track of the number of accounts created. Implement necessary constructors and destructors to update totalAccounts. Write a function to display the total number of accounts.**

#include <iostream>

using namespace std;

class Account {

private:

static int totalAccounts; // Static data member to keep track of total accounts

public:

// Constructor

Account() {

totalAccounts++;

}

// Destructor

~Account() {

totalAccounts--;

}

// Static function to display total number of accounts

static int getTotalAccounts() {

return totalAccounts;

}

};

// Initialize static data member

int Account::totalAccounts = 0;

void displayTotalAccounts() {

cout << "Total Accounts: " << Account::getTotalAccounts() << endl;

}

int main() {

// Create some account objects

Account acc1;

Account acc2;

Account acc3;

// Display total accounts

displayTotalAccounts(); // Output: Total Accounts: 3

{

// Create a new scope

Account acc4;

displayTotalAccounts(); // Output: Total Accounts: 4

}

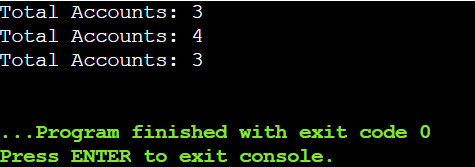
// Display total accounts after acc4 is destroyed

displayTotalAccounts(); // Output: Total Accounts: 3

return 0;

}

**Output:**

****

**Friend Functions:**

**Implement a class Box that has private data members length, breadth, and height. Write a friend function volume() that calculates and returns the volume of the box. Create objects of Box and use the friend function to compute their volumes.**

#include <iostream>

using namespace std;

class Box {

private:

float length, breadth, height;

public:

Box(float l, float b, float h) : length(l), breadth(b), height(h) {}

// Friend function declaration and definition inside class

friend float volume(const Box& box) {

return box.length \* box.breadth \* box.height;

}

};

int main() {

Box box1(2.5f, 3.5f, 4.5f);

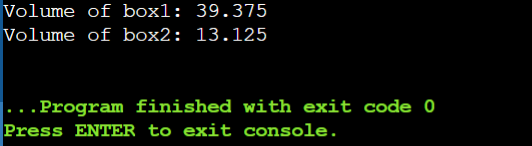
cout << "Volume of box1: " << volume(box1) << endl;

Box box2(1.5f, 2.5f, 3.5f);

cout << "Volume of box2: " << volume(box2) << endl;

return 0;

}  
**Output:**



**Templates:**

**Write a template class Array that can store an array of any data type. Include member functions to perform operations like adding an element, removing an element, and displaying the array. Demonstrate the functionality with different data types.**

#include <iostream>

using namespace std;

template <typename T, int Capacity>

class Array {

private:

T elements[Capacity];

int size;

public:

Array() : size(0) {}

void addElement(const T& element) {

if (size < Capacity) {

elements[size++] = element;

} else {

cout << "Array is full. Cannot add more elements." << endl;

}

}

void removeElement(int index) {

if (index < 0 || index >= size) {

cout << "Invalid index. Cannot remove element." << endl;

} else {

elements[index] = elements[size - 1]; // Swap with the last element

size--;

}

}

void displayArray() const {

if (size == 0) {

cout << "Array is empty." << endl;

} else {

cout << "Array elements: ";

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

cout << elements[i] << " ";

}

cout << endl;

}

}

};

int main() {

// Usage remains the same as in the previous example

Array<int, 5> intArray;

intArray.addElement(10);

intArray.addElement(20);

intArray.addElement(30);

intArray.displayArray();

// Remove an element

intArray.removeElement(1);

intArray.displayArray(); // Output: Array elements: 10 30

Array<string, 2> stringArray;

stringArray.addElement("Hello");

stringArray.addElement("World");

stringArray.displayArray();

// Remove an element

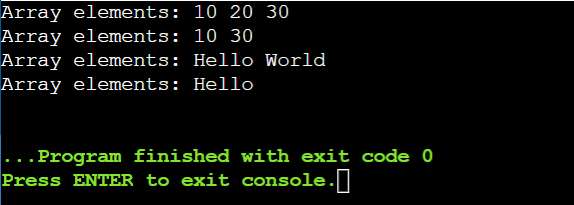
stringArray.removeElement(1);

stringArray.displayArray(); // Output: Array elements: Hello

return 0;

}

**Output:**

****

**Pointers:**

**Design a class Student with data members name and age. Create an array of Student objects dynamically using pointers. Implement functions to set and display the details of students. Also, write a function to deallocate the memory.**

#include <iostream>

#include <string>

using namespace std;

// Student class definition

class Student {

private:

string name;

int age;

public:

// Constructor to initialize name and age

Student(const string& studentName, int studentAge) : name(studentName), age(studentAge) {}

// Function to display student details

void displayDetails() const {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

const int numStudents = 3; // Number of students in the array

// Create an array of Student pointers

Student\* studentArray[numStudents];

// Allocate memory for each Student object and set details

studentArray[0] = new Student("Alice", 20);

studentArray[1] = new Student("Bob", 22);

studentArray[2] = new Student("Charlie", 21);

// Display details of each student

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

cout << "Student " << (i + 1) << ": ";

studentArray[i]->displayDetails();

}

// Deallocate memory for each Student object

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

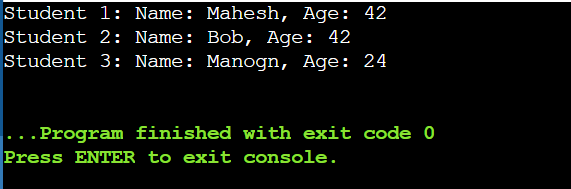
delete studentArray[i];

}

return 0;

}

Output:



**Polymorphism with Abstract Classes:**

**Create an abstract class Animal with a pure virtual function sound(). Derive classes Dog, Cat, and Cow from Animal and override the sound() function in each derived class. Write a program to demonstrate polymorphism using these classes.**

#include <iostream>

using namespace std;

// Abstract class Animal

class Animal {

public:

// Pure virtual function sound

virtual void sound() const = 0;

};

// Derived class Dog

class Dog : public Animal {

public:

void sound() const override {

cout << " Woof!" << endl;

}

};

// Derived class Cat

class Cat : public Animal {

public:

void sound() const override {

cout << "Meow!" << endl;

}

};

// Derived class Cow

class Cow : public Animal {

public:

void sound() const override {

cout << "Moo!" << endl;

}

};

int main() {

// Array of Animal references

const int numAnimals = 3;

Animal\* animals[numAnimals] = {

new Dog(),

new Cat(),

new Cow()

};

// Iterate through the array and call sound() polymorphically

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

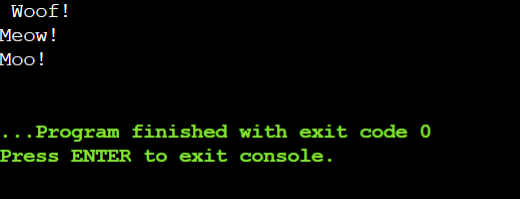
animals[i]->sound();

}

return 0;

}

**Output:**

****

**Static Member Functions:**

**Implement a class Math that has static member functions for basic mathematical operations like addition, subtraction, multiplication, and division. Demonstrate the use of these functions without creating an object of the class.**

#include <iostream>

using namespace std;

class Math {

public:

static int addition(int a, int b) {

return a + b; }

static int subtraction(int a, int b) {

return a - b; }

static int multiplication(int a, int b) {

return a \* b; }

static double division(int a, int b) {

if (b == 0) {

cerr << "Error: Division by zero!" << endl;

return 0.0; // Error case }

return static\_cast<double>(a) / b; }

};

int main() {

int x = 9, y = 5; // Using static member functions without creating an object

cout << "Addition: " << Math::addition(x, y) << endl;

cout << "Subtraction: " << Math::subtraction(x, y) << endl;

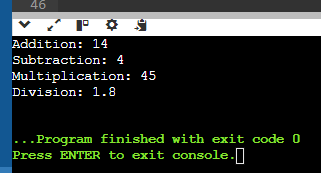
cout << "Multiplication: " << Math::multiplication(x, y) << endl;

cout << "Division: " << Math::division(x, y) << endl;

return 0;

}

**OUTPUT:**



**Friend Classes:**

**Create two classes Alpha and Beta. Make Beta a friend class of Alpha so that it can access private data members of Alpha. Implement functions in Beta to manipulate the private data of Alpha.**

#include <iostream>

using namespace std;

class Beta; // Forward declaration of Beta

class Alpha {

private:

int data;

public:

Alpha(int value) : data(value) {}

void display() const {

cout << "Alpha data: " << data << endl; }

friend class Beta; // Make Beta a friend class of Alpha

};

class Beta {

public:

void setData(Alpha& a, int value) {

a.data = value; }

void addData(Alpha& a, int value) {

a.data += value; }

};

int main() {

Alpha a(10);

a.display();

Beta b;

b.setData(a, 20);

cout << "After setting data to 20:" << endl;

a.display();

b.addData(a, 5);

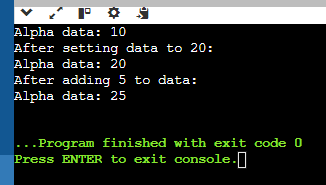
cout << "After adding 5 to data:" << endl;

a.display();

return 0;

}

**OUTPUT:**



**Class Templates with Multiple Parameters:**

**Write a class template Pair that can store a pair of values of any two data types. Include member functions to set and get the values. Demonstrate the usage of this template with different data types.**

#include <iostream>

#include <string>

using namespace std;

template <typename T1, typename T2>

class Pair {

private:

T1 first;

T2 second;

public:

void setValues(const T1& f, const T2& s) {

first = f;

second = s; }

T1 getFirst() const {

return first; }

T2 getSecond() const {

return second; }

void display() const {

cout << "First: " << first << ", Second: " << second << endl; }

};

int main() {

Pair<int, double> p1; // Pair of int and double

p1.setValues(42, 3.14);

cout << "Pair1: " << endl;

p1.display();

Pair<string, int> p2; // Pair of string and int

p2.setValues("Age", 25);

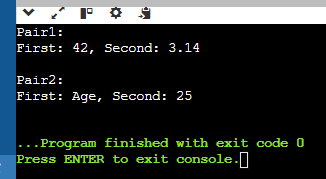
cout << "\nPair2:" << endl;

p2.display();

return 0;

}

**OUTPUT:**



**Pointer to Objects:**

**Define a class Book with data members title and author. Create an array of pointers to Book objects. Write functions to input details for each book, display the details, and search for a book by title.**

#include <iostream>

#include <string>

using namespace std;

class Book {

private:

string title;

string author;

public:

void setDetails(const string& t, const string& a) {

title = t;

author = a; }

string getTitle() const {

return title; }

void display() const {

cout << "Title: " << title << ", Author: " << author << endl; }

};

void inputDetails(Book\* books[], int n) { // Function to input details for each book

string title, author;

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

cout << "Enter details for book " << i + 1 << endl;

cout << "Title: ";

getline(cin, title);

cout << "Author: ";

getline(cin, author);

books[i] = new Book();

books[i]->setDetails(title, author);

}

} void displayDetails(Book\* books[], int n) { // Function to display details of all books

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

books[i]->display();

}

}

Book\* searchByTitle(Book\* books[], int n, const string& title) {

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

if (books[i]->getTitle() == title) {

return books[i];

} }

return nullptr;

}

int main() {

int n;

cout << "Enter the number of books: ";

cin >> n;

cin.ignore();

Book\* books[n];

inputDetails(books, n);

cout << "\nDisplaying book details:\n";

displayDetails(books, n);

string title;

cout << "\nEnter the title of the book to search: ";

getline(cin, title);

Book\* foundBook = searchByTitle(books, n, title);

if (foundBook) {

cout << "\nBook found:\n";

foundBook->display();

} else {

cout << "\nBook not found.\n";

}

return 0; }

**OUTPUT:**  
