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>

class Shape {

public:

virtual void draw() const {

std::cout << "Draw a shape." << std::endl;

}

virtual ~Shape() = default;

};

class Circle : public Shape {

public:

void draw() const override {

std::cout << "Draw a circle." << std::endl;

}

};

class Rectangle : public Shape {

public:

void draw() const override {

std::cout << "Draw a rectangle." << std::endl;

}

};

class Triangle : public Shape {

public:

void draw() const override {

std::cout << "Draw a triangle." << std::endl;

}

};

void drawShape(const Shape\* shape) {

shape->draw();

}

int main() {

Shape\* shape1 = new Circle();

Shape\* shape2 = new Rectangle();

Shape\* shape3 = new Triangle();

drawShape(shape1);

drawShape(shape2);

drawShape(shape3);

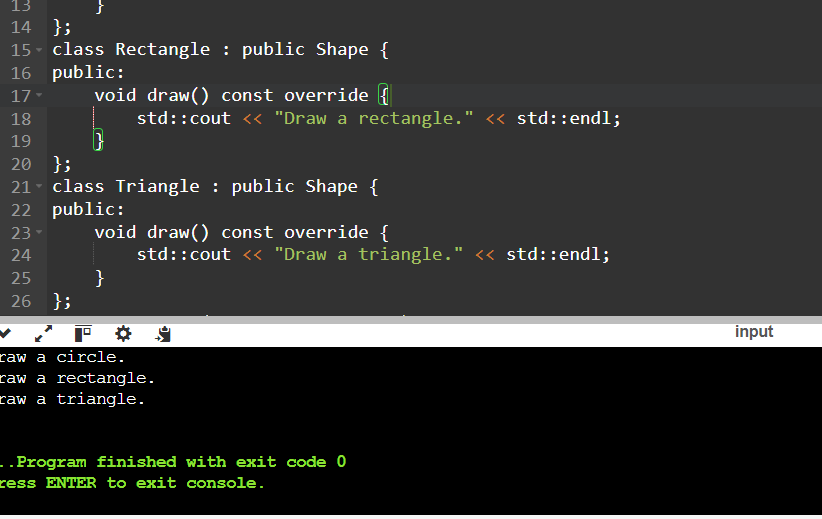
delete shape1;

delete shape2;

delete shape3;

return 0;

}



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>

class Account {

public:

static int totalAccounts;

Account() {

totalAccounts++;

}

~Account() {

totalAccounts--;

}

static void displayTotalAccounts() {

std::cout << "Total Accounts: "<<totalAccounts << std::endl;

}

};

int Account::totalAccounts = 3;

int main() {

Account acc202;

Account acc203;

Account acc204;

Account::displayTotalAccounts(); {

Account acc203;

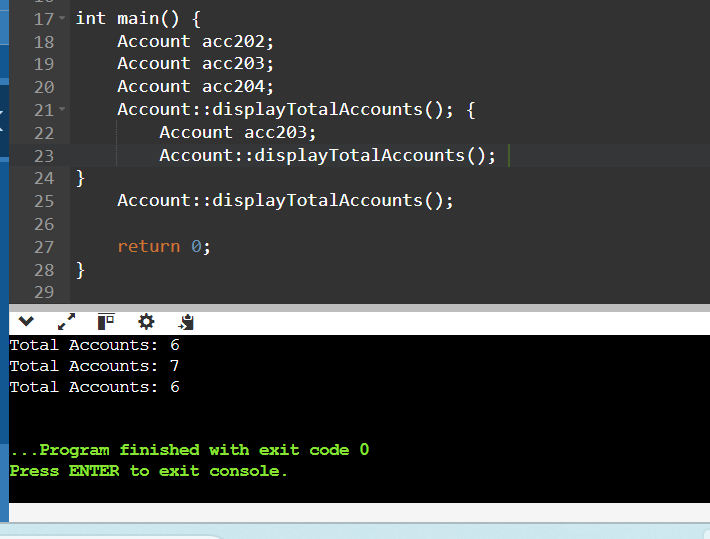
Account::displayTotalAccounts();

}

Account::displayTotalAccounts();

return 0;

}



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;

float breath;

float height;

public:

Box() : length(0), breath(0), height(0) {}

Box(float length,float breath,float height){

this->length=length;

this->breath=breath;

this->height=height;

}

double getLength() const {

return length;

}

double getBreadth() const {

return breath;

}

double getHeight() const {

return height;

}

friend float volume(const Box& b);

};

float volume(const Box& b) {

return b.length \* b.breath \* b.height;

}

int main() {

Box box1(2.5,6.0,8.9);

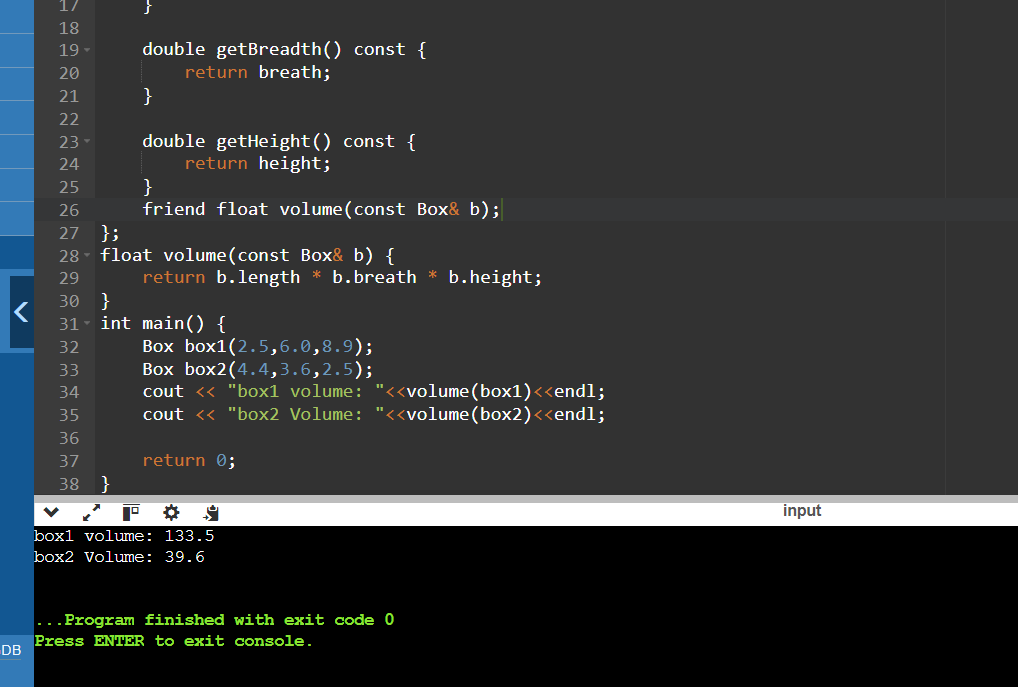
Box box2(4.4,3.6,2.5);

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

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

return 0;

}



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>

template <typename T>

class Array {

private:

T\* arr;

int size;

int capacity;

void resize() {

capacity \*= 2;

T\* newArr = new T[capacity];

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

newArr[i] = arr[i];

}

delete[] arr;

arr = newArr;

}

public:

// Constructor

Array() : size(0), capacity(10) {

arr = new T[capacity];

}

// Destructor

~Array() {

delete[] arr;

}

// Add an element to the array

void addElement(const T& element) {

if (size == capacity) {

resize();

}

arr[size++] = element;

}

void removeElement(int index) {

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

for (int i = index; i < size - 1; ++i) {

arr[i] = arr[i + 1];

}

--size;

} else {

std::cout << "Index out of bounds." << std::endl;

}

}

void display() const {

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

std::cout << arr[i] << " ";

}

std::cout << std::endl;

}

};

int main() {

Array<int> intArray;

intArray.addElement(1);

intArray.addElement(2);

intArray.addElement(3);

std::cout << "Integer array: ";

intArray.display();

intArray.removeElement(1);

std::cout << "Integer array after removing element at index 1: ";

intArray.display();

Array<double> doubleArray;

doubleArray.addElement(4.3);

doubleArray.addElement(6.6);

doubleArray.addElement(5.3);

std::cout << "Double array: ";

doubleArray.display();

doubleArray.removeElement(0);

std::cout << "After removing element at index 0: ";

doubleArray.display();

Array<std::string> stringArray;

stringArray.addElement("Hey everyone");

stringArray.addElement("this is me");

std::cout << "String array: ";

stringArray.display();

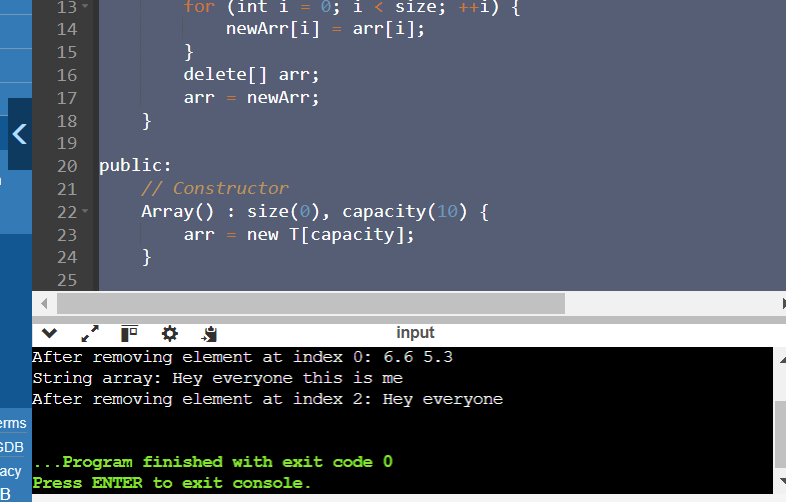
stringArray.removeElement(1);

std::cout << "After removing element at index 2: ";

stringArray.display();

return 0;

}



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;

class Student {

private:

string name;

int age;

public:

Student() : name(""), age(0) {}

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

void setDetails(const string& name, int age) {

this->name = name;

this->age = age;

}

void display() const {

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

}

};

Student\* createStudents(int count) {

return new Student[count];

}

int main() {

int numStudents;

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

cin >> numStudents;

Student\* students = createStudents(numStudents);

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

string name;

int age;

std::cout << "Enter details for student " << i + 1 << std::endl;

std::cout << "Name: ";

std::cin >> name;

std::cout << "Age: ";

std::cin >> age;

students[i].setDetails(name, age);

}

cout << "\nStudent details:" << endl;

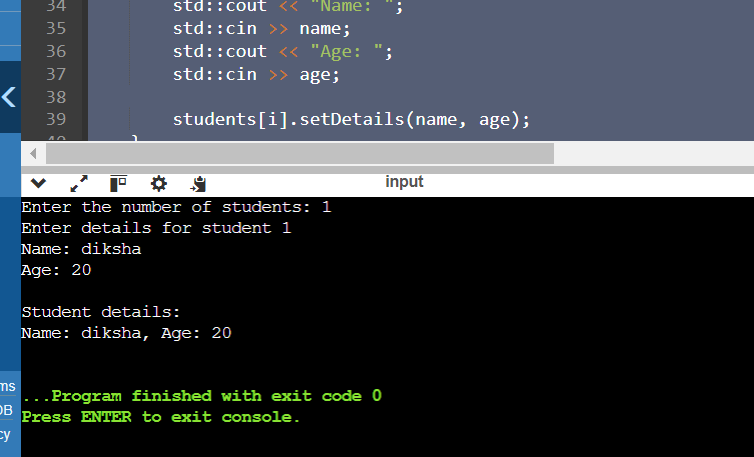
for (int i = 0; i < numStudents; ++i) {

students[i].display();

}

return 0;

}



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>

class Animal {

public:

virtual void sound() const = 0;

virtual ~Animal() = default;

};

class Cat : public Animal {

public:

void sound() const override {

std::cout << "Meeeeooowww" << std::endl;

}

};

class Dog : public Animal {

public:

void sound() const override {

std::cout << "Wooooff" << std::endl;

}

};

class Cow : public Animal {

public:

void sound() const override {

std::cout << "moooo" << std::endl;

}

};

void makeSound(const Animal& animal) {

animal.sound();

}

int main() {

Cat cat;

Dog dog;

Cow cow;

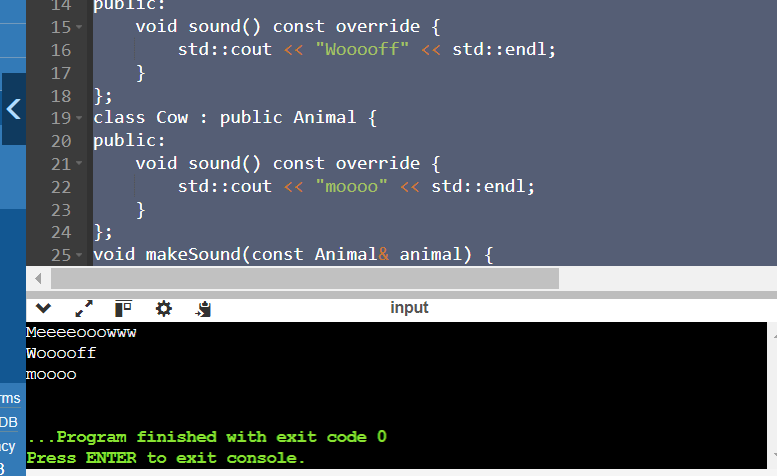
makeSound(cat);

makeSound(dog);

makeSound(cow);

return 0;

}



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>

class Math {

public:

static int add(int a, int b) { return a + b; }

static int subtract(int a, int b) { return a - b; }

static int multiply(int a, int b) { return a \* b; }

static double divide(int a, int b) {

return b != 0 ? static\_cast<double>(a) / b : 0;

}

};

int main() {

std::cout << "Addition: " << Math::add(231, 115) << std::endl;

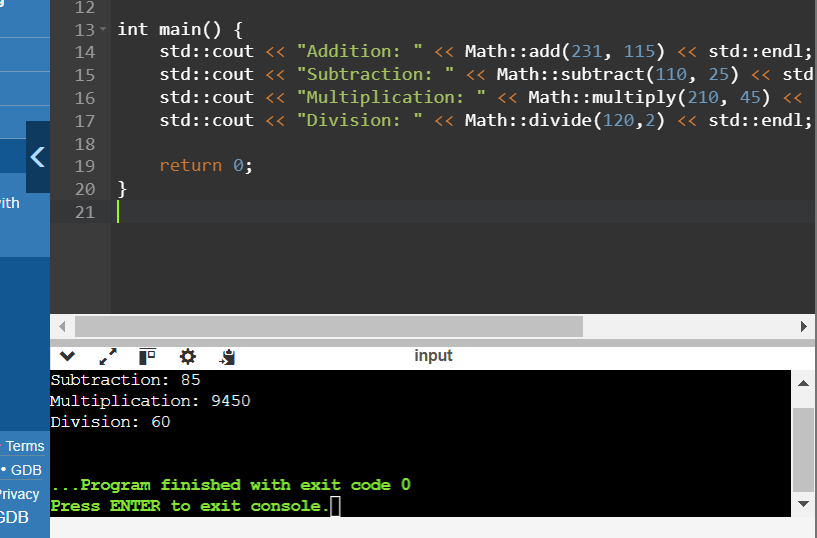
std::cout << "Subtraction: " << Math::subtract(110, 25) << std::endl;

std::cout << "Multiplication: " << Math::multiply(210, 45) << std::endl;

std::cout << "Division: " << Math::divide(120,2) << std::endl;

return 0;

}



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;

class Alpha {

private:

int x;

friend class Beta;

public:

Alpha(int val = 0) : x(val) {}

void display() {

cout << "Alpha::x = " << x << endl;

}

};

class Beta {

public:

void manipulateAlpha(Alpha& alpha, int val) {

alpha.x = val;

}

void displayAlpha(const Alpha& alpha) {

cout << "Alpha::x = " << alpha.x << endl;

}

};

int main() {

Alpha alpha(20);

alpha.display();

Beta beta;

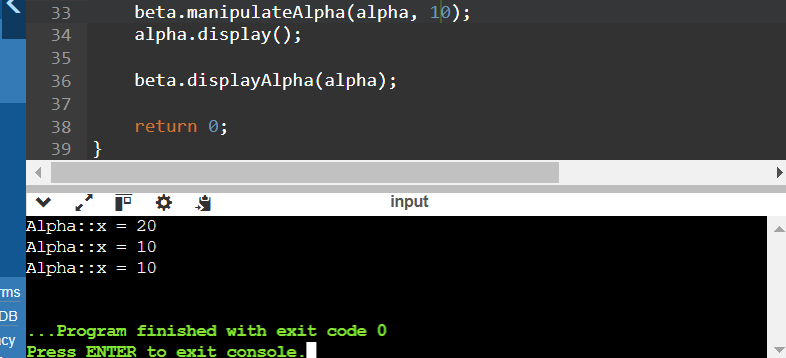
beta.manipulateAlpha(alpha, 10);

alpha.display();

beta.displayAlpha(alpha);

return 0;

}



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>

template <typename T1, typename T2>

class Pair {

private:

T1 first;

T2 second;

public:

Pair(const T1& f, const T2& s) : first(f), second(s) {}

void setFirst(const T1& f) { first = f; }

void setSecond(const T2& s) { second = s; }

T1 getFirst() const { return first; }

T2 getSecond() const { return second; }

};

int main() {

Pair<int, double> pair1(24, 3.14);

Pair<std::string, char> pair2("Hey", 'Diksha');

std::cout << "Pair 1: " << pair1.getFirst() << ", " << pair1.getSecond() << std::endl;

std::cout << "Pair 2: " << pair2.getFirst() << ", " << pair2.getSecond() << std::endl;

pair1.setFirst(50);

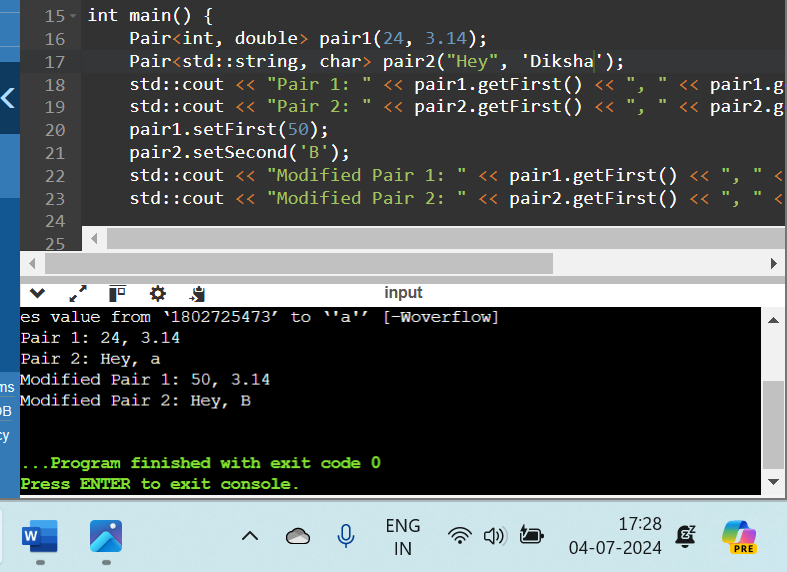
pair2.setSecond('B');

std::cout << "Modified Pair 1: " << pair1.getFirst() << ", " << pair1.getSecond() << std::endl;

std::cout << "Modified Pair 2: " << pair2.getFirst() << ", " << pair2.getSecond() << std::endl;

return 0;

}



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>

class Book {

private:

std::string title;

std::string author;

public:

Book(const std::string& t, const std::string& a) : title(t), author(a) {}

void displayDetails() const {

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

}

const std::string& getTitle() const {

return title;

}

};

int main() {

const int MAX\_BOOKS = 3;

Book\* library[MAX\_BOOKS];

for (int i = 0; i < MAX\_BOOKS; ++i) {

std::string title, author;

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

std::cout << "Title: ";

std::getline(std::cin >> std::ws, title);

std::cout << "Author: ";

std::getline(std::cin >> std::ws, author);

library[i] = new Book(title, author);

}

std::cout << "\nLibrary Books:\n";

for (int i = 0; i < MAX\_BOOKS; ++i) {

std::cout << "Book " << i + 1 << ": ";

library[i]->displayDetails();

}

std::string searchTitle;

std::cout << "\nEnter title to search: ";

std::getline(std::cin >> std::ws, searchTitle);

bool found = false;

for (int i = 0; i < MAX\_BOOKS; ++i) {

if (library[i]->getTitle() == searchTitle) {

std::cout << "Book found at index " << i + 1 << ": ";

library[i]->displayDetails();

found = true;

break;

}

}

if (found) {

std::cout << "Book with title \"" << searchTitle << "\" not found." << std::endl;

}

for (int i = 0; i < MAX\_BOOKS; ++i) {

delete library[i];

}

return 0;

}

