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 Shape

class Shape {

public:

// Virtual function draw() to be overridden by derived classes

virtual void draw() const = 0;

// Virtual destructor

virtual ~Shape() {}

};

// Derived class Circle

class Circle : public Shape {

public:

void draw() const override {

cout << "Drawing a Circle" << endl;

}

};

// Derived class Rectangle

class Rectangle : public Shape {

public:

void draw() const override {

cout << "Drawing a Rectangle" << endl;

}

};

// Derived class Triangle

class Triangle : public Shape {

public:

void draw() const override {

cout << "Drawing a Triangle" << endl;

}

};

// Function to render a shape

void renderShape(const Shape\* shape) {

shape->draw();

}

// Main function

int main() {

Circle circle;

Rectangle rectangle;

Triangle triangle;

// Array of pointers to Shape objects

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

// Iterate through the array and call draw() on each pointer

for (const Shape\* shape : shapes) {

renderShape(shape);

}

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>

using namespace std;

class Account {

public:

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

// Constructor

Account() {

totalAccounts++;

}

// Destructor

~Account() {

totalAccounts--;

}

// Static member function to display the total number of accounts

static void displayTotalAccounts() {

cout << "Total number of accounts: " << totalAccounts << endl;

}

};

// Initialize static data member

int Account::totalAccounts = 0;

int main() {

// Creating accounts

Account acc1;

Account acc2;

Account acc3;

// Display the total number of accounts

Account::displayTotalAccounts();

// Destroy one account

{

Account acc4;

Account::displayTotalAccounts();

} // acc4 goes out of scope and is destroyed here

// Display the total number of accounts again

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:

double length;

double breadth;

double height;

public:

// Constructor to initialize the dimensions of the box

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

// Friend function declaration

friend double volume(const Box& box);

};

// Friend function definition

double volume(const Box& box) {

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

}

int main() {

// Creating Box objects

Box box1(3.0, 4.0, 5.0);

Box box2(6.0, 7.0, 8.0);

// Compute and display the volumes of the boxes

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

cout << "Volume of box2: " << 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>

using namespace std;

template <typename T>

class Array {

private:

T\* arr;

int capacity;

int size;

public:

// Constructor to initialize the array with a given capacity

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

arr = new T[capacity];

}

// Destructor to clean up dynamically allocated memory

~Array() {

delete[] arr;

}

// Function to add an element to the array

void addElement(const T& element) {

if (size < capacity) {

arr[size++] = element;

} else {

cout << "Array is full, cannot add element" << endl;

}

}

// Function to remove an element from the array (removes last element)

void removeElement() {

if (size > 0) {

size--;

} else {

cout << "Array is empty, cannot remove element" << endl;

}

}

// Function to display the array

void display() const {

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

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

}

cout << endl;

}

};

int main() {

// Create an Array of integers

Array<int> intArray(5);

intArray.addElement(1);

intArray.addElement(2);

intArray.addElement(3);

cout << "Integer Array: ";

intArray.display();

intArray.removeElement();

cout << "After removing an element: ";

intArray.display();

// Create an Array of doubles

Array<double> doubleArray(5);

doubleArray.addElement(1.1);

doubleArray.addElement(2.2);

doubleArray.addElement(3.3);

cout << "Double Array: ";

doubleArray.display();

doubleArray.removeElement();

cout << "After removing an element: ";

doubleArray.display();

// Create an Array of strings

Array<string> stringArray(5);

stringArray.addElement("Hello");

stringArray.addElement("World");

stringArray.addElement("Template");

cout << "String Array: ";

stringArray.display();

stringArray.removeElement();

cout << "After removing an element: ";

stringArray.display();

return

}

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:

// Function to set the details of the student

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

this->name = name;

this->age = age;

}

// Function to display the details of the student

void display() const {

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

}

};

// Function to create an array of Student objects dynamically

Student\* createStudentArray(int size) {

return new Student[size];

}

// Function to set the details of students in the array

void setStudentDetails(Student\* students, int size) {

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

string name;

int age;

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

cout << "Name: ";

cin >> name;

cout << "Age: ";

cin >> age;

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

}

}

// Function to display the details of students in the array

void displayStudentDetails(const Student\* students, int size) {

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

cout << "Details of student " << i + 1 << ":" << endl;

students[i].display();

}

}

// Function to deallocate the memory of the student array

void deallocateStudentArray(Student\* students) {

delete[] students;

}

int main() {

int size;

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

cin >> size;

// Create an array of Student objects dynamically

Student\* students = createStudentArray(size);

// Set the details of students

setStudentDetails(students, size);

// Display the details of students

displayStudentDetails(students, size);

// Deallocate the memory of the student array

deallocateStudentArray(students);

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>

using namespace std;

class Animal {

public:

// Pure virtual function

virtual void sound() const = 0;

// Virtual destructor

virtual ~Animal() {}

};

class Dog : public Animal {

public:

void sound() const override {

cout << "Woof!" << endl;

}

};

class Cat : public Animal {

public:

void sound() const override {

cout << "Meow!" << endl;

}

};

class Cow : public Animal {

public:

void sound() const override {

cout << "Moo!" << endl;

}

};

int main() {

// Create pointers to Animal objects

Animal\* dog = new Dog();

Animal\* cat = new Cat();

Animal\* cow = new Cow();

// Demonstrate polymorphism

dog->sound(); // Output: Woof!

cat->sound(); // Output: Meow!

cow->sound(); // Output: Moo!

// Clean up dynamically allocated memory

delete dog;

delete cat;

delete 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>

using namespace std;

class Math {

public:

// Static member functions for basic mathematical operations

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) {

if (b == 0) {

cout << "Error: Division by zero" << endl;

return 0;

}

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

}

};

int main() {

int a = 10;

int b = 5;

// Demonstrate the use of static member functions without creating an object of the class

cout << "Addition: " << Math::add(a, b) << endl;

cout << "Subtraction: " << Math::subtract(a, b) << endl;

cout << "Multiplication: " << Math::multiply(a, b) << endl;

cout << "Division: " << Math::divide(a, b) << 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; // Forward declaration of Beta class

class Alpha {

private:

int value;

public:

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

// Function to display the value

void display() const {

cout << "Alpha value: " << value << endl;

}

// Declare Beta as a friend class

friend class Beta;

};

class Beta {

public:

// Function to set the value of Alpha

void setValue(Alpha& a, int val) {

a.value = val;

}

// Function to get the value of Alpha

int getValue(const Alpha& a) const {

return a.value;

}

};

int main() {

Alpha a(10);

Beta b;

// Display initial value of Alpha

a.display();

// Manipulate the value of Alpha using Beta

b.setValue(a, 20);

// Display modified value of Alpha

a.display();

// Get the value of Alpha using Beta

cout << "Beta accessed value: " << b.getValue(a) << endl;

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>

using namespace std;

template <typename T1, typename T2>

class Pair {

private:

T1 first;

T2 second;

public:

// Constructor

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

// Member functions to set values

void setFirst(T1 f) {

first = f;

}

void setSecond(T2 s) {

second = s;

}

// Member functions to get values

T1 getFirst() const {

return first;

}

T2 getSecond() const {

return second;

}

// Function to display the pair

void display() const {

cout << "Pair(" << first << ", " << second << ")" << endl;

}

};

int main() {

// Create a Pair of int and double

Pair<int, double> p1(10, 20.5);

p1.display();

// Create a Pair of string and char

Pair<string, char> p2("Hello", 'A');

p2.display();

// Create a Pair of float and bool

Pair<float, bool> p3(3.14f, true);

p3.display();

// Modify the pairs using set functions

p1.setFirst(15);

p1.setSecond(30.5);

p1.display();

p2.setFirst("World");

p2.setSecond('B');

p2.display();

p3.setFirst(1.618f);

p3.setSecond(false);

p3.display();

// Access the values using get functions

cout << "First value of p1: " << p1.getFirst() << endl;

cout << "Second value of p1: " << p1.getSecond() << endl;

cout << "First value of p2: " << p2.getFirst() << endl;

cout << "Second value of p2: " << p2.getSecond() << endl;

cout << "First value of p3: " << p3.getFirst() << endl;

cout << "Second value of p3: " << p3.getSecond() << 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>

using namespace std;

class Book {

private:

string title;

string author;

public:

// Constructor

Book(string t = "", string a = "") : title(t), author(a) {}

// Function to set the details of the book

void setDetails(string t, string a) {

title = t;

author = a;

}

// Function to display the details of the book

void displayDetails() const {

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

}

// Function to get the title of the book

string getTitle() const {

return title;

}

};

void inputBookDetails(Book\* books[], int n) {

string title, author;

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

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

cout << "Title: ";

cin.ignore();

getline(cin, title);

cout << "Author: ";

getline(cin, author);

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

}

}

void displayBooks(Book\* books[], int n) {

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

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

books[i]->displayDetails();

}

}

Book\* searchBookByTitle(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;

// Create an array of pointers to Book objects

Book\* books[n];

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

books[i] = new Book;

}

// Input book details

inputBookDetails(books, n);

// Display all books

displayBooks(books, n);

// Search for a book by title

string searchTitle;

cout << "Enter the title of the book to search: ";

cin.ignore();

getline(cin, searchTitle);

Book\* foundBook = searchBookByTitle(books, n, searchTitle);

if (foundBook) {

cout << "Book found:" << endl;

foundBook->displayDetails();

} else {

cout << "Book not found." << endl;

}

// Deallocate memory

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

delete books[i];

}

return 0;

}