**First Exam Of C++:**

Q1. 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.

Solution:

#include <iostream>

class Shape {

public:

virtual void draw() = 0; // Pure virtual function, must be implemented in derived classes

};

class Circle : public Shape {

public:

void draw() override {

std::cout << "Drawing a circle\n";

}

};

class Rectangle : public Shape {

public:

void draw() override {

std::cout << "Drawing a rectangle\n";

}

};

class Triangle : public Shape {

public:

void draw() override {

std::cout << "Drawing a triangle\n";

}

};

// Function to draw any shape using polymorphism

void drawShape(Shape\* shape) {

shape->draw();

}

int main() {

Circle circle;

Rectangle rectangle;

Triangle triangle;

drawShape(&circle);

drawShape(&rectangle);

drawShape(&triangle);

return 0;

}

Q2. 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.

Solution:

#include <iostream>

class Account {

public:

// Default constructor

Account() {

totalAccounts++;

}

// Destructor

~Account() {

totalAccounts--;

}

static void displayTotalAccounts() {

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

}

private:

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

};

// Initialize static member outside the class (required)

int Account::totalAccounts = 0;

int main() {

Account account1;

Account account2;

Account account3;

Account::displayTotalAccounts(); // Access static member function with class name

return 0;

}

Q3. 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.

Solution:

#include <iostream>

class Box {

private:

double length;

double breadth;

double height;

public:

// Constructor to initialize data members

Box(double len, double brth, double hgt) : length(len), breadth(brth), height(hgt) {}

// Friend function to calculate volume with access to private members

friend double volume(const Box& box);

};

// Friend function definition outside the class

double volume(const Box& box) {

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

}

int main() {

Box box1(2.0, 3.0, 4.0);

Box box2(5.0, 6.0, 7.0);

double vol1 = volume(box1);

double vol2 = volume(box2);

std::cout << "Volume of box 1: " << vol1 << std::endl;

std::cout << "Volume of box 2: " << vol2 << std::endl;

return 0;

}

Q4. 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.

Solution:

#include <iostream>

template <typename T>

class Array {

private:

T\* data; // Pointer to hold the array elements

int size; // Size of the array

public:

// Constructor to allocate memory for the array

Array(int capacity) {

size = capacity;

data = new T[size];

}

// Destructor to deallocate memory

~Array() {

delete[] data;

}

// Check if the array is full

bool isFull() const {

return size == capacity;

}

// Check if the array is empty

bool isEmpty() const {

return size == 0;

}

// Add an element at the end of the array

void add(const T& element) {

if (isFull()) {

std::cout << "Array is full\n";

return;

}

data[size++] = element;

}

// Remove the last element from the array

void remove() {

if (isEmpty()) {

std::cout << "Array is empty\n";

return;

}

size--;

}

// Display all elements of the array

void display() const {

if (isEmpty()) {

std::cout << "Array is empty\n";

return;

}

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

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

}

std::cout << std::endl;

}

};

int main() {

// Array of integers

Array<int> intArray(5);

intArray.add(10);

intArray.add(20);

intArray.add(30);

intArray.display(); // Output: 10 20 30

// Array of doubles

Array<double> doubleArray(3);

doubleArray.add(1.5);

doubleArray.add(2.7);

doubleArray.display(); // Output: 1.5 2.7

// Array of strings

Array<std::string> stringArray(2);

stringArray.add("Hello");

stringArray.add("World");

stringArray.display(); // Output: Hello World

return 0;

}

Q5. 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.

Solution:

#include <iostream>

class Student {

private:

std::string name;

int age;

public:

// Function to set student details

void setDetails(const std::string& n, int a) {

name = n;

age = a;

}

// Function to display student details

void displayDetails() const {

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

}

};

int main() {

int numStudents;

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

std::cin >> numStudents;

// Dynamically allocate memory for an array of Student objects

Student\* students = new Student[numStudents];

// Set student details

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

std::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);

}

// Display student details

std::cout << "\nStudent Details:\n";

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

students[i].displayDetails();

}

// Deallocate memory

delete[] students;

students = nullptr; // Set pointer to null for safety

return 0;

}

Q6. 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.

Solution:

#include <iostream>

class Animal {

public:

virtual void sound() = 0; // Pure virtual function, must be implemented in derived classes

};

class Dog : public Animal {

public:

void sound() override {

std::cout << "Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

void sound() override {

std::cout << "Meow!" << std::endl;

}

};

class Cow : public Animal {

public:

void sound() override {

std::cout << "Moo!" << std::endl;

}

};

void makeAnimalSound(Animal\* animal) {

animal->sound(); // Virtual dispatch based on actual object type

}

int main() {

Dog dog;

Cat cat;

Cow cow;

makeAnimalSound(&dog); // Output: Woof!

makeAnimalSound(&cat); // Output: Meow!

makeAnimalSound(&cow); // Output: Moo!

return 0;

}

Q7. 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.

Solution:

#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 double multiply(double a, double b) {

return a \* b;

}

static double divide(double a, double b) {

if (b == 0) {

std::cout << "Error: Division by zero\n";

return 0.0; // Handle division by zero (replace with appropriate action)

}

return a / b;

}

};

int main() {

int sum = Math::add(5, 3);

int difference = Math::subtract(10, 4);

double product = Math::multiply(2.5, 6.0);

double quotient = Math::divide(12.0, 3.0);

std::cout << "Sum: " << sum << std::endl;

std::cout << "Difference: " << difference << std::endl;

std::cout << "Product: " << product << std::endl;

std::cout << "Quotient: " << quotient << std::endl;

return 0;

}

Q8. 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.

Solution:

#include <iostream>

class Alpha {

private:

int data;

public:

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

// Grant access to private data member to Beta class

friend class Beta;

};

class Beta {

public:

// Function to modify private data of Alpha object

void modifyAlphaData(Alpha& alpha, int newValue) {

alpha.data = newValue; // Access private member 'data' of Alpha

}

// Function to display private data of Alpha object

void displayAlphaData(const Alpha& alpha) {

std::cout << "Alpha's private data: " << alpha.data << std::endl;

}

};

int main() {

Alpha alpha(10);

Beta beta;

beta.modifyAlphaData(alpha, 20); // Modify alpha's data through Beta

beta.displayAlphaData(alpha); // Display modified data through Beta

return 0;

}

Q.9. 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.

Solution:

#include <iostream>

template <typename T1, typename T2>

class Pair {

private:

T1 value1;

T2 value2;

public:

// Constructor to set values

Pair(const T1& val1, const T2& val2) : value1(val1), value2(val2) {}

// Getters for individual values

const T1& getFirst() const { return value1; }

const T2& getSecond() const { return value2; }

// Function to display the pair

void display() const {

std::cout << "Pair: (" << value1 << ", " << value2 << ")" << std::endl;

}

};

int main() {

// Pair of integers

Pair<int, int> intPair(10, 20);

intPair.display(); // Output: Pair: (10, 20)

// Pair of strings

Pair<std::string, double> stringDoublePair("Hello", 3.14);

stringDoublePair.display(); // Output: Pair: (Hello, 3.14)

return 0;

}

Q.10. . 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.

Solution:

#include <iostream>

#include <string>

class Book {

private:

std::string title;

std::string author;

public:

// Function to set book details

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

title = t;

author = a;

}

// Function to display book details

void displayDetails() const {

std::cout << "Title: " << title << std::endl;

std::cout << "Author: " << author << std::endl;

}

};

int main() {

const int MAX\_BOOKS = 10; // Maximum number of books

Book\* bookArr[MAX\_BOOKS]; // Array of pointers to Book objects

int numBooks = 0;

char choice;

do {

if (numBooks == MAX\_BOOKS) {

std::cout << "Array is full!" << std::endl;

break;

}

// Allocate memory for a new Book object

bookArr[numBooks] = new Book;

// Input book details

std::string title, author;

std::cout << "Enter book title: ";

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

std::cout << "Enter book author: ";

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

bookArr[numBooks]->setDetails(title, author);

numBooks++;

std::cout << "Add another book (y/n)? ";

std::cin >> choice;

std::cin.ignore(); // Consume newline character

} while (choice == 'y' || choice == 'Y');

// Display all books

std::cout << "\nBook List:\n";

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

bookArr[i]->displayDetails();

std::cout << std::endl;

}

// Search for a book by title

std::string searchTitle;

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

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

bool found = false;

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

if (bookArr[i]->title == searchTitle) {

std::cout << "Book found!\n";

bookArr[i]->displayDetails();

found = true;

break;

}

}

if (!found) {

std::cout << "Book not found.\n";

}

// Deallocate memory

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

delete bookArr[i];

bookArr[i] = nullptr; // Set pointer to null for safety

}

return 0;

}