Part A

Q5)--------------------------------------------------------------------------------🡪

#include <iostream>

using namespace std;

int fib(int x) {

if((x==1)||(x==0)) {

return(x);

}else {

return(fib(x-1)+fib(x-2));

}

}

int main() {

int x , i=0;

cout << "Enter the number of terms of series : ";

cin >> x;

cout << "Fibonnaci Series : ";

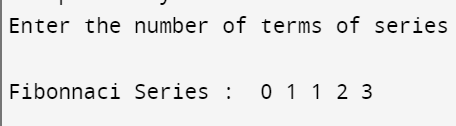
while(i < x) {

cout << " " << fib(i);

i++;

}

return 0;



}Q2)--------------------------------------------------------------------------🡪

#include <bits/stdc++.h>

using namespace std;

int partition(int arr[], int start, int end)

{

int pivot = arr[start];

int count = 0;

for (int i = start + 1; i <= end; i++) {

if (arr[i] <= pivot)

count++;

}

int pivotI = start + count;

swap(arr[pivotI], arr[start]);

int i = start, j = end;

while (i < pivotI && j > pivotI) {

while (arr[i] <= pivot) {

i++;

}

while (arr[j] > pivot) {

j--;

} if (i < pivotI && j > pivotI) {

swap(arr[i++], arr[j--]);

}

}

return pivotI;

void quickSort(int arr[], int start, int end)

{ if (start >= end)

return;

int p = partition(arr, start, end);

quickSort(arr, start, p - 1);

quickSort(arr, p + 1, end);

}

int main()

{

int arr[] = { 9, 3, 4, 2, 1, 8 };

int n = 6;

quickSort(arr, 0, n - 1);

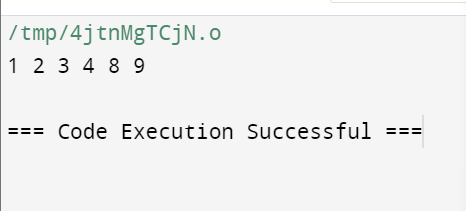
for (int i = 0; i < n; i++) {

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

}

return 0;

}



Q3)-----------------------------------------------------------------------------------------🡪

#include <iostream>

using namespace std;

// Function to display matrix

void displayMatrix(int \*\*matrix, int rows, int cols) {

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

for (int j = 0; j < cols; j++) {

cout << matrix[i][j] << " ";

}

cout << endl;

}

}

// Function to add two matrices

int\*\* addMatrices(int \*\*matrix1, int \*\*matrix2, int rows, int cols) {

int \*\*result = new int\*[rows];

for (int k = 0; k < rows; k++) {

result[i] = new int[cols];

for (int j = 0; j < cols; j++) {

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

}

return result;

}

// Function to subtract two matrices

int\*\* subtractMatrices(int \*\*matrix1, int \*\*matrix2, int rows, int cols) {

int \*\*result = new int\*[rows];

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

result[i] = new int[cols];

for (int j = 0; j < cols; j++) {

result[i][j] = matrix1[i][j] - matrix2[i][j];

}

}

return result;

}

// Function to multiply two matrices

int\*\* multiplyMatrices(int \*\*matrix1, int \*\*matrix2, int rows1, int cols1, int cols2) {

int \*\*result = new int\*[rows1];

for (int i = 0; i < rows1; i++) {

result[i] = new int[cols2];

for (int j = 0; j < cols2; j++) {

result[i][j] = 0;

for (int k = 0; k < cols1; k++) {

result[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

return result;

}

// Function to transpose a matrix

int\*\* transposeMatrix(int \*\*matrix, int rows, int cols) {

int \*\*result = new int\*[cols];

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

result[i] = new int[rows];

for (int j = 0; j < rows; j++) {

result[i][j] = matrix[j][i];

}

}

return result;

}

int main() {

int rows1, cols1, rows2, cols2;

// Input dimensions of first matrix

cout << "Enter the number of rows and columns of first matrix: ";

cin >> rows1 >> cols1;

// Input dimensions of second matrix

cout << "Enter the number of rows and columns of second matrix: ";

cin >> rows2 >> cols2;

if (cols1 != rows2) {

cout << "Matrices cannot be multiplied due to incompatible dimensions." << endl;

return 1;

}

// Input elements of first matrix

cout << "Enter elements of first matrix:" << endl;

int \*\*matrix1 = new int\*[rows1];

for (int i = 0; i < rows1; i++) {

matrix1[i] = new int[cols1];

for (int j = 0; j < cols1; j++) {

cin >> matrix1[i][j];

}

}

// Input elements of second matrix

cout << "Enter elements of second matrix:" << endl;

int \*\*matrix2 = new int\*[rows2];

for (int i = 0; i < rows2; i++) {

matrix2[i] = new int[cols2];

for (int j = 0; j < cols2; j++) {

cin >> matrix2[i][j];

}

}

// Perform operations

int \*\*resultAddition = addMatrices(matrix1, matrix2, rows1, cols1);

int \*\*resultSubtraction = subtractMatrices(matrix1, matrix2, rows1, cols1);

int \*\*resultMultiplication = multiplyMatrices(matrix1, matrix2, rows1, cols1, cols2);

int \*\*transpose1 = transposeMatrix(matrix1, rows1, cols1);

int \*\*transpose2 = transposeMatrix(matrix2, rows2, cols2);

// Display results

cout << "Matrix Addition:" << endl;

displayMatrix(resultAddition, rows1, cols1);

cout << "Matrix Subtraction:" << endl;

displayMatrix(resultSubtraction, rows1, cols1);

cout << "Matrix Multiplication:" << endl;

displayMatrix(resultMultiplication, rows1, cols2);

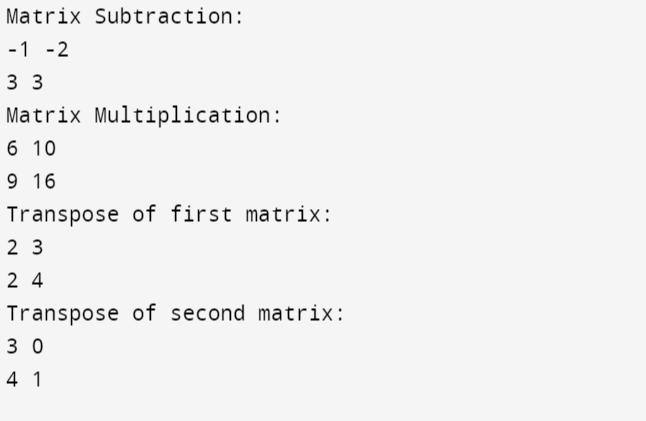
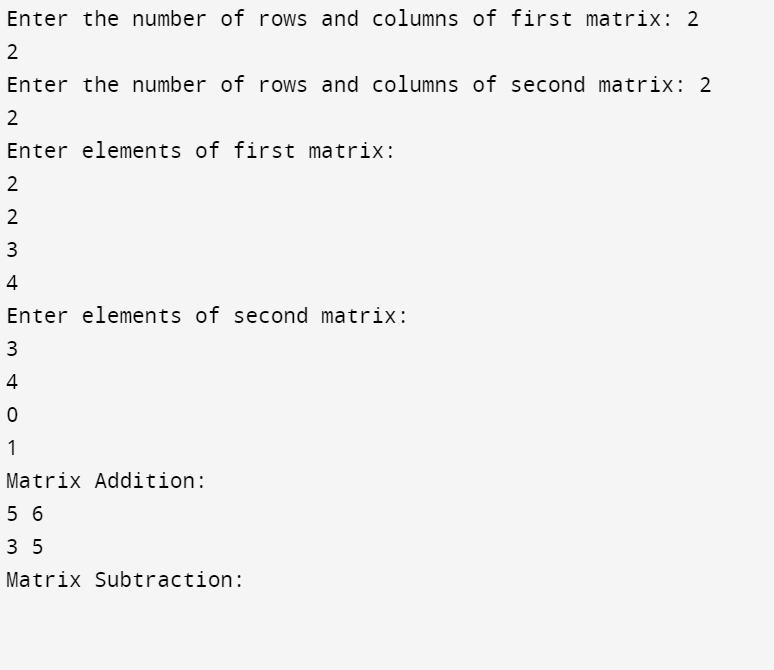
cout << "Transpose of first matrix:" << endl;

displayMatrix(transpose1, cols1, rows1);

cout << "Transpose of second matrix:" << endl;

displayMatrix(transpose2, cols2, rows2);

return 0;}



Part 2

Q1)--------------------------------------------------------------------------------------------------🡪

Pass By Value

Pass By value is used to pass the copy of the value

The function works on this copy not on the original value.

Changes in the parameter inside the function do not affect the original value outside the function.

Suitable for primitive data types and small-sized objects as copies are efficient

Pass by Reference:

When passing arguments by reference a reference of the original variable is passed to the function.

Any changes made to the parameter inside the function directly affect the original variable outside the function.

References are used to pass the original variable.

Suitable when you need to modify the original value inside the function without making copies.

Q2---------------------------------------------------------------------------------------------------------------------🡪

CLASS

1)all function and variables are private;

2)need to tell what need to be public

3)class also supports inheritance

Struct

1)all function and variables are public;

2)need to tell what need to be public and private

3)struct do not initially supports inheritance

Q4)------------------------------------------------------------------------------------------------------🡪

Constructor are used to initialize object variables for memry . They are of same name as class

They are member function . They are of 3 types .

1)default Constructor

2) paramaterized Constructor

3) copy Constructor

A constructor is a member function that gets invoked automatically when an object of a class is created.

They do not consist of any return type

If a class does not define any constructor, the compiler provides a default constructor, which initializes data members with default values

class MyClass {

public:

MyClass() { }

MyClass(int x, int y) {

}

};

Destructor

They also same name as class but used to free vthe memory . A destructor there is no parameter used .

They are prefixed with ~ operator

They are used to clean up the space and memory

class MyClass {

public:

MyClass() {

}

~MyClass() {

//code

}

};

In C++, constructors and destructors are special member functions of a class that are used to initialize and clean up class objects, respectively. Here are the key differences between them:

Constructor:

A constructor is a special member function that gets invoked automatically when an object of a class is created.

It is used to initialize the object's data members, allocate resources, and perform any necessary setup.

Constructors have the same name as the class and no return type (not even void).

Constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.

If a class does not define any constructor, the compiler provides a default constructor, which initializes data members with default values (e.g., default constructor initializes primitive types to zero).

Example:

cpp

Copy code

class MyClass {

public:

MyClass() {

}

// Parameterized constructor

MyClass(int x, int y) {

// Initialization code

}

};

Destructor:

A destructor is a special member function that gets invoked automatically when an object of a class goes out of scope, or when delete operator is called on a dynamically allocated object.

It is used to release resources, perform cleanup operations, or deallocate memory allocated by the constructor.

Destructors have the same name as the class prefixed with a tilde (~), and they do not take any parameters nor return any value.

Unlike constructors, destructors cannot be overloaded.

Example:

cpp

Copy code

class MyClass {

public:

// Constructor

MyClass() {

// Initialization code

}

// Destructor

~MyClass() {

// Cleanup code

}

};

Usage:

Constructors are typically used to initialize the object's state, set default values, and allocate resources.

Destructors are used to release resources, perform cleanup tasks, close files, and deallocate memory allocated during the object's lifetime.

Q5--------------------------------------------------------------------------------------------🡪

Inheritace is one of the pillar of c++ language.

It allows to inherit the property of parent into child

This enables code reuse and the creation of heirachy

Inheritance can be categorized into several types, including single inheritance, multiple inheritance, hierarchical inheritance, and multilevel inheritance. Let's focus on single inheritance and multiple inheritance:

Single Inheritance

Single inheritance involves one base class and one derived class.

The derived class inherits from the base class.

The derived class can extend the functionality of the base class by adding new members or by overriding existing members.

class Animal {

public:

void eat() {

cout << "Animal is eating" << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Dog is barking" << endl;

}

};

int main() {

Dog dog;

dog.eat();

dog.bark();

return 0;

}

Multiple Inheritance:

Multiple inheritance involves a derived class inheriting from two or more base classes.

The derived class inherits all the members of each base class.

It can lead to the diamond problem, where ambiguity arises if two or more base classes have a member with the same name.

class Parent1 {

public:

void display1() {

cout << "Display from Parent1" << endl;

}

};

class Parent2 {

public:

void display2() {

cout << "Display from Parent2" << endl;

}

};

class Child : public Parent1, public Parent2 {

public:

void display() {

cout << "Display from Child" << endl;

}

};

int main() {

Child obj;

obj.display1();

obj.display2();

obj.display();

return 0;

}

Q7)----------------------------------------------------------------------------------------🡪

Virtual function is used in run time polymorphism

, a virtual function is a member function declared within a base class that can be overridden by derived classes. When you call a virtual function through a pointer or reference to a base class object, the actual function that gets executed is determined at runtime based on the type of the object pointed to or referenced, rather than the type of the pointer or reference.

A function virtual void func() is to be made to make class use late binding but not early binding