Exercise:

CW Task Must complete:

**Task - 01: Design a Matrix Class with Multiple Constructors**  
Create a class Matrix that represents a 2D matrix of **double** values. It must handle **row-major** dynamic allocation internally.

**Details**:

* + **Constructors**:
    1. **Default constructor**: Initializes a 0x0 matrix (rows = 0, cols = 0) or a nullptr-managed array.
    2. **Parameterized constructor (rows, cols)**: Allocates a 2D matrix of size rows x cols. Initialize elements to 0.
    3. **Copy constructor**: Deep copy.
    4. **Move constructor**: Transfers ownership of the matrix data.
    5. **Destructor**: Deallocates the dynamic memory.
  + **Member functions**:
    1. getRows() and getCols() to return the dimensions.
    2. at(int r, int c): returns a reference to the element at row r, column c.
    3. fill(double value): fills the entire matrix with the provided value.
    4. transpose(): returns a **new** Matrix object that is the transpose of the current matrix. Make sure to print it as well.

**Task - 02: Implement a Polynomial Class**

1. **Overview**:  
   A Polynomial can be represented by its coefficients. For example,

a0​+a1​x+a2​x2+…+an​xn

Create a class Polynomial that stores these coefficients dynamically.

1. **Details**:
   * **Data members**:
     + A dynamically allocated array of double (for coefficients).
     + An int that represents the highest power (degree) of the polynomial.
   * **Constructors**:
     + **Default constructor**: Creates a polynomial of degree 0 with a coefficient array of size 1 (initialized to 0).
     + **Parameterized constructor**: Takes an int degree and an array of double for coefficients. Copies them dynamically.
     + **Copy constructor**: Deep copy of the coefficients.
     + **Move constructor**: Transfers ownership of the coefficient array.
     + **Destructor**: Cleans up the allocated memory.
   * **Member functions**:
     + int getDegree() const: returns the polynomial’s degree.
     + double evaluate(double x) const: computes the polynomial value at x.
     + Polynomial add(const Polynomial& other) const: returns a new Polynomial that is the sum of the current polynomial and other.
     + Polynomial multiply(const Polynomial& other) const: returns a new Polynomial that is the product of the two polynomials.

**Task - 03: Implement a DynamicArray Class with the “Rule of Five”**

1. **Overview**:  
   Create a class called DynamicArray that manages a **dynamic** array of integers.
   * The class should allocate and deallocate its own memory.
   * It must correctly handle copy and move operations.
2. **Details**:
   * Provide these **constructors and operators**:
     1. **Default constructor**: Initializes an empty array or an array of size zero.
     2. **Parameterized constructor**: Accept an integer n that indicates the array size, allocate an array of that size, and default-initialize all elements to 0.
     3. **Copy constructor**: Performs a deep copy of the array.
     4. **Move constructor**: Efficiently transfers ownership of the dynamic array from an rvalue object (leaving the source in a valid but empty state).
     5. **Copy assignment operator (operator=)**: Deep copies the array from one object to another, properly handling self-assignment.
     6. **Move assignment operator (operator=)**: Transfers ownership from an rvalue object, deallocating any previously held memory.
     7. **Destructor**: Deallocates the dynamic array.
   * Provide additional **member functions**:
     1. size(): returns the current size of the array.
     2. at(int index): returns a reference to the element at index (and possibly a const version for read-only).
     3. resize(int newSize): resizes the dynamic array to a new size, copying old data if newSize > 0.

H.W

Task - 01:

Your task is to create a Circle constructor that creates a circle with a radius provided by an argument. The circles constructed must have two getters getArea() (PIr^2) and getPerimeter() (2PI\*r) which give both respective areas and perimeter (circumference).

Task - 02:

Create an Account class that a bank might use to represent customers bank accounts. Include a data member to represent the account balance. Provide three member functions. Member function credit should add an amount to the current balance. Member function debits should withdraw money from the Account. Member function get Balance should return the current balance.

Task - 03:

Create A class called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four pieces of information as instance variables - a part number (type String), a part description (type String), a quantity of the item being purchased (type int) and a price per item (double). Your class should have a constructor that initialize the four instance variables. In addition, provide a method named getInvoiceAmount that calculates the invoice amount (i.e., multiples the quantity by the price per item), then returns the amount as a double value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0.0.

Task - 04:

A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise “Required copies not in stock” is displayed. Design a system using a class called books with suitable member functions and constructors.

Task - 05:

Write a class called CoffeeShop, which has three instance variables:

**Name:** a string (basically, of the shop)  
**Menu:** an array of items (of type MenuItem), with each item containing the item (name of the item), type (whether a food or a drink) and price.  
**Orders:** an empty array and seven methods:  
**addOrder:** adds the name of the item to the end of the orders array if it exists on the menu. Otherwise, return “This item is currently unavailable”  
**fulfillOrder:** if the orders array is not empty, return “The {item} is ready” . If the orders array is empty, return “All orders have been fulfilled”  
**listOrders**: returns the list of orders taken, otherwise, an empty array.  
**dueAmount**: returns the total amount due for the orders taken.  
**cheapestItem**: returns the name of the cheapest item on the menu.  
**drinksOnly**: returns only the item names of type drink from the menu.  
foodOnly: returns only the item names of type food from the menu.