Lab Task: 2D Array Operations with Header File

Objective

Create a header file `Array2D.h` to hold various functions for a 2D array in C++. Implement and test these functions in a main program file `main.cpp`.

Instructions

1. Create a header file named ` Array2D.h` containing the following functions:

- - void initArray(int v): Initializes the 2D array with all elements set to the integer value `v`.
- - void printArray(): Prints the array in a matrix format.
- - void printLeftDiagonal(): Prints the elements along the left diagonal of the array.
- - void printRightDiagonal(): Prints the elements along the right diagonal of the array.
- - void printMagicSquare(): Checks if the array is a magic square (where all rows, columns, and diagonals sum to the same value) and prints 'Magic Square' or 'Not a Magic Square'.
- - int sumUp(): Returns the sum of all elements in the array.
- - void printAlternate(): Prints the array like a chessboard, where only the elements in alternating cells (like black squares on a chessboard) are displayed.

2. Implement a main program file `main.cpp` that includes `MatrixOperations.h` and demonstrates each function.

3. Assume the array size is 3x3 for simplicity. Initialize it with sample values in `main.cpp` to verify the functions.

Example Output

Given the 3x3 array initialized with values:

Array = [[4, 9, 2], [3, 5, 7], [8, 1, 6]]

1. printArray()

Prints the array in a matrix format:

```
4 9 2
3 5 7
8 1 6
```

2. printLeftDiagonal()

Prints the left diagonal elements:

```
4
  5
    6
```

3. printRightDiagonal()

Prints the right diagonal elements:

```
    2
  5
8
```

4. printMagicSquare()

Checks if the matrix is a magic square:

```
Magic Square
```

5. sumUp()

Calculates the sum of all elements in the array:

```
Sum of all elements: 45
```

6. printAlternate()

Prints the array in a chessboard pattern (only alternate cells):

```
4   2
  5
8   6
```

1. Complete these tasks except magicSquare
2. Then read and practice the below document to understand Dynamic Memory allocation in C++ and its pros and cons. Please read carefully.
3. After you are complete learning dynamic memory allocation in 2D array, convert these functions in another header file named Array2DDynamic.h.
4. **Upload these 2 header files in portal before coming to class.**

Dynamic Memory Allocation for 2D Arrays in C++

In C++, dynamic memory allocation is particularly useful when the size of a 2D array is not known at compile-time or needs to be changed during runtime. Instead of declaring a static array with fixed dimensions, you can use pointers to allocate memory on the heap. This approach allows you to create flexible, resizable arrays.

Concept of Dynamic Memory Allocation for 2D Arrays

For a 2D array, we need to allocate memory for each row and each column dynamically. The common approach involves:

1. Creating an array of pointers, where each pointer represents a row.
2. Allocating memory separately for each row, which allows us to manage and manipulate each row independently.

Steps for Dynamic Allocation of a 2D Array

Suppose we want to create a 2D array of size m x n, where m represents the number of rows and n represents the number of columns.

- - Create an array of pointers (each pointer for a row).
- - Allocate memory for each row individually using a loop.
- - Access and modify the array elements using standard index notation.

Example Code: Dynamic Allocation of a 2D Array

The following example demonstrates the creation, use, and deletion of a dynamically allocated 2D array in C++.

```
#include <iostream>
using namespace std;

int main() {
    int m, n;
    cout << "Enter the number of rows: ";
    cin >> m;
    cout << "Enter the number of columns: ";
    cin >> n;

    // Step 1: Create an array of pointers (each pointer represents a row)
    int** array = new int*[m];

    // Step 2: Allocate memory for each row
    for (int i = 0; i < m; i++) {
        array[i] = new int[n];
    }

    // Initializing and displaying the array
    cout << "Enter elements of the array:\n";
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> array[i][j];
        }
    }
    // Displaying the array
    cout << "The 2D array is:\n";
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }

    // Step 3: Deallocate memory to avoid memory leaks
    for (int i = 0; i < m; i++) {
        delete[] array[i]; // Delete each row
    }
    delete[] array; // Delete the array of row pointers

    return 0;
}
```

Explanation of the Example

- - Memory Allocation: The line int** array = new int*[m]; allocates an array of m pointers, each capable of pointing to a row. The for loop then allocates memory for each row with array[i] = new int[n];, creating an array of n integers for each row.
- - Using the Array: The array is used in the same way as a regular 2D array with array[i][j].
- - Memory Deallocation: Deleting each row individually with delete[] array[i]; ensures all memory allocated for each row is released. delete[] array; then frees up the memory allocated for the row pointers.

Output Example

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter elements of the array:
1 2 3
4 5 6
The 2D array is:
1 2 3
4 5 6
```

Important Points to Remember

- - Always deallocate memory with delete[] after you're done to prevent memory leaks.
- - Dynamic allocation is more memory-efficient than static allocation, especially when dealing with large or flexible data structures.
- - This approach is particularly useful when the array size needs to be decided at runtime or may vary.