This slide set covers pointers and arrays in C++. You should read Chapter 8 from your Deitel & Deitel book.
Multidimensional arrays

In C++, to allocate a multidimensional array, you can use

- Fixed-size array declaration
- Dynamic allocation with `new`
- Combination of these two
  - Where one dimension is allocated with a fixed-size array declaration and another through dynamic allocation

We will talk about these cases separately

```cpp
void foo() {
  // These are the necessary declarations to represent different types of two-dimensional arrays.
  // We will discuss the necessary allocations and deallocations for each type in the next slides.

  // A two-dimensional fixed-size array
  int a1[3][4];

  // A pointer of pointers
  int** a2;

  // An array of pointers
  int* a3[3];
}
```

Similar to one dimensional arrays, memory for an array dimension is taken from

- The stack if it is a local declaration
- The heap if it is a dynamic allocation
Declaring fixed-size multidimensional arrays

- The size for each dimension should be specified at declaration
  - In standard C++, the size should be a positive integer (either a literal or a constant variable)
  - It cannot be changed throughout the execution

- All array items are kept in consecutive memory locations
  - C++ uses the row major order to keep the array items

- Multiple subscript operators are to be used to access the array items

```cpp
const int firstDim = 3;    // B1 and B2 are two-dimensional fixed-size arrays of Book objects. If these are local
const int secondDim = 4;  // declarations, both of their dimensions are kept in the stack.
Book B1[ firstDim ][ secondDim ];
Book B2[ 5 ][ 2 ];

// int no = 3;
// Book B3[ no ][ 4 ];
//
// delete []B2;
```

In standard C++, size of each dimension should be a literal or a constant variable. no is not a constant variable. **If you need to use a non-constant size, use a pointer and new operator**

Run-time error: B2 is not a dynamic array.
Initializing fixed-size multidimensional arrays

- If it is a local array declaration, array items have garbage values
- Array items can be initialized at array declaration using an initializer list
  - You may omit the size for the first dimension. In this case, the compiler determines this size based on the number of initializers. But, you have to specify the sizes for all other dimensions (otherwise, it causes a compile-time error).

```c
int a1[2][3] = {{ 1, 2, 3 }, { 4, 5, 6 }};
int a2[2][3] = {{ 1 }, { 2, 3 }};
int a3[2][3] = { 1, 2, 3, 4 };
int a4[][3] = {{ 1 }, { 2, 3 }};

// All declarations below give a compile-time error since either the list
// contains more initializers or the size for the 2nd dim is left as empty
// int b1[2][3] = {{ 1, 2, 3 }, { 4, 5, 6, 7 } };
// int b2[2][3] = {{ 1, 2, 3 }, { 4, 5, 6 }, {7, 8, 9 } };
// int b3[2][3] = { 1, 2, 3, 4, 5, 6, 7 };
// int b4[2][] = {{ 1, 2, 3 }, { 4, 5, 6 } };
// int b5[][] = {{ 1, 2, 3 }, { 4, 5, 6 } };
```
Passing fixed-size multidimensional arrays to functions

- Functions can take multidimensional arrays as arguments
- Function parameter list must specify an array as a parameter
  - In an array parameter declaration, the size of its first dimension is not required
  - However, the size of the subsequent dimensions are required (so that the compiler can know how many bytes to move for accessing the second item of the first dimension)
- The size of array dimensions should also be specified as parameters

```cpp
void displayArray( const int arr[][3], const int firstDim, const int secondDim ) {
    for ( int i = 0; i < firstDim; i++ ) {
        for ( int j = 0; j < secondDim; j++ )
            cout << arr[i][j] << "\t";
        cout << endl;
    }
}
```
Example: Extend the GradeBook class such that it keeps
the multiple grades of multiple students (use a fixed-size two-dimensional array)

class GradeBook{
public:
    const static int studentNo = 5;
    const static int examNo = 3;

    GradeBook( int, const int [][] examNo );
    void displayExamAverage();

private:
    int courseNo;
    int grades[ studentNo ][ examNo ];
};

GradeBook::GradeBook( int cno, const int arr[][ examNo ] ) {
    courseNo = cno;
    for ( int i = 0; i < studentNo; i++ )
        for ( int j = 0; j < examNo; j++ )
            grades[i][j] = arr[i][j];
}
```
// Global function to calculate the average of the items in a 1D array
double calculateAverage( const int arr[], int arrSize ){
    double avg = 0.0;
    for ( int i = 0; i < arrSize; i++ )
        avg += arr[i];
    if ( arrSize > 0 )
        return avg / arrSize;
    return 0;
}
void GradeBook::displayExamAverage(){
    for ( int i = 0; i < studentNo; i++ )
        // How to call calculateAverage for exam grades of each student?
        cout << ??? << endl;
}
int main() {
    int arr[ GradeBook::studentNo ][ GradeBook::examNo ];
    // ...
    GradeBook gb( 201, arr );
    gb.displayExamAverage();
    return 0;
}
```
Dynamic 2D arrays  
(*pointer to pointers*)

- Each dimension of a 2D array is dynamically allocated
  - Its first dimension is dynamically allocated to keep an array of pointers
  - Each array item in this allocated array keeps the starting address of another array that will be dynamically allocated

- All allocations are done using the `new` operator
  - Thus, memory is taken from the heap
  - This memory should be released by the `delete` operator

```cpp
void foo( int dim1, int dim2 ){
    // arr is a pointer of pointers
    // Since arr is a local variable, it is kept in the stack.
    int** arr;

    // First dimension is dynamically allocated (from the heap).
    arr = new int* [ dim1 ];

    // Second dimension is dynamically allocated (from the heap). Each array item in the first dimension keeps the starting address of each new allocation
    for ( int i = 0; i < dim1; i++ )
        arr[i] = new int [ dim2 ];

    // First, arrays corresponding to the second dimension should be deallocated.
    for ( int i = 0; i < dim1; i++ )
        delete [] arr[i];

    // Then, the array corresponding to the first dimension should be deallocated
    delete [] arr;
}
```
Dynamic 2D arrays
(array of pointers)

- First dimension of this 2D array is a fixed-size array
  - Thus, it is allocated by declaration
  - This array should NOT be deallocated by delete

- Each array item in the first dimension corresponds to a dynamically allocated array
  - Each keeps the starting address of an array that will be dynamically allocated by new
  - Each should be released by delete

```cpp
void foo(int dim2) {
    // arr is the name of a fixed-size array of pointers. The first dimension is allocated by declaration. The size used here should be constant. All pointers kept in this array are in the stack.
    const int dim1 = 5;
    int* arr[dim1];

    // Second dimension is dynamically allocated (from the heap). Each array item in the first dimension keeps the starting address of each new allocation
    for (int i = 0; i < dim1; i++)
        arr[i] = new int[dim2];

    // Arrays corresponding to the second dimension should be deallocated.
    for (int i = 0; i < dim1; i++)
        delete[] arr[i];

    // However, the array corresponding to the first dim should NOT be deallocated
}
```
**Example:** Write a global function that takes a square matrix as an input and returns its upper triangular part.

```c
int** takeUpperTriangular( int** mat, const int size ) {

    if ( size <= 0 )
        return NULL;

    int** upper = new int* [size];
    for ( int i = 0; i < size; i++ ) {
        upper[i] = new int [size - i];
        for ( int j = i; j < size; j++ )
            upper[i][j - i] = mat[i][j];
    }
    return upper;
}
```
**Example:** Write a global function that takes a 2D array as an input and deletes the last row and the last column of this 2D array. You may assume that rowNo > 0 and colNo > 0.

```c
void deleteLastRowLastColumn( int**& arr, int& rowNo, int& colNo ) {

    int** temp = arr;
    int tempRowNo = rowNo;
    if ( rowNo == 1 || colNo == 1 ) {
        arr = NULL;
        rowNo = colNo = 0;
    }
    else {
        arr = new int* [rowNo - 1];
        for ( int i = 0; i < rowNo - 1; i++ ) {
            arr[i] = new int [colNo - 1];
            for ( int j = 0; j < colNo - 1; j++ )
                arr[i][j] = temp[i][j];
        }
        rowNo--;  
        colNo--;  
    }
    for ( int i = 0; i < tempRowNo; i++ )
        delete [] temp[i];
    delete [] temp;
}
```