Multidimensional Arrays

Introduction

- A two-dimensional array can be declared as follows:
  - ```
    int chessTable[8][8];
    ```
  - This declares the chessTable array with 8 sets of 8 integers element. Note how each dimension is between its own pair of square brackets
  - You can visualize a two-dimensional array as a rectangular arrangement like, in our previous example, a table which has `8 rows` and each row has `8 columns`.
- Even though we can visualize them as rows and columns, in memory, all the elements are stored sequentially.
  - For example, let's suppos we have the following two-dimensional array: `int numbers[3][4]`
  - In memory, the elements are stored sequentially as follows:
    - `numbers[0][0] | numbers[0][1] | numbers[0][2] | numbers[1][0] | numbers[1][1] | numbers[1][2] | numbers[2][0] | numbers[2][1] | numbers[2][2]`
- Another way of visualizing a two-dimensional array is like an array of elements where each element is an array itself
  - A simple array is also called a `one-dimensional` array of elements

Initializing Multidimensional Arrays

- Initializing a two-dimensional array is similar to initialization of a one-dimensional array. The difference is that you put the initial values for each row between braces, `{}`, and then enclose all the rows between braces:
  - ```
    int numbers[3][4] = {
        {10,20,30,40}, // Values for first row
        {15,25,35,45}, // Values for second row
        {47,48,49,50} // Values for third row
    }
    ```
- You need a nested loop to process all the elements in a multidimensional array. Below you can see how you can sum the elements in a multidimensional array:
  - ```
    #include <iostream>
    using namespace std;
    ```
    ```

```
int main(){
    int numbers[3][4] = {
            {10,20,30,40}, // Values for first row
            {15,25,35,45}, // Values for second row
            {47,48,49,50} // Values for third row
    };
    int sum = 0;

    for(int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            sum += numbers[i][j];
        }
    }
    cout << "The sum of the elements is " << sum << endl;
    return 0;
}
```

Using the `sizeof` operator

- You can use the sizeof operator to work out the number of elements in each dimension. You just need to be a bit more careful on what the `sizeof` operator is producing:
  - Assuming we are using the same numbers multidimensional array from the previous example, the following statements are true:

    - `sizeof(numbers) / sizeof(numbers[0])` -> it tells you how many rows are in our multidimensional array.

      - The same principles as for the simple, one dimensional arrays, applies here when we compute this formula.
      - Let's suppose that `int` type occupies `4 bytes` in our computer
      - `numbers[3][4]` means that we have a total of `3x4` integer numbers which means 12 integers, which occupies `12x4 bytes`, which sums up to `48 bytes`, thus, `sizeof(numbers)` is `48 bytes`
      - The first row has `4 elements` of `int` type, which means that it occupies `4x4 bytes`, namely `16 bytes`, thus `sizeof(numbers[0])` is `16 bytes`
      - Now the final formula is translated to `48/16 => 3`, where 3 is the number of rows.

    - `sizeof(numbers[0] / sizeof(numbers[0][0]` -> it tells you how many elements (or columns) are in the first row (0 index);

      - Homework => proove that the above formula gives you how many elements are in the first row. Inspire from our previous example.

Accessing elements in a two-dimensional array

- As with simple arrays, or one-dimension array, in order to access an element in a two-dimension array, you have to specify two values:
  - one for the row of your element
  - and another one for the column of your element

- e.g: in order to access the element on the third row and on the 4th column, in a two-dimension array called numbers, we type: `numbers[3][4]`.

Class exercises

1. Write a C++ program which prints the element on the main diagonal of a two-dimension array with n rows and n columns.
   - Input:
     - ```
       1 2 3
       4 5 6
       7 8 9
       ```

       ```
       ```
   - Output: `The elements on the main diagonale are: 1 5 9`
   - Solution:
     - ```
       #include <iostream>
       using namespace std;
       int main(){
           int numbers[3][4] = {
                   {1,2,3},
                   {4,5,6},
                   {7,8,9}
           };
           cout << "The elements on the main diagonale are: ";
           for(int i = 0; i < 3; i++) {
               cout<<numbers[i][i]<< " ";
           }
           return 0;
       }
       ```
   - Explanation:
     - As you can see, the elements on the main diagonale are `1, 5, and 9`.
     - The indexes corresponding to these elements are:
       - `numbers[0][0]`
       - `numbers[1][1]`
       - `numbers[2][2]`
     - We can easily observe that the formula for a number to be on the main diagonale is `numbers[row][col]` where `row` and `col`, which represents the corresponding index of the row and column, have the same value.
     - In conclusion, we only need one loop to iterate through the values.

2. Write a C++ program which displays the numbers on the second diagonale of a two-dimension array.

   - Input:

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

     ```
     ```

   - Output: `The elements on the main diagonale are: 3 5 7`
   - Solution:

     - ```cpp
       #include <iostream>
       using namespace std;
       int main(){
           int numbers[3][3] = {
                   {1,2,3},
                   {4,5,6},
                   {7,8,9}
           };
           int numberOfRows = sizeof(numbers) / sizeof(numbers[0]);
           cout << "The elements on the second diagonal are: ";
           for(int i = 0; i < numberOfRows; i++) {
           cout<<numbers[i][numberOfRows-i-1]<< " ";
           }
           return 0;
       }
       ```

   - Explanation:
     - As we can see, the elements on the second diagonale are: `3,5, and 7`
     - Their corresponding indexes are:
       - `numbers[0][2] | numbers[1][1] | numbers[2][0]`
     - If we try to create a formula, based on the number of rows which gives us these number, we reach the following:
       - let `n` be the number of rows meaning `3`
         - for the first number we see that that we can obtain the indexes as follows
           - `3 (number of rows) - 0 - 1` for the row
           - `0` for the column
         - for the second number we can obtain the indexes by:
           - `3 (number of rows) - 1 -1` for the row
           - `1` for the column
         - for the third number we can obtain the indexes by:
           - `3 (number of rows) -2 -1` for the row
           - `2` for the column

- We can generalize the formula as follows:
  - `n-1-i` for the row index
  - `i` for the column index

3. Write a C++ program which reads a matrix from the standard input and computes its transpose.

   - Input:

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

   - Output:

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

   - Solution:

     - ```cpp
       #include <iostream>
       using namespace std;
       int main() {
           int numbers[3][3] = {
                   {1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}
           };

           int transpose[3][3];

           for(int i = 0; i < 3; i++){
               for (int j = 0; j < 3; j++) {
                   transpose[i][j] = numbers[j][i];
               }
           }

           for(int i = 0; i < 3; i++) {
               for (int j = 0; j < 3; j++) {
                   cout << transpose[i][j] << " ";
               }
               cout << endl;
           }
       }
       ```

   - Explanation:
     - By transpose of a matrix, we understand that the rows becomes columns and viceversa.
       ```

- To start, we declare another two dimensional array which will be able to store the transpose.
- Then we iterate through the elements of the original matrix, and at each step, we simply store at the current position (`transpose[i][j]`), the element which can be found at the location given by `j` for the rows and `i` for the columns

Homework exercises:

1. Write a C++ program which computes the average of the prime numbers which can be found on the main diagonale of a matrix (Two-Dimensional array)
   - Input:
     - ```
       1  2  3  4
       5  5  7  8
       9  10 11 12
       13 14 15 15
       ```
   - Output: `The average of the prime numbers on the main diagonale is 8`
2. Write a C++ program which computes the harmonic mean (media armonica) of the numbers on the second diagonale.
   - Input:
     - ```
       1  2  3  4
       5  5  7  8
       9  10 11 12
       13 14 15 15
       ```
   - Output: `The harmonic mean of the numbers on the second diagonale is: 8,08`
3. Write a C++ program which computes the sum of the numbers on the main diagonale and the sum of the numbers on the second diagonale and then displays which one is greater
   - Input:
     - ```
       1  2  3  4
       5  5  7  8
       9  10 11 12
       13 14 15 15
       ```
   - Output: `The second diagonale has the sum greater.`
4. Write a C++ program which displays if a certain number, read from the standard input, is present in a Two-Diagonale array. Also the program should display how many times the number has appeared in
       ```

the matrix

- Input: Enter the number to search for: 5 - `C++ 1 2 3 4 5 5 7 8 9 10 11 12 13 14 15 15`
- Output: `The number appears in the matrix 2 times`
- Input 2: Enter the number to search for: 51
  - ```
    1  2  3  4
    5  5  7  8
    9  10 11 12
    13 14 15 15
    ```
- Output: `The number could be found in the matrix`

Advanced exercises

1. Write a C++ program which displays the number below the main diagonal (including the ones on the diagonal itself). The numbers should be displayed in a nice and formatted way. See the output below
   - Input:
     - ```
       1  2  3  4
       5  5  7  8
       9  10 11 12
       13 14 15 15
       ```
   - Output:
     - ```
       1
       5   5
       9   10  11
       13  14  15  15
       ```
2. Write a C++ program which displays the prime numbers below the main diagonal (including the ones on the diagonal itself).
   - Input:
     - ```
       1  2  3  4
       5  5  7  8
       9  10 11 12
       13 14 15 15
       ```
   - Output:

- ```
  5 5 11 13
  ```
  ```