20b. Multi-dimensional Arrays

Objectives

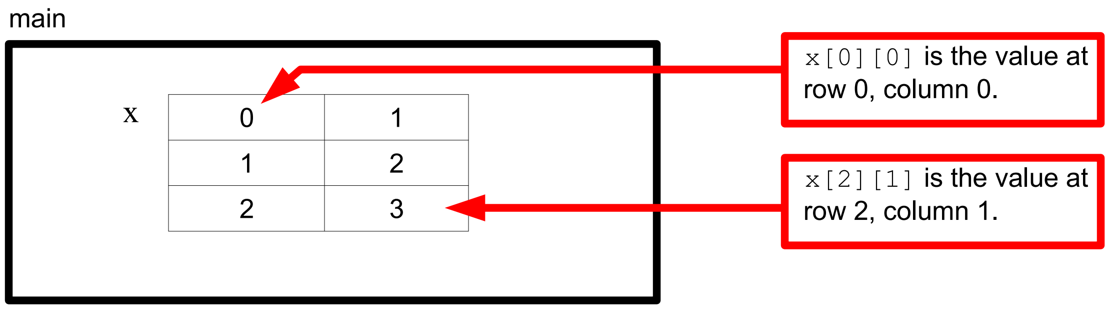
- Declare 2D arrays with or without initialization
- Get and set a value in a 2D array
- Scan a 2D array using loops
- Pass a 2D array to a function
- Work with an array of any dimension

2D Arrays

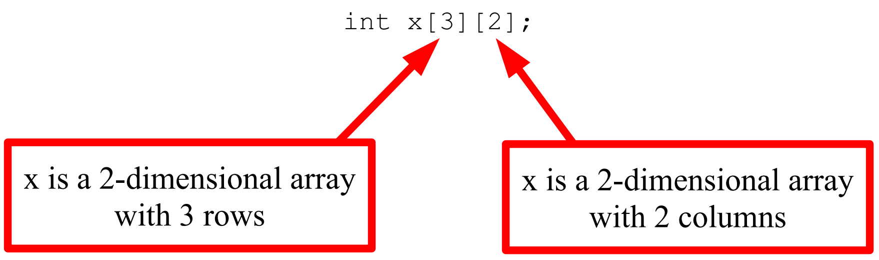
Once you understand a 1-dimensional (1D) array, a 2-dimensional (2D) array is a piece of cake. Try this:

```
int x[3][2];
x[0][0] = 0;
x[0][1] = 1;
x[1][0] = 1;
x[1][1] = 2;
x[2][0] = 2;
x[2][1] = 3;
```

Sometimes we visualize the a 2D array like this:


main
x
0
1
1
2
2
3
x[0][0] is the value at row 0, column 0.
x[2][1] is the value at row 2, column 1.

You can think of this array as having 3 rows and 2 columns.


int x[3][2];
x is a 2-dimensional array with 3 rows
x is a 2-dimensional array with 2 columns

Of course how you draw x is up to you. The computer actually stores the array in a different way.

Sometimes instead of saying “x is a 2-dimensional array of integers with size 3 for the first dimension and size 2 the second dimension” (phew) I will just say “x is a 3-by-2 array of integers”.

Exercise. Declare a 5-by-10 array y of doubles.

Initialization

Suppose you want to declare a 2D array of integers and initialize it to this:

0	0	0
0	0	0
0	0	0

Of course you can do this:

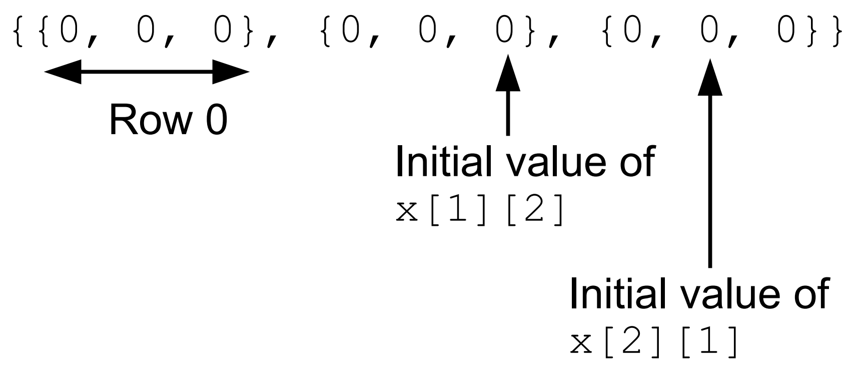
```
int x[3][3];
for (int row = 0; row < 3; row++)
{
    for (int col = 0; col < 3; col++)
    {
        x[row][col] = 0;
    }
}
```

But that would be a series of assignments and not initialization.

You can initialize using the initializer list again. But in this case you should do this:

```
int x[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
```

Of course:



Of course you can always write this (which is more readable) if you like:

```
int x[3][3] = {{0, 0, 0},
               {0, 0, 0},
               {0, 0, 0}};
```

The easiest way to initialize all the values in an integer array to zero is this:

```
int x[3][3] = {{0}};
```

This is similar to the case of 1D array where if you initialize some values in an array, then those without implicit initial values will be initialized to 0.

Exercise. Suppose you want to set the second row of x to the values 2, 3, 4. Can you do this:

```
int x[3][3] = {{0}};
x[2] = {2, 3, 4};
```

Exercise. Declare a tic-tac-toe game board as a 2D array of characters. The board is 3-by-3. Initialize all the values of the array to the space character. Print all the values in your array to verify your initialization.

Exercise. Declare a 3-by-3 array of characters and initialize it to this:

`' '`	`' '`	`' '`
`' '`	`'X'`	`' '`
`'X'`	`'X'`	`'X'`

Write a double for-loop to print the contents. (This is a tetrad in the Tetris game.)

Exercise. Continuing the above, set the array to the following with 4 assignment statements.

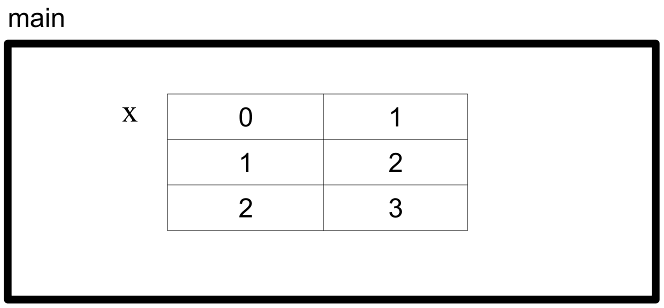
`' '`	`' '`	`'X'`
`' '`	`'X'`	`'X'`
`' '`	`' '`	`'X'`

Print the contents of the array to verify that your assignments work correctly.

Scanning 2D arrays: Row-by-row, column-by-column

Of course you can use loops to scan an array. Since there are two dimensions, it's common to scan the array with a double for-loop, one for each dimension.

```
int x[3][2];
for (int row = 0; row < 3; row++)
{
    for (int col = 0; col < 2; col++)
    {
        x[row][col] = row + col;
    }
}
```



This has the same effect as our previous program:

```
int x[3][2];
x[0][0] = 0;
x[0][1] = 1;
x[1][0] = 1;
x[1][1] = 2;
x[2][0] = 2;
x[2][1] = 3;
```

Here's the same code but with constants. I've also added a segment of code to print the values in the array. If I think of the first index as the "row" index and the second as the "column" index, the `row` and `col` variables will make the program easier to read too.

```
const int ROW_SIZE = 3;
const int COL_SIZE = 2;
int x[ROW_SIZE][COL_SIZE];

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        x[row][col] = row + col;
    }
}

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl; // goto next line
}
```

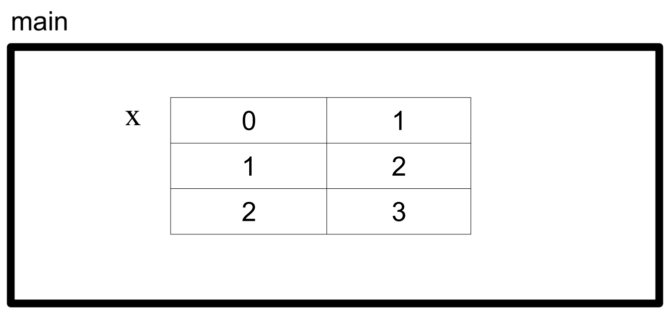
Note that with the constants, the program is more flexible to change.

Exercise. Change the `ROW_SIZE` to 10 and `COL_SIZE` to 15 and run the program. Get it?

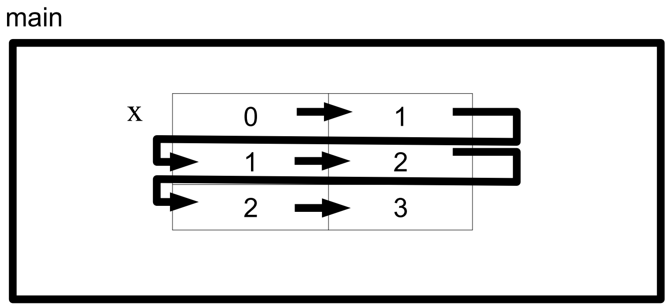
Look at the above double for-loops. They look like this:

```
for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        ... do something with x[row][col] ...
    }
}
```

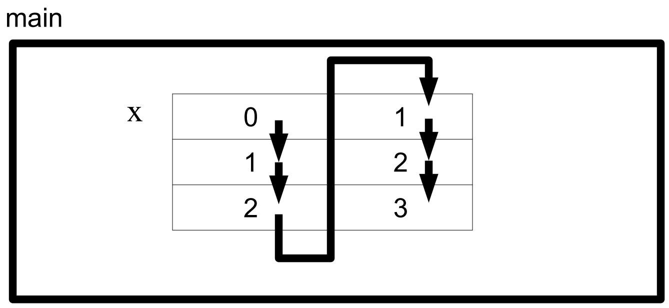
if you use this as your mental picture of the array, where the first index value represents the row and the second represents the column:



the scanning of the array looks like this:



Now suppose (for some reason) you scan the array in this order:



The second for-loop in this code scans the array this way:

```
const int ROW_SIZE = 3;
const int COL_SIZE = 2;
int x[NUM_ROWS][NUM_COLS];

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        x[row][col] = row + col;
    }
}

for (int col = 0; col < COL_SIZE; col++)
{
    for (int row = 0; row < ROW_SIZE; row++)
    {
        std::cout << x[row][col] << ' ';
    }
}
std::cout << std::endl;
```

So just like the 1-dimensional array where you can scan forward or backward, you can scan a 2-dimensional array in different ways.

Exercise. Write a while-loop that prompts the user for a row `r`, column `c` and value `v` and set `x[r][c]` to `v`. Print the whole array after each change to the array. If the user enters a negative value for `r` or `c`, exit the while-loop

```
const int ROW_SIZE = 3;
const int COL_SIZE = 2;
int x[ROW_SIZE][COL_SIZE];

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        x[row][col] = row + col;
    }
}

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}

while (1)
{
    // Prompt for r. If r < 0, break the loop.
    // Prompt for c. If c < 0, break the loop.
    // Prompt for v and set x[r][c] to v
    // Print the array
}
```

Test your program.

Exercise. Continuing with the above example, initialize the 3-by-3 tic-tac-toe board to

' '	'O'	' '
' '	'O'	' '
'X'	'X'	'X'

and write code to check that 'X' has a winning row. The code should detect a winning row for row 2. Now add code to detect either a winning row or winning column. The code should detect a winning column for this

' '	'O'	'X'
' '	'O'	'X'
' '	' '	'X'

Exercise. Declare an array of 5-by-10 integers. Assign random integers from 1 to 10 to the array. Print the array in a 5-by-10 grid of values separating values in each row by a space and separating each row with a newline. Compute and print the average of the values in the whole array.

Exercise. Declare an array of 5-by-10 integers. Assign random integers from 1 to 10 to the array. Print the array in a 5-by-10 grid of values separating values in each row by a space and separating each row with a newline. Prompt the user for a row and print the average of the values in that row. Repeat this exercise but let the user specify a ***column***.

Exercise. You work for the national weather service. You need to declare a 2D array of doubles for measuring temperature. For instance

```
temp[5][15]
```

is the temperature at 5:15 hours. Declare the array to measure temperature for a day. Prompt the user for a default temperature and set all the values in the array to this default temperature. In a loop, prompt the user for the hour and minute and temperature and set the array accordingly. If the user enters negative value for either the hour or the minute or the temperature, stop prompting the user and print the temperature in a two dimensional grid of values with hour for the rows and minute for the columns and print the average temperature.

Exercise. You're given this code:

```
const int ROW_SIZE = 5;
const int COL_SIZE = 5;
int x[ROW_SIZE][COL_SIZE]= {{0}};

// YOUR CODE

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

Write ***a single for-loop*** that sets first and last row of the array to all 9. In other words the output should be

```
9 9 9 9 9
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
9 9 9 9 9
```

Exercise. You're given this code:

```
const int ROW_SIZE = 5;
const int COL_SIZE = 5;
int x[ROW_SIZE][COL_SIZE]= {{0}};

// YOUR CODE
```

```
for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < ROW_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

Write two separate for-loops that sets first and last row and first and last column of the array to all 9. In other words the output should be

```
9 9 9 9 9
9 0 0 0 9
9 0 0 0 9
9 0 0 0 9
9 9 9 9 9
```

(If the array is a character array which is initialized to spaces, and 'X' are placed at the first and last row, first and last column, then the above shows you how to build a 2D world with boundary.)

Scanning 2D arrays: Exotic paths in the array

Now let's try scanning the array in a different way. As a matter of fact we'll only scan part of the array. You are given the following code which declares and initializes a 5-by-5 integer array with 0s (the row size is the same as the column size.)

```
const int ROW_SIZE = 5;
int x[ROW_SIZE][ROW_SIZE]= {{0}};

// YOUR CODE

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < ROW_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

We want to write a double for-loop that sets upper half right triangle of the array to all 9. In other words the output should be

```
9 9 9 9 9
0 9 9 9 9
0 0 9 9 9
0 0 0 9 9
0 0 0 0 9
```

Notice that the row and column index values where the value should be set to 9 are

row	col
0	0, 1, 2, 3, 4
1	1, 2, 3, 4
2	2, 3, 4
3	3, 4
4	4

If you write a double for loop that prints the above row,col values you can easily access the value of that index position and set the value to 9. So the first step is actually to write a program that prints

```
row:0
        col:0
        col:1
        col:2
        col:3
        col: 4
row:1
        col:1
```

col: 2
col: 3
col:4
etc.

But ... **WAIT!!!** You've actually seen something like this before ... in the notes on for-loops. The code is similar to this:

```
for (int i = 0; i < 5; i++)
{
    std::cout << "i:" << i << std::endl;
    for (int j = i; j < 5; j++)
    {
        std::cout << "  j:" << j << std::endl;
    }
}
```

If you don't see it immediately, you had better check your notes on for-loops … immediately!!!

Since we now have control over the index values that we want, we complete the code:

```
const int ROW_SIZE = 5;
int x[ROWS_SIZE][ROW_SIZE]= {{0}};

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = row; col < ROW_SIZE; col++)
    {
        x[row][col] = 9;
    }
}

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < ROW_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

Exercise. You're given this code:

```
const int ROW_SIZE = 5;
int x[ROW_SIZE][ROWS_SIZE]= {{0}};

// YOUR CODE

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < ROW_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

Write a double for-loop that sets lower half left triangle of the array to all 9. In other words the output should be

```
9 0 0 0 0
9 9 0 0 0
9 9 9 0 0
9 9 9 9 0
9 9 9 9 9
```

Exercise. You are given the following code:

```
const int ROW_SIZE = 5;
int x[ROW_SIZE][ROW_SIZE]= {{0}};

// YOUR CODE

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < ROW_SIZE; col++)
    {
        std::cout << x[row][col] << ' ';
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

Write a for-loop that sets the values on the main diagonal of the array to all 9s. In other words the output should be

```
9 0 0 0 0
0 9 0 0 0
0 0 9 0 0
0 0 0 9 0
0 0 0 0 9
```

[Hint: What index values do you need? You need row = 0, col = 0, then row = 1, col = 1, then row = 2, col = 2, then row = 3, col = 3 and finally row = 4, col = 4.]

Exercise. You are given the following code that declares a 5-by-5 array of characters, initializing it with spaces.

```
const int ROW_SIZE = 5;
char x[ROW_SIZE][ROW_SIZE];
for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < COL_SIZE; col++)
    {
        x[row][col] = ' ';
    }
}

// YOUR CODE

for (int row = 0; row < ROW_SIZE; row++)
{
    for (int col = 0; col < ROW_SIZE; col++)
    {
        std::cout << x[row][col];
    }
    std::cout << std::endl;
}
std::cout << std::endl;
```

Write a for-loop that sets the values on the main diagonal of the array to `'*'`. In other words the output should be

```
*
 *
  *
   *
    *
```

[Hint: See previous example.] Now modify your code so that it produces this output:

```
*   *
 * *
  *
 * *
*   *
```

Of course that tells you that you can use a 2D array to do ASCII art!!!You can think of the 2D array as a drawing canvas. There is a big difference between the ASCII art problems in the notes for loops and using a 2D array. You can draw a star any where you like in the array. On the other hand, ASCII art using only loops and print statements requires you to print on the console window from left-to-right, row-by-row.

There is however a restriction when using arrays for ASCII art: You have to specify the row and column sizes of your array. This is what you can do: You can specify a really huge 2D array (say 1000-by-1000) and then only use part of it for drawing.

Functions

Everything is similar to functions for 1-dimensional arrays except for one thing. **LISTEN UP!!!** When you wrote function prototypes or function headers to be general, instead of writing

```
void f(int x[3]);
```

we write

```
void f(int x[]);
```

When declaring a 2D array (of ints or doubles or bools or ...) you **must specify the size of the second dimension**. For instance this is **WRONG**

```
void f(int x[][]);
```

This is **CORRECT**:

```
void f(int x[][42]);
```

Here's a simple example:

```
#include <iostream>

const int COL_SIZE = 2;


void print(int x[][COL_SIZE], int numRows)
{
    for (int row = 0; row < numRows; row++)
    {
        for (int col = 0; col < COL_SIZE; col++)
        {
            std::cout << x[row][col] << ' ';
        }
        std::cout << std::endl; // goto next line
    }
}


int main()
{
    int x[2][COL_SIZE];
    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = 3;
    x[1][1] = 4;
    print(x, 2);
    return 0;
}
```

Note that the code now seems to be highly “nonsymmetric”: you do not have control over the second dimension.

To make the code more symmetric, you might want to do this, especially if you do not need full generality for the first dimension:

```
#include <iostream>

const int ROW_SIZE = 2;
const int COL_SIZE = 2;


void print(int x[ROW_SIZE][COL_SIZE])
{
    for (int row = 0; row < ROW_SIZE; row++)
    {
        for (int col = 0; col < COL_SIZE; col++)
        {
            std::cout << x[row][col] << ' ';
        }
        std::cout << std::endl; // goto next line
 }
}


int main()
{
  int x[ROW_SIZE][COL_SIZE];
  x[0][0] = 1;
  x[0][1] = 2;
  x[1][0] = 3;
  x[1][1] = 4;
  print(x);
  return 0;
}
```

Exercise. Can you change values in a 2D array through a function? Write a simple program and verify.

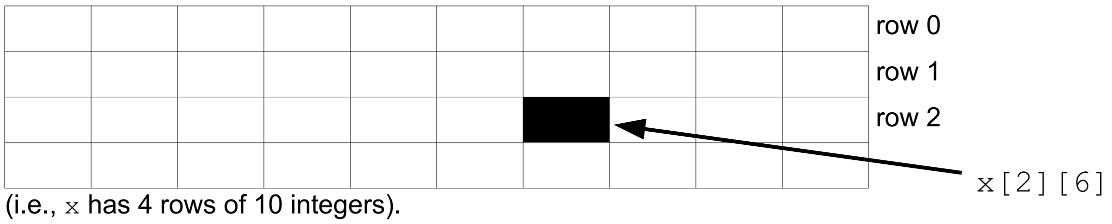
“Why do I need to specify the size for the second dimension?”

Why? You have to wait till CISS3 ... oh well ... OK I'll explain.

Suppose you have this code:

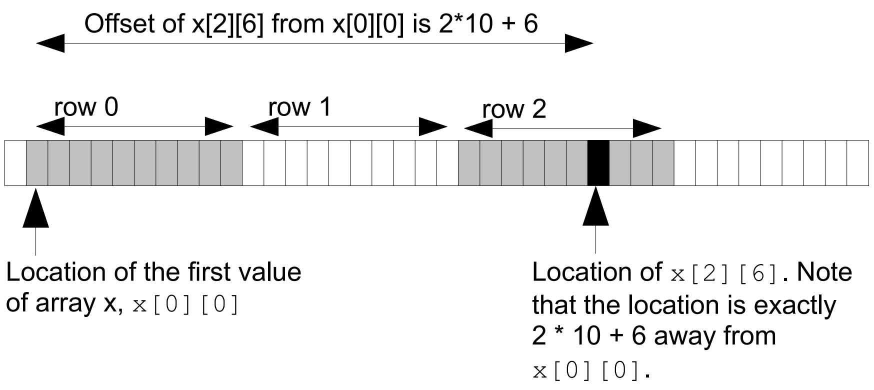
```
...                                                          ...
    int x[4][10];
    std::cin >> x[2][6];
```

First of all, our mental picture of the values of array x looks like this:



(i.e., x has 4 rows of 10 integers).

Actually in your computer (at least up to this point in history!), memory is organized in a **linear** fashion, i.e., **on a straight line**. So here's a picture of the memory with the part occupied by x shown. I've colored alternate blocks of 10 integers. The location in the computer's memory for x[2][6] is shown.


Offset of x[2][6] from x[0][0] is 2*10 + 6
row 0
row 1
row 2
Location of the first value of array x, x[0][0]
Location of x[2][6]. Note that the location is exactly 2 * 10 + 6 away from x[0][0].

How does C++ locate the value in the array that you want? C++ actually only remembers the location of x[0][0].When you want x[2][6], C++ computes the position of x[2][6] as an offset from x[0][0].

Now the location of x[2][6] is

2 * 10 + 6

away from the first value x[0][0]. Right? Once C++ knows the computer memory location of the value you want, it can then retrieve the value or assign a value to that spot.

So ... one more time:

- Values are stored in memory which is organized in a straight line.
- For an array, C++ remembers the position of the first element of the array.
- If your program needs `x[i][j]` of an array x, C++ will compute the location of that value by computing it's offset from `x[0][0]`.

OK. So now look at the above example:

```
int x[20][10];
std::cin >> x[2][6];
```

If you need `x[2][6]`, C++ will look at the value that is

2 * 10 + 6

away from `x[0][0]`. But ... notice something? The offset is

2 * 10 + 6

which depends on 2 and 6 which is provided in your code when you write `x[2][6]`. But notice that it also depends on ... **the number 10:**

2 * **10** + 6

Where does that come from? It's the **size of the second dimension**

This is the reason why the size of the second dimension must be stated. Otherwise C++ does not know how to access a required value in the 2D array.

Exercise: tic-tac-toe

Write a tic-tac-toe program using a 2D array for the board.

Exercise. Write a function

```
const int n = 3;

void init(char board[n][n]);
```

that initialization a tic-tao-toe board.. Note your program should have a global constant for the board size. That way if we change 3 to another size such as 5, you'll get a 5-by-5 tic-tac-toe board. So in your code do not assume the board size is 3-by-3.

Exercise. Write a function to print the tic-tac-toe board:

```
...
void print(char board[n][n]);
```

Suppose the board is filled with spaces and n is 3. Then the output is

```
+-+-+-+
| | | |
+-+-+-+
| | | |
+-+-+-+
| | | |
+-+-+-+
```

If row 0 and column 1 of the board has 'X', then the output is

```
+-+-+-+
| |X| |
+-+-+-+
| | | |
+-+-+-+
| | | |
+-+-+-+
```

Exercise. Write a function that checks if there's a winning row at row r for character `player`:

```
bool has_winning_row(char board[n][n], int r,
                     char player);
```

The character `player` is either `'X'` or `'O'`.

Exercise. Write a function that checks if there's a winning column at column c for character `player`:

```
bool has_winning_column(char board[n][n], int c,
                        char player);
```

Character `player` is either `'X'` or `'O'`.

Exercise. Write a function that checks if there's a winning down-diagonal for character `player`:

```
bool has_winning_down_diag(char board[n][n], int c,
                           char player);
```

Character `player` is either `'X'` or `'O'`. The down-diagonal is the diagonal on the board that goes from row 0, column 0 to row n – 1, column n – 1.

Exercise. Write a function that checks if there's a winning up-diagonal for character `player`:

```
bool has_winning_up_diag(char board[n][n], int c,
                         char player);
```

Character `player` is either `'X'` or `'O'`. The up-diagonal is the diagonal on the board that goes from row n-1, column 0 to row 0, column n – 1.

Exercise. Write a function that checks if the board if filled (i.e. no spaces left):

```
bool is_filled(char board[n][n]);
```

Character `player` is either `'X'` or `'O'`. The up-diagonal is the diagonal on the board that goes from row n-1, column 0 to row 0, column n – 1.

Make sure you test the above function thoroughly. Now complete the turn-based 2-player tic-tac-toe-game!

Exercise. The above is a 2-player game. Try to write a tic-tac-toe game for 1 player playing against the computer.

Exercise. Now write a 2-players chess game!

A 2-d array as an array of 1-d arrays

If you have a 2-d array like this:

```
      char h[][3] = {{'*', ' ', '*'},
                     {'*', ' ', '*'},
                     {'*', '*', '*'},
                     {'*', ' ', '*'},
                     {'*', ' ', '*'}};
```

You can actually think of `h[0]` (the first row) as an array of 3 characters, `h[1]` (the second row) as an array of 3 characters, etc. You can therefore do this:

```
#include <iostream>

void print3char(char x[])
{
    for(int i = 0; i < 3; i++)
    {
        std::cout << x[i]
    }
    std::cout << std::endl;
}

int main()
{
    char h[][3] = {{'*', ' ', '*'},
                   {'*', ' ', '*'},
                   {'*', '*', '*'},
                   {'*', ' ', '*'},
                   {'*', ' ', '*'}};

    print3char(h[0]);

    return 0;
}
```

Exercise. Write a program that prompts the user to enter his/her first name and then last name and stores them both in an array called name. `name` should be declared like so:

```
char name[2][256];
```

Print the name that they entered with a “Hi” – see execution below.

```
What’s you name? John Doe
Hi John Doe.
```

Exercise.
Write a program that prompts the user for a sentence ending with a period of question mark. The program then prints the number of words entered by the user and list the words. Here’s an execution:

```
Hi my name is John.
5 words
1. Hi
2. my
3. name
4. is
5. John
```

Here’s another:

```
How are you?
3 words
1. How
2. are
3. you
```

The user can enter at most 100 words.

Exercise.
Write a program that asks for a name and then say hi. Here’s an execution

```
What’s your name?
My name is John.
Hi John.
```

Here’s another:

```
What’s your name?
I’m Julie.
Hi Julie.
```

And another:

```
What’s your name?
Tom
Hi Tom. Please learn to use the period.
```

And another:

```
What’s your name?
I am tom.
Hi Tom. Please learn to capitalize your name.
```

Even more dimensions!

Is it too surprising for you when I say that you can have a 3-dimensional array like this:

```
      double x[10][15][20];
```

Exercise. Using a triple for-loop, initialize all values in the above x to 3.14.

The 2-dimensional array helps us model data that is inherently 2-dimensional. For instance:

```
      const int FLOOR_SIZE = 5;
      const int NUM_ROOMS = 20;
      int max_occupancy[FLOOR_SIZE][NUM_ROOMS];
```

models the maximum occupancy for rooms in a building with 5 floors and 20 rooms per floor. The data is inherently 2-dimensional. For instance `max_occupancy[3][19]` is the maximum occupancy of ROOM319 in the building.

But three dimensional data occurs too. For instance if you want to model the presence of physical bodies in a 3-dimensional grid of 1-by-1-by-1 cubic feet of space, you can have a 3-dimensional array of booleans:

```
      const int MAX_X = 100;
      const int MAX_Y = 100;
      const int MAX_Z = 100;
      bool has_body[MAX_X][MAX_Y][MAX_Z];
```

Or for instance in the Tetris game, you have tetrads. Each tetrad is a 2-dimensional shape. For instance there's the T tetrad:

' '	' '	' '
' '	'X'	' '
'X'	'X'	'X'

But each tetrad has 4 orientations. Here are the other three orientations of the T tetrad:

' '	' '	'X'
' '	'X'	'X'
' '	' '	'X'

'X'	'X'	'X'

' '	'X'	' '
' '	' '	' '

'X'	' '	' '
'X'	'X'	' '
'X'	' '	' '

Therefore all the diagrams for the T tetrad actually can be modeled with a 3-dimensional array of characters:

```
const int ROTATIONS = 4;
char t_tetrad[ROTATIONS][3][3] =
   {
    {{' ', ' ', ' '},
     {' ', 'X', ' '},
     {'X', 'X', 'X'}},
    {{' ', ' ', 'X'},
     {' ', 'X', 'X'},
     {' ', ' ', 'X'}},
    {{'X', 'X', 'X'},
     {' ', 'X', ' '},
     {' ', ' ', ' '}},
    {{'X', ' ', ' '},
     {'X', 'X', ' '},
     {'X', ' ', ' '}}
   };
```

Exercise. Prompt the user for an integer from 0 to 3 and print the corresponding T tetrad. There should only be one single triple for-loop (not 4). For instance if the user enters 0 the program should print:

```

 X
XXX
```

If the user enters 1 the program should print

```
  X
 XX
  X
```

Squeezing the initializer

By the way for the 2-dimensional and 3-dimensional case, you can actually initialize them with a 1-dimensional initializer. This is what I mean:

First try this:

```
char t[3][3] = {{'O', 'X', ' '},
                {'X', 'O', ' '},
                {' ', 'X', 'O'}};

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        std::cout << t[i][j];
    }
    std::cout << std::endl;
}
```

No surprises. Now try this:

```
char t[3][3] = {'O', 'X', ' ',
                'X', 'O', ' ',
                ' ', 'X', 'O'};

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        std::cout << t[i][j];
    }
    std::cout << std::endl;
}
```

As you can see, I'm using a 1-dimensional initializer to initialize my 2-dimensional array. You can do that. It's not wrong. However it does make the program a little confusing sometimes. But I leave it to you. This implies that another way to initialize a 2-dimensional array of integers to 0 is this:

```
int x[10][10] = {0}; // do not need {{0}}
```

And for a 3-dimensional case you can do this:

```
int x[10][10][10] = {0};
```

Exercises

Exercise. Declare an array of 5-by-5 characters and write loops to assignment values to the array so that it looks like this:

'X'	'X'	'X'	'X'	'X'
'X'	' '	' '	' '	'X'
'X'	' '	' '	' '	'X'
'X'	' '	' '	' '	'X'
'X'	'X'	'X'	'X'	'X'

(This builds a wall for a 5-by-5 maze.) Rewrite the code so that it works for any size by using a constant for the size.

Exercise. Declare an array of 5-by-10 integers. Put random integers 0-9 into the array. Print the array. Write a code segment that swaps the values of row 2 and 3. Print the array again and visually verify that row 2 and 3 are swapped.

Exercise. Write a function that accepts a 4-by-4 array of integers and returns true if for each row, the first non-zero value of each row is to the right of the previous one. For instance the following will return true:

1	3	3	4
0	0	3	4
0	0	0	2
0	0	0	0

Exercise. Write a function that accepts a 4-by-4 array of integers and returns true if and only if array as a matrix is symmetric. For instance, the following is symmetric:

1	3	3	4
3	2	1	8
3	1	0	2
4	8	2	7

Think of the diagonal with values 1, 2, 0, 7 as a mirror. So the example is symmetric since for example the entry at row 0, column 3 is the same as the entry at row 3, column 0, i.e. 4.

Exercise. Given an integer array of size m-by-m, write a code segment to compute the sum of the entries on the main diagonal. For instance for the array:

1	2	3
4	5	6

7	8	9

The sum is 1 + 5 + 9.

Exercise. Given an integer array of size m-by-m, write a code segment to compute the sum of the entries on and above the main diagonal. For instance for the array:

1	2	3
4	5	6
7	8	9

The sum is 1 + 2 + 3 + 5 + 6 + 9.