2D Arrays

Due this week

- **Project 1**
  - It’s been released, so start working on it
  - There will be a mandatory interview grading (will provide more information on Friday)
  - Discuss with us regarding any concerns you may have

2D Arrays

Two-Dimensional Arrays

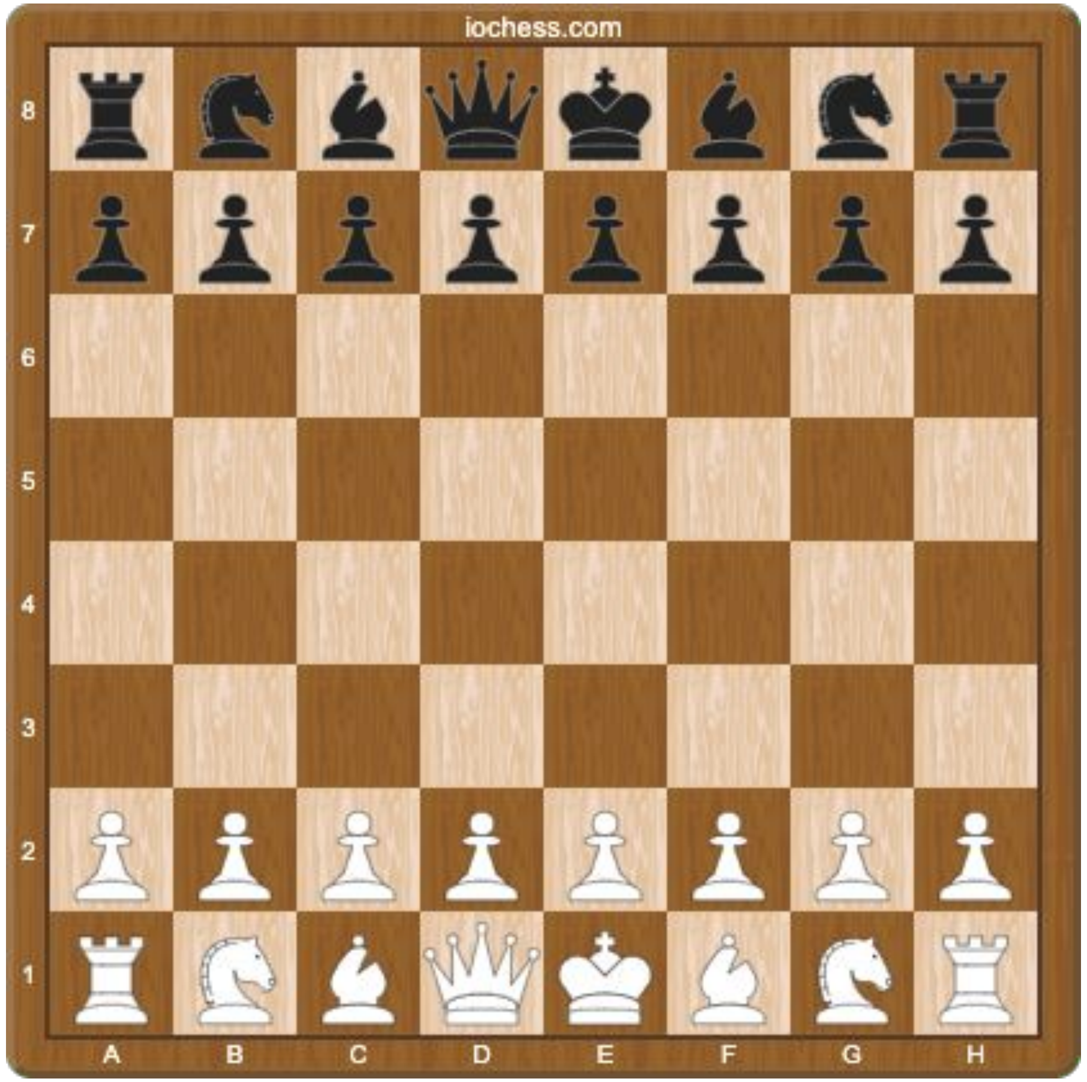
- It often happens that you want to store collections of values that have a two-dimensional layout.

- Such data sets commonly occur in financial and scientific applications. Let’s look into a few of such scenarios

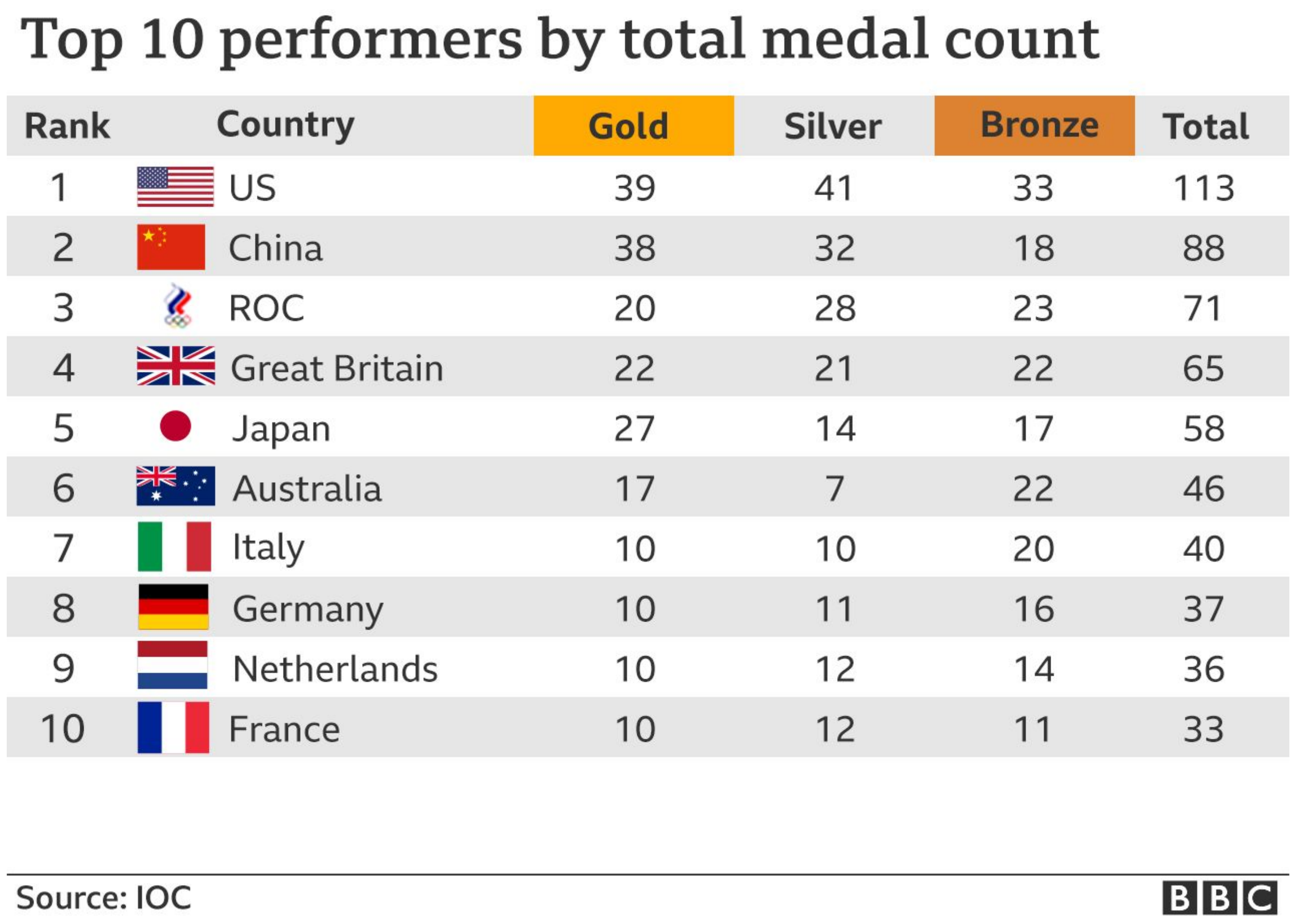
- An arrangement consisting of *tabular data (rows and columns* of values) is called:

  a ***two-dimensional array***, or a ***matrix***

The chessboard



Olympics Medal Count



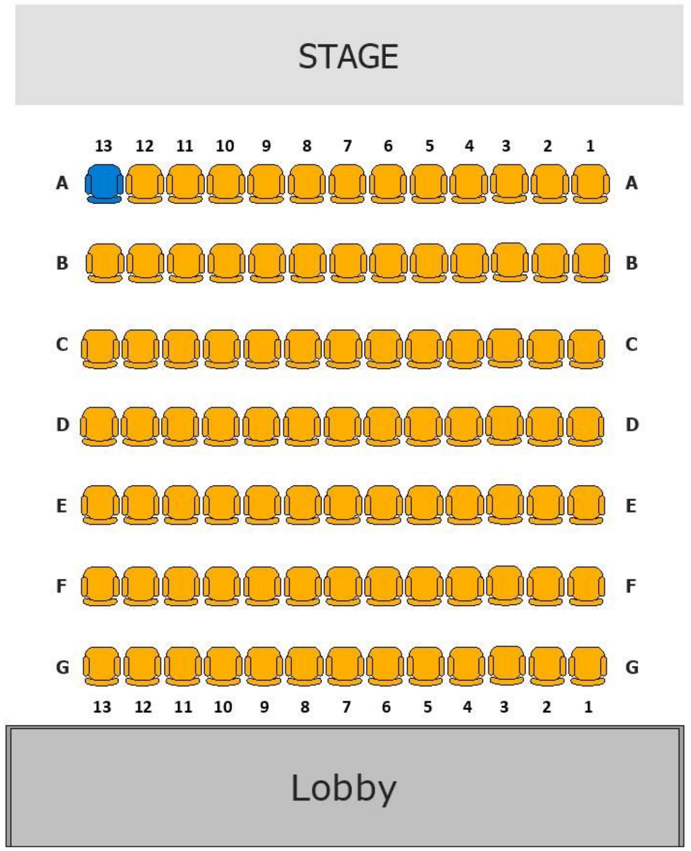
Top 10 performers by total medal count

Rank	Country	Gold	Silver	Bronze	Total
1	US	39	41	33	113
2	China	38	32	18	88
3	ROC	20	28	23	71
4	Great Britain	22	21	22	65
5	Japan	27	14	17	58
6	Australia	17	7	22	46
7	Italy	10	10	20	40
8	Germany	10	11	16	37
9	Netherlands	10	12	14	36
10	France	10	12	11	33

Source: IOC

BBC

A Seat Matrix



How we define 2D arrays

C++ uses an array with *two* subscripts to store a *2D* array.

For instance,

```
const int COUNTRIES = 10;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 10 rows and 3 columns is suitable for storing our medal count data.

Initializing 2D arrays

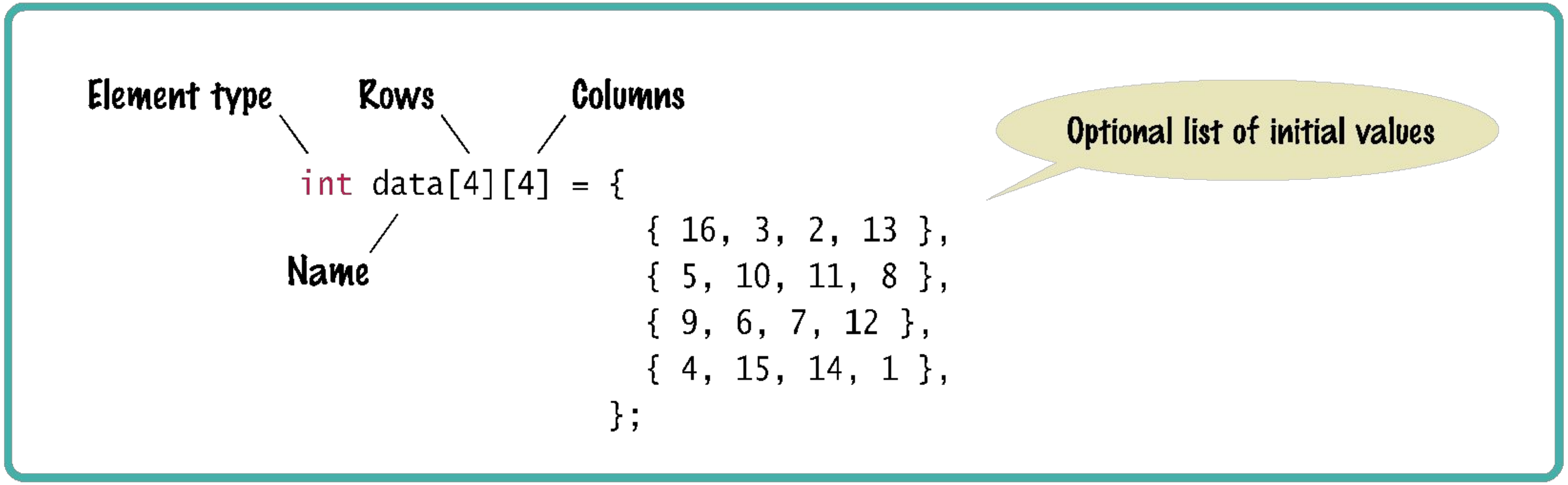
Just as with one-dimensional arrays, you ***cannot*** change the size of a two-dimensional array once it has been defined.

You can initialize them.

```
int counts[10][3] =
  {
     { 39, 41, 33 },
     { 38, 32, 18 },
     { 20, 28, 23 },
     { 22, 21, 22 },
     { 27, 14, 17 },
     { 17, 7, 22 },
     { 10, 10, 20 },
     { 10, 11, 16 },
     { 10, 12, 14 },
     { 10, 12, 11 }
  };
```

A look at 2D array definition

Two-Dimensional Array Definition



```
int data[4][4] = {
                   { 16, 3, 2, 13 },
                   { 5, 10, 11, 8 },
                   { 9, 6, 7, 12 },
                   { 4, 15, 14, 1 },
                 };
```

Copyright © 2017 by John Wiley & Sons. All rights reserved

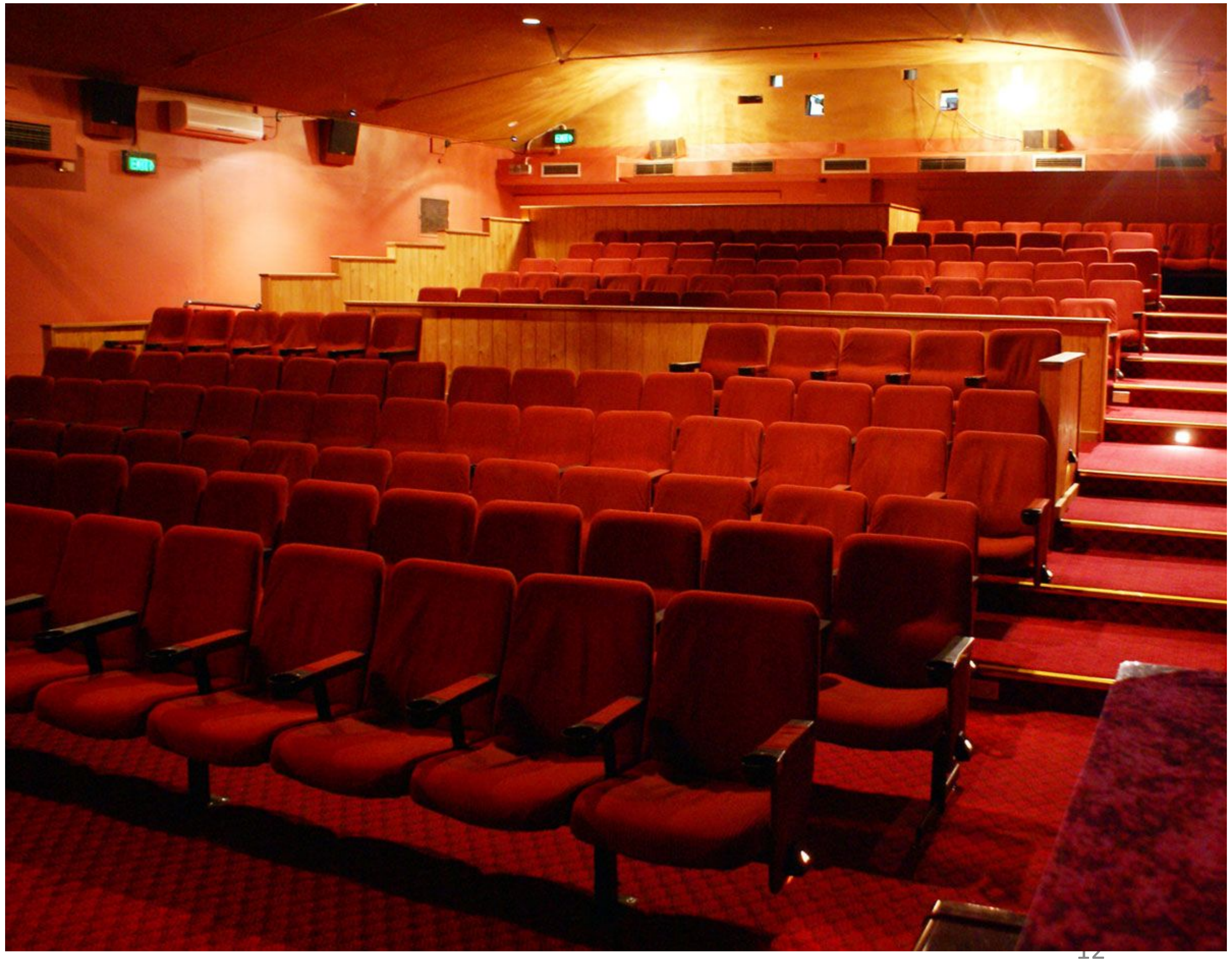
Test your understanding

- Identify the scenarios where you can use 2D arrays!
  - Movie theatre seat matrix
  - A tic tac toe board
  - Images! How??
  - Educational records
  - Geolocations

Anything that is represented as a grid is a 2D array!

Two-Dimensional Arrays – Accessing Elements

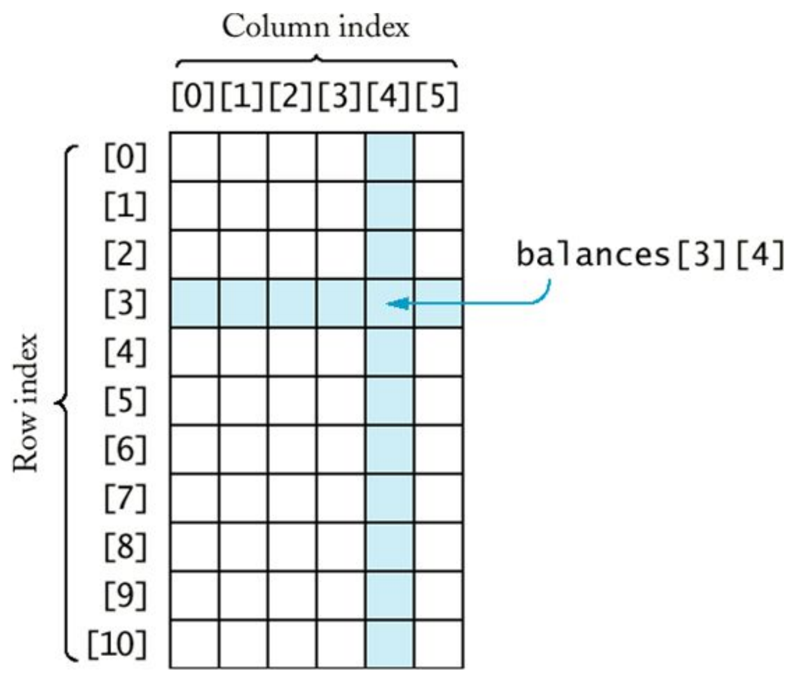
Think of this like you're trying to find your seat in a theatre



Two-Dimensional Arrays – Accessing Elements

```
int value = balances[3][4];
cout << value;

// change the element
balances[3][4] = 100;
```

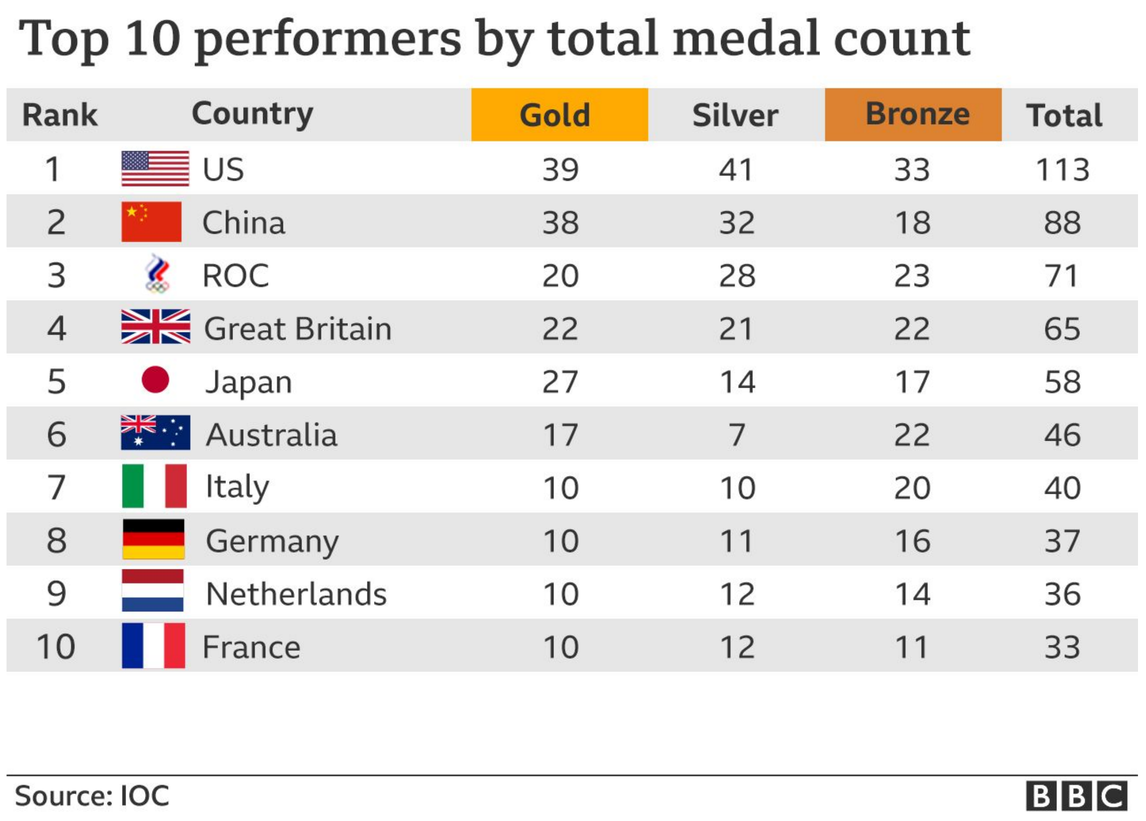
Give the row and the column index of the element you wish to access in square brackets.



Print 1st row elements in a 2D Array

```
[0][0], [0][1], [0][2]
// Print the 1st row:
cout << setw(8) << counts[0][0];
cout << setw(8) << counts[0][1];
cout << setw(8) << counts[0][2];
```

You get the picture of how to print elements now. Like before, provide the row and the column index.



Top 10 performers by total medal count

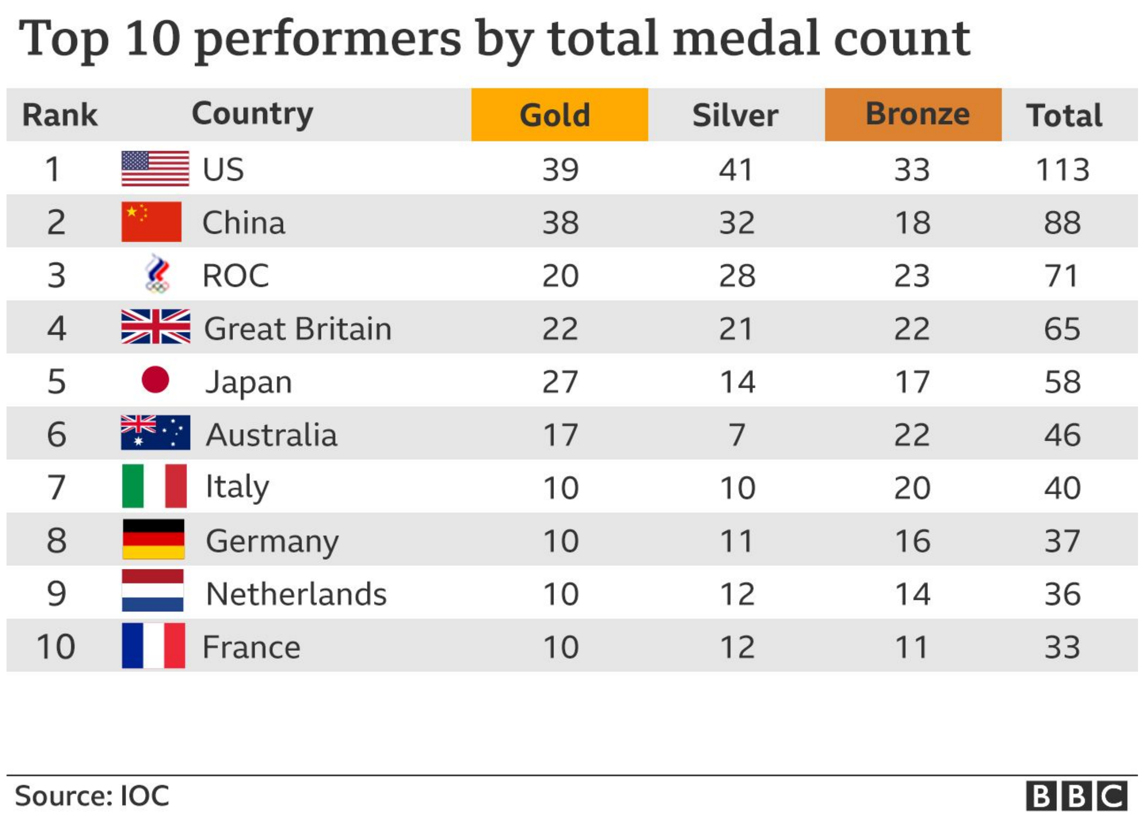
Rank	Country	Gold	Silver	Bronze	Total
1	US	39	41	33	113
2	China	38	32	18	88
3	ROC	20	28	23	71
4	Great Britain	22	21	22	65
5	Japan	27	14	17	58
6	Australia	17	7	22	46
7	Italy	10	10	20	40
8	Germany	10	11	16	37
9	Netherlands	10	12	14	36
10	France	10	12	11	33

Source: IOC

BBC

Print 1st row elements in a 2D Array (using loop)

```
[0][0], [0][1], [0][2]
// Process the 1st row:
for (int j = 0; j < MEDALS; j++)
{
  cout << setw(8) << counts[0][j];
}
```



Top 10 performers by total medal count

Rank	Country	Gold	Silver	Bronze	Total
1	US	39	41	33	113
2	China	38	32	18	88
3	ROC	20	28	23	71
4	Great Britain	22	21	22	65
5	Japan	27	14	17	58
6	Australia	17	7	22	46
7	Italy	10	10	20	40
8	Germany	10	11	16	37
9	Netherlands	10	12	14	36
10	France	10	12	11	33

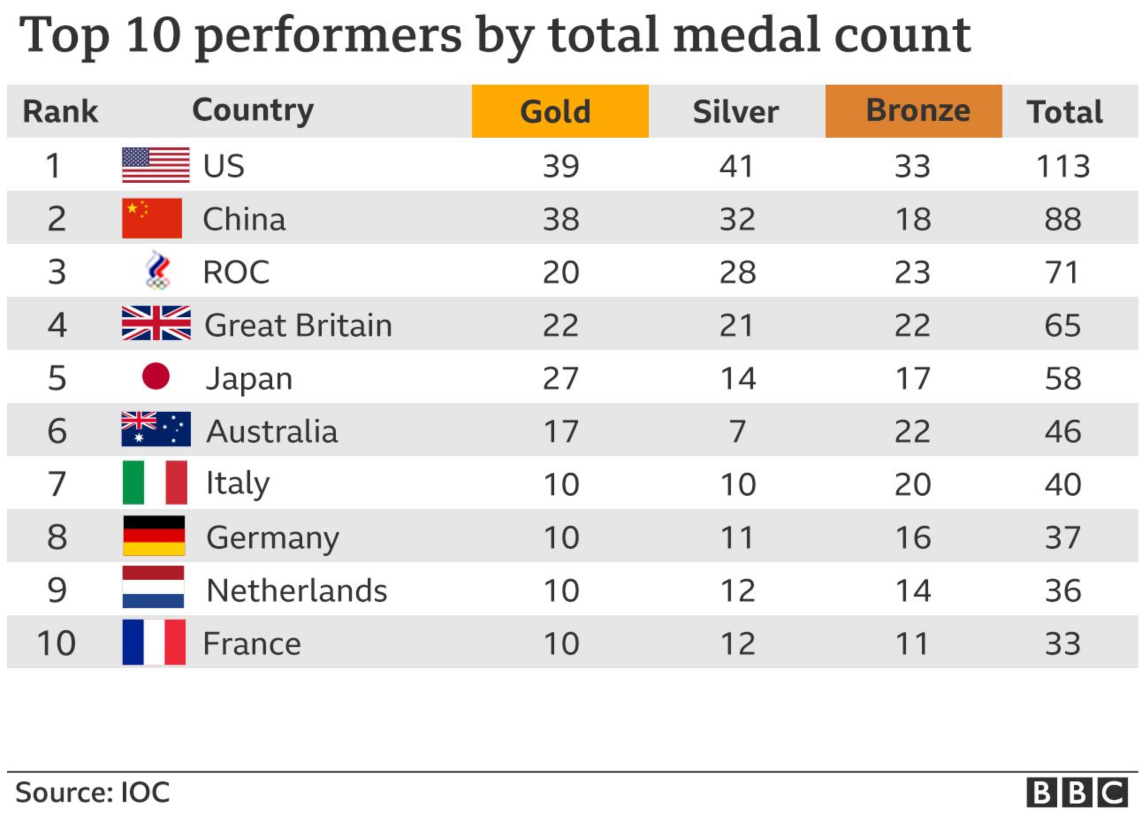
Source: IOC

BBC

Print all elements in a 2D Array

```
[0][0], [0][1], [0][2]
[1][0], [1][1], [1][2]
[2][0], [2][1], [2][2]
```

```
How do we print this?
```



Top 10 performers by total medal count

Rank	Country	Gold	Silver	Bronze	Total
1	US	39	41	33	113
2	China	38	32	18	88
3	ROC	20	28	23	71
4	Great Britain	22	21	22	65
5	Japan	27	14	17	58
6	Australia	17	7	22	46
7	Italy	10	10	20	40
8	Germany	10	11	16	37
9	Netherlands	10	12	14	36
10	France	10	12	11	33

Source: IOC

BBC

Print all elements in a 2D Array

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Process the ith row
   for (int j = 0; j < MEDALS; j++)
   {
      // Process the jth column in the ith row
      cout << setw(8) << counts[i][j];
   }
   // Start a new line at the end of the row
   cout << endl;
}
```

In order to print each element, we need two for loops:

- one to loop over all rows,
- and another to loop over all columns.

Computing Row and Column Totals

- One application is to compute the row and the column totals
    - What information do you need to compute the sum of a row?
    - What about a column?

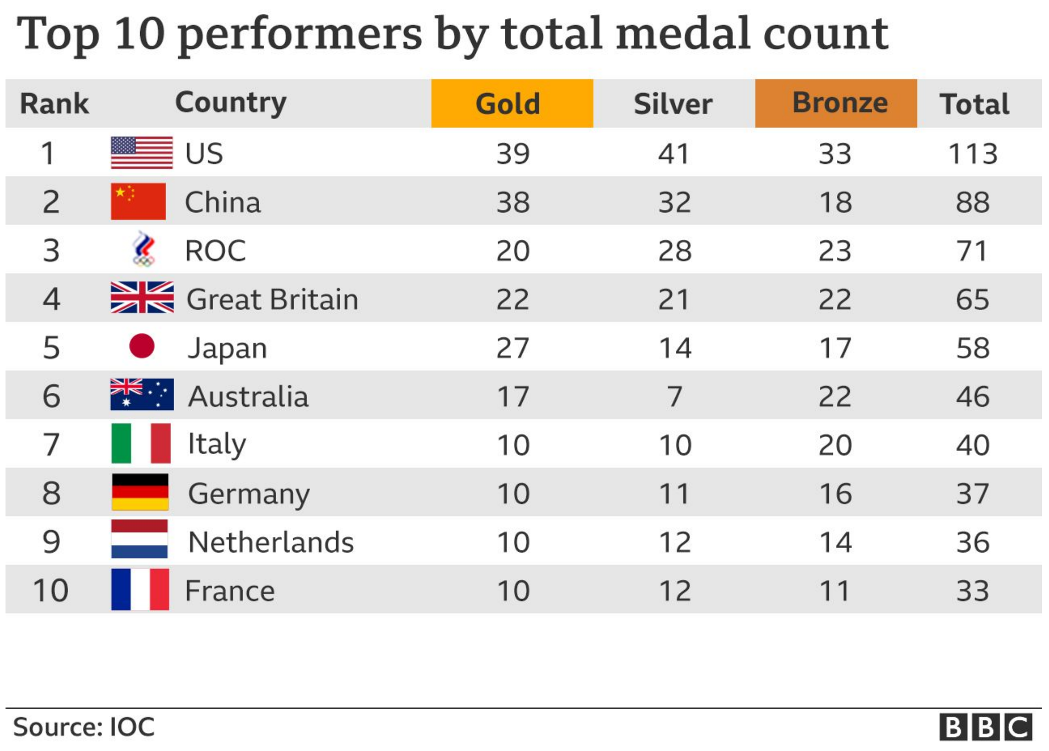
Computing Column Totals: Code Example

Column totals:

Let `j=1` be the silver column:

Let COUNTRIES=10

Let the 2d array be named as counts



Top 10 performers by total medal count

Rank	Country	Gold	Silver	Bronze	Total
1	US	39	41	33	113
2	China	38	32	18	88
3	ROC	20	28	23	71
4	Great Britain	22	21	22	65
5	Japan	27	14	17	58
6	Australia	17	7	22	46
7	Italy	10	10	20	40
8	Germany	10	11	16	37
9	Netherlands	10	12	14	36
10	France	10	12	11	33

Source: IOC

BBC

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given (Provided as second parameter)
  - 2nd dimension size IS given
- Example:

```
void displayMedalCount(const char count[][3], int countries)
{
   for (int index1=0; index1 < countries; index1++)
   {
      for (int index2=0; index2 < 3; index2++) {
         cout << p[index1][index2];

      }
      cout << endl;
    }

}
```

Two-Dimensional Array Parameters

- When passing a two-dimensional array to a function, you must specify the number of columns *as a constant* when you write the parameter type, so the compiler can pre-calculate the memory addresses of individual elements.
- This function computes the total of a given row.

```
int getRowTotal(int table[][3], int row)
{
   // Some code here
}

// calling this function (in main function)

int total = getRowTotal(table, 2);
```

Two-Dimensional Array Parameter Columns Hardwired

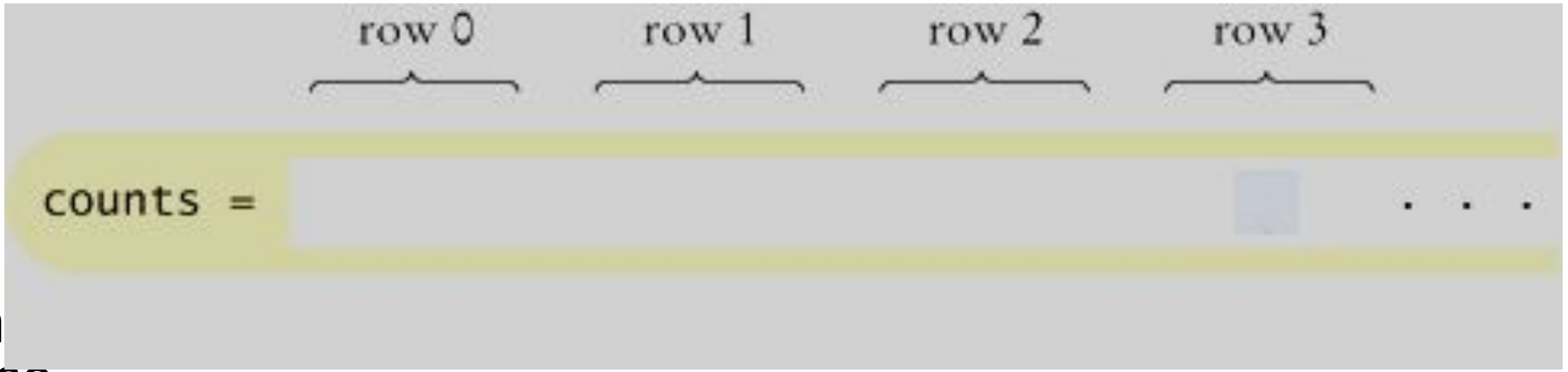
That function works for only arrays of 3 columns.

If you need to process an array with a different number of columns, like 4, you would have to write ***a different function*** that has 4 as the parameter.

Unfolding Two-Dimensional Arrays

Two-Dimensional Array Storage

What’s the reason
behind this?



Although the array appea
stored as a linear sequence.

`counts` is stored as a sequence of rows, each 3 long.

So where is **`counts[3][1]`**?

The offset (calculated by the compiler) from the start of the array is

`3 x` ***`number of columns`*** **`+ 1`**

Two-Dimensional Array Parameters: Rows

The **`row_total`** function did not need to know the number of rows of the array.

If the number of rows is required, pass it in:

```
int column_total(int table[][COLUMNS], int rows, int col)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][col];
   }
   return total;
}
```

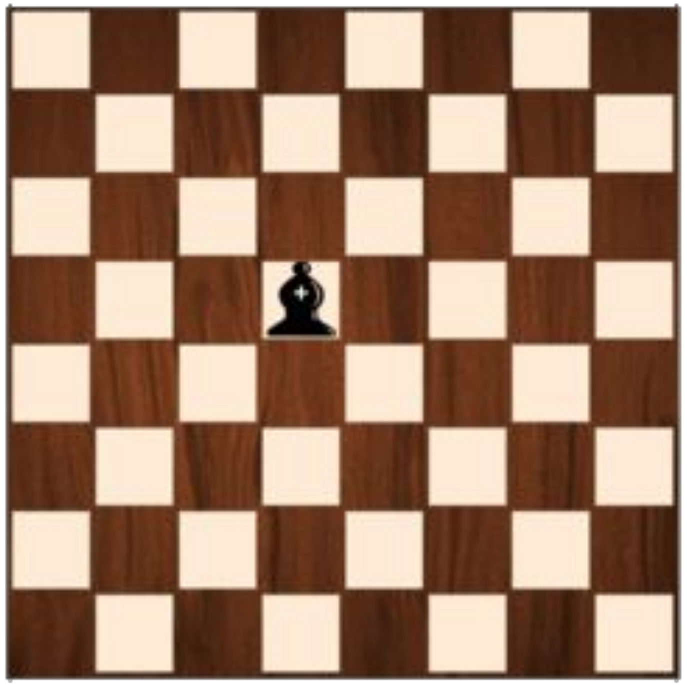
What are some of it’s drawbacks?

- The size of an array cannot be changed after it is created.
- You have to get the size right before you define an array
- The compiler needs to know the size in order to build the array, and functions need to be told number of elements in array, and possibly its capacity (and arrays can’t hold more than their initial capacity)

Practice It

- Check if the piece at row 4 and column 4 is a bishop

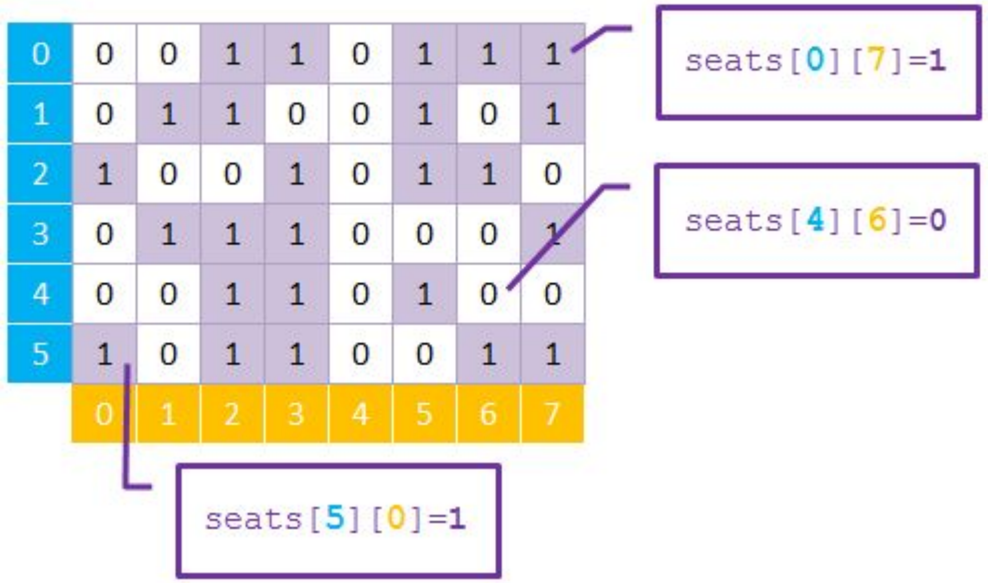
```
if () {
   cout << “It is a bishop”;
}


board[4][4] == “bishop”

board[3][3] == “bishop”

board[3][3] = “bishop”

board[4][4] = “bishop”
```



Practice It

- Show all empty seats in a theatre

```
for (i=0; i<rows; i++){
   for (j=0; j<cols; j++) {
      // check if the seat is
      // empty
   }
}
```



0	0	0	1	1	0	1	1	1
1	0	1	1	0	0	1	0	1
2	1	0	0	1	0	1	1	0
3	0	1	1	1	0	0	0	1
4	0	0	1	1	0	1	0	0
5	1	0	1	1	0	0	1	1
	0	1	2	3	4	5	6	7