

# 2D Arrays

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

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

# Defining 2D arrays

## Two-Dimensional Array Definition

Diagram illustrating the components of a 2D array definition:

- Element type**: `int`
- Rows**: `4`
- Columns**: `4`
- Name**: `data`

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

Optional list of initial values

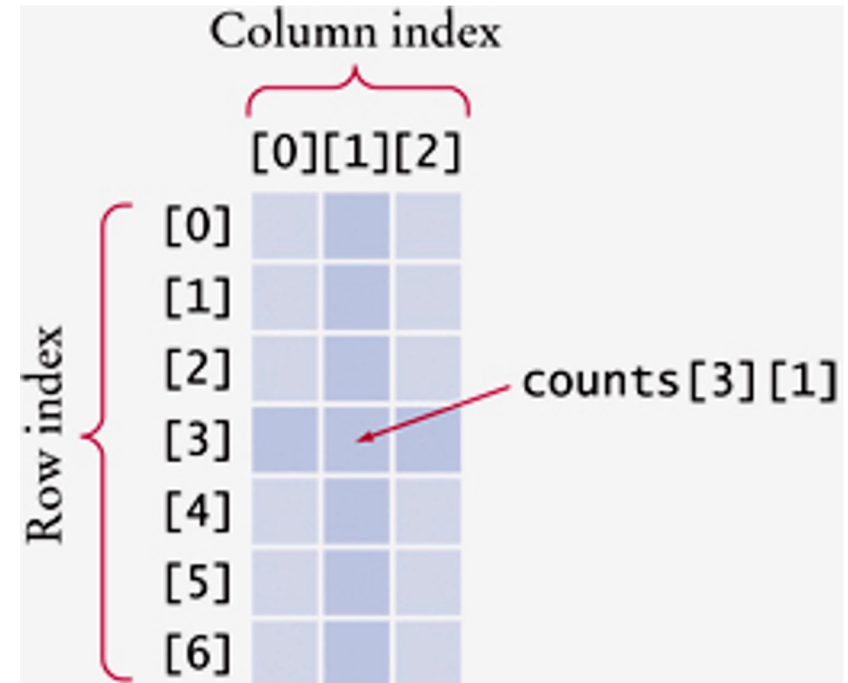
# Two-Dimensional Arrays – Accessing Elements

```
// copy to num what is currently  
// stored in the array at [3][1]
```

```
int num = counts[3][1];
```

```
// Then set that position in the  
array to 8
```

```
counts[3][1] = 8;
```



# Two-Dimensional Array Example

---

Consider the sales data from 2018 - 2021

|           | 2018 | 2019 | 2020  | 2021  |
|-----------|------|------|-------|-------|
| January   | 57   | 47.2 | 61    | 32    |
| February  | 68.1 | 34   | 46    | 54    |
| March     | 43   | 58   | 38.6  | 67.5  |
| April     | 82.4 | 43.2 | 57    | 29    |
| May       | 27   | 34.5 | 42.4  | 35    |
| June      | 34.8 | 18   | 31    | 80    |
| July      | 55   | 21   | 47.1  | 115   |
| August    | 61.6 | 71.7 | 67    | 98    |
| September | 107  | 91.4 | 87.3  | 100   |
| October   | 86   | 51   | 54.1  | 65    |
| November  | 142  | 112  | 173.5 | 210.5 |
| December  | 74   | 83   | 107.2 | 140   |

# Defining Two-Dimensional Arrays

---

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

```
const int MONTHS = 12;  
const int YEARS = 4;  
int sales[MONTHS][YEARS];
```

An array with 12 rows and 4 columns is suitable for storing our sales data.

- 2D arrays are built up as an array of 1D arrays!
- Each row is a 1D array

# Defining Two-Dimensional Arrays – Initializing

---

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.

```
double sales[MONTHS][YEARS] = { {57, 47.2, 61, 32},  
                                  {68.1, 34, 46, 54},  
                                  {43, 58, 38.6, 67.5},  
                                  {82.4, 43.2, 57, 29},  
                                  {27, 34.5, 42.4, 35},  
                                  {34.8, 18, 31, 80},  
                                  {55, 21, 47.1, 115},  
                                  {61.6, 71.7, 67, 98},  
                                  {107, 91.4, 87.3, 100},  
                                  {86, 51, 54.1, 65},  
                                  {142, 112, 173.5, 210.5},  
                                  {74, 83, 107.2, 140}};
```



# Print 1<sup>st</sup> row elements in a 2D Array

---

|           | 2018 | 2019 | 2020  | 2021  |
|-----------|------|------|-------|-------|
| January   | 57   | 47.2 | 61    | 32    |
| February  | 68.1 | 34   | 46    | 54    |
| March     | 43   | 58   | 38.6  | 67.5  |
| April     | 82.4 | 43.2 | 57    | 29    |
| May       | 27   | 34.5 | 42.4  | 35    |
| June      | 34.8 | 18   | 31    | 80    |
| July      | 55   | 21   | 47.1  | 115   |
| August    | 61.6 | 71.7 | 67    | 98    |
| September | 107  | 91.4 | 87.3  | 100   |
| October   | 86   | 51   | 54.1  | 65    |
| November  | 142  | 112  | 173.5 | 210.5 |
| December  | 74   | 83   | 107.2 | 140   |

```
[0][0], [0][1], [0][2], [0][3]
// Print the 1st row:
cout << setw(8) << sales[0][0];
cout << setw(8) << sales[0][1];
cout << setw(8) << sales[0][2];
cout << setw(8) << sales[0][3];
```

# Print 1<sup>st</sup> row elements in a 2D Array

---

|           | 2018 | 2019 | 2020  | 2021  |
|-----------|------|------|-------|-------|
| January   | 57   | 47.2 | 61    | 32    |
| February  | 68.1 | 34   | 46    | 54    |
| March     | 43   | 58   | 38.6  | 67.5  |
| April     | 82.4 | 43.2 | 57    | 29    |
| May       | 27   | 34.5 | 42.4  | 35    |
| June      | 34.8 | 18   | 31    | 80    |
| July      | 55   | 21   | 47.1  | 115   |
| August    | 61.6 | 71.7 | 67    | 98    |
| September | 107  | 91.4 | 87.3  | 100   |
| October   | 86   | 51   | 54.1  | 65    |
| November  | 142  | 112  | 173.5 | 210.5 |
| December  | 74   | 83   | 107.2 | 140   |

```
[0][0], [0][1], [0][2], [0][3]
// Process the 1st row:
for (int j = 0; j < YEARS; j++)
{
    cout << setw(8) << sales[0][j];
}
```

# Print all elements in a 2D Array

---

|           | 2018 | 2019 | 2020  | 2021  |
|-----------|------|------|-------|-------|
| January   | 57   | 47.2 | 61    | 32    |
| February  | 68.1 | 34   | 46    | 54    |
| March     | 43   | 58   | 38.6  | 67.5  |
| April     | 82.4 | 43.2 | 57    | 29    |
| May       | 27   | 34.5 | 42.4  | 35    |
| June      | 34.8 | 18   | 31    | 80    |
| July      | 55   | 21   | 47.1  | 115   |
| August    | 61.6 | 71.7 | 67    | 98    |
| September | 107  | 91.4 | 87.3  | 100   |
| October   | 86   | 51   | 54.1  | 65    |
| November  | 142  | 112  | 173.5 | 210.5 |
| December  | 74   | 83   | 107.2 | 140   |

```
[0][0], [0][1], [0][2], [0][3]
[1][0], [1][1], [1][2], [1][3]
[2][0], [2][1], [2][2], [2][3]
for (int j = 0; j < YEARS; j++)
{
    cout << setw(8) << sales[i][j];
}
```

# Print all elements in a 2D Array

---

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

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

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

# Computing Row and Column Totals

---

- We must be careful to get the right indices.
- For each row  $i$ , we must use the column indices:

$0, 1, \dots (\text{YEARS} - 1)$

# Computing Column Totals: Code Example

Column totals:

Let  $j$  be the 2021 column:

|           | 2018 | 2019 | 2020  | 2021  |
|-----------|------|------|-------|-------|
| January   | 57   | 47.2 | 61    | 32    |
| February  | 68.1 | 34   | 46    | 54    |
| March     | 43   | 58   | 38.6  | 67.5  |
| April     | 82.4 | 43.2 | 57    | 29    |
| May       | 27   | 34.5 | 42.4  | 35    |
| June      | 34.8 | 18   | 31    | 80    |
| July      | 55   | 21   | 47.1  | 115   |
| August    | 61.6 | 71.7 | 67    | 98    |
| September | 107  | 91.4 | 87.3  | 100   |
| October   | 86   | 51   | 54.1  | 65    |
| November  | 142  | 112  | 173.5 | 210.5 |
| December  | 74   | 83   | 107.2 | 140   |

```
int total = 0; //loop to sum down the rows
int j = 3;
for (int i = 0; i < MONTHS; i++)
{
    total = total + sales[i][j];
}
```

# Multidimensional Array Parameters

---

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

```
void DisplayPage(const char p[][100], int sizeDimension1)
{
    for (int index1=0; index1<sizeDimension1; index1++)
    {
        for (int index2=0; index2 < 100; 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.

```
const int COLUMNS = 3;
int row_total(int table[COLUMNS], int row)
{
    int total = 0;
    for (int j = 0; j < COLUMNS; j++)
    {
        total = total + table[row][j];
    }
    return total;
}
```



# Two-Dimensional Array Parameter Columns Hardwired

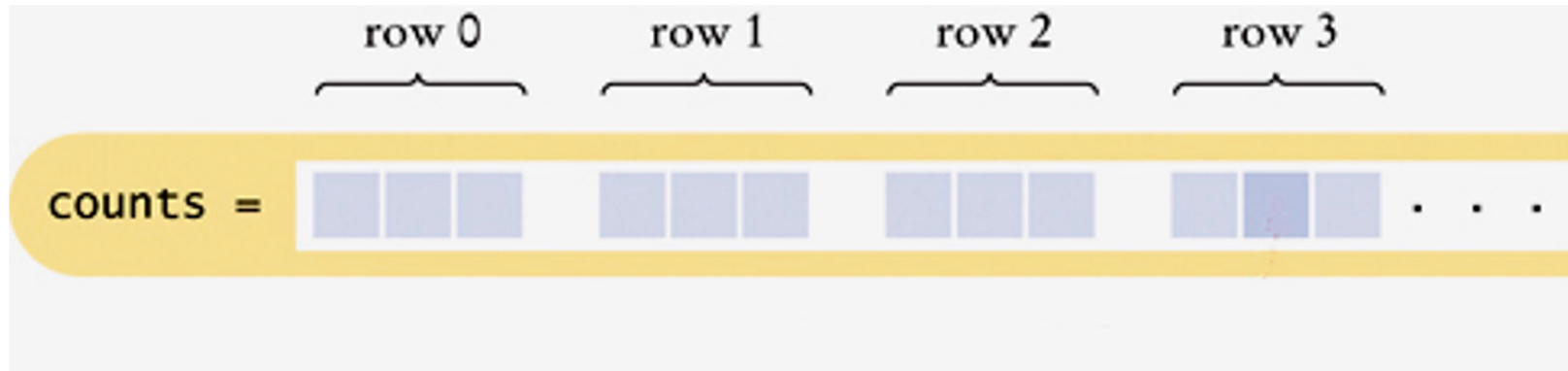
---

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.

# Two-Dimensional Array Storage

What's the reason behind this?



Although the array appears to be two-dimensional, the elements are still 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;
}
```

# Arrays -- fixed size can be a drawback

---

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)
- Later this week, we'll talk about vectors, which can have variable size and some other nice flexible features that arrays don't have.

# Multidimensional Arrays

---

- 2D array
- 3D array
- 4D array