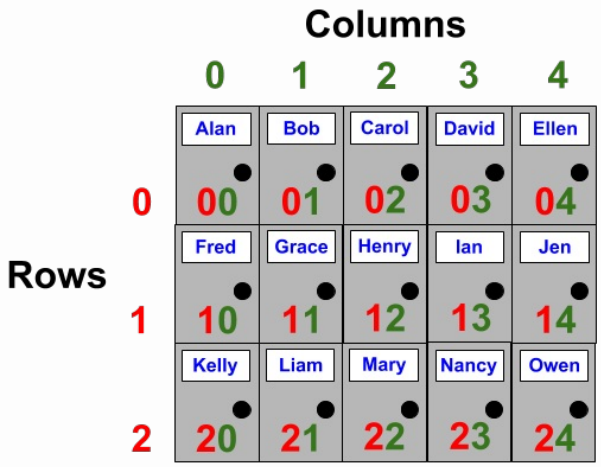
Learning Objectives: 2D Arrays

- **Create and initialize a 2D array**
- **Access and modify 2D array elements**
- **Iterate through 2D arrays using both a regular `for` loop and an *enhanced* `for` loop**
- **Determine 2D array output**

Creating a 2D Array

An Array Inside Another Array

An array inside another array is called a **2D array**. A 2D arrays is symbolic of a table where there are rows and columns. The first index number represents the **row** position and the second index number represents the **column** position. Together, the row and column indices enable elements to be stored at specific locations.



.guides/img/2DArray

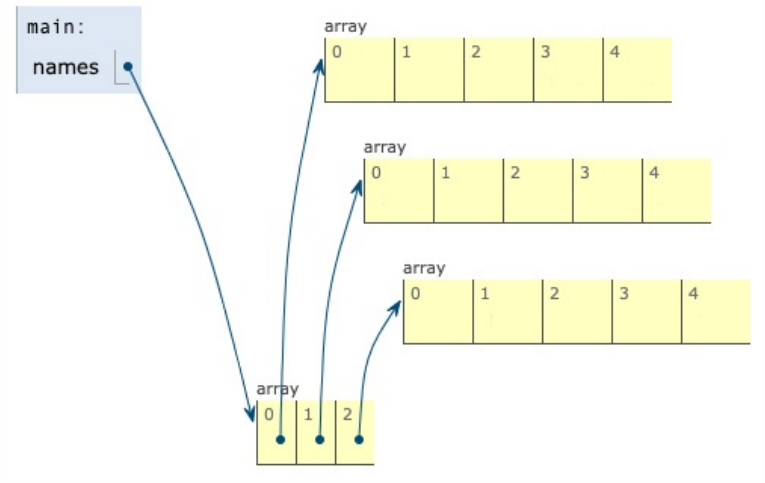
```
string names[3][5];
```

2D Array Syntax

- Array type followed by a name for the 2D array followed by **two** empty pairs of brackets `[][]`.
- The number of **rows** goes inside the **first** pair of brackets and the number of **columns** goes inside the **second** pair of brackets.

Why Array Inside Array?

The way 2D arrays store elements is a little unique. Rather than creating an actual table like shown above, each initial *row* of the 2D array actually refers to another *column* array. This is why a 2D array is considered to be *an array inside of another array*.



.guides/img/2DArrayReference

To determine the number of rows and columns in the 2D array, we can use the `sizeof()` operator like we did for arrays.

```
string names[3][5];

cout << sizeof(names) / sizeof(names[0]) << " rows" << endl;
cout << sizeof(names[0]) / sizeof(string) << " columns" << endl;
```

important

IMPORTANT

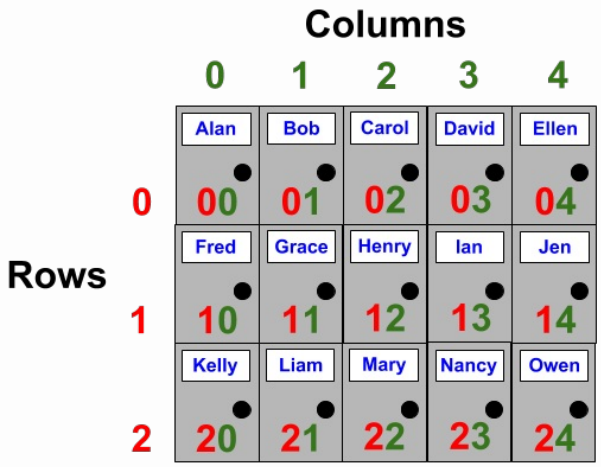
Note that when determining column length, you must refer to the 2D array's 0th row index. For example, `names[0]` doesn't just refer to the first element in the row, it also refers to the entire column of elements. **See image above.**

`sizeof(names[0])` calculates the size of the entire column while `sizeof(string)` calculates the size of one string. Thus, dividing the two, `sizeof(names[0]) / sizeof(string)`, will calculate how many string elements there are.

Accessing and Modifying a 2D Array

2D Array Access

To access and modify elements inside a 2D array, you need to specify the *row* and *column* indices at which the elements are located. For example `names[1][2]` refers to the element that's at row index 1 and column index 2.



.guides/img/2DArray

Below is a code block showcasing a 2D array that contains fifteen P.O. Box names from a postal office. Note that you can initialize the elements inside your 2D array just like how you initialize elements inside a regular array. Each *column* array is separated by curly braces `{}` as well as a comma `,`.

```
string names[ ][5] = { {"Alan", "Bob", "Carol", "David",
        "Ellen"},
                       {"Fred", "Grace", "Henry", "Ian", "Jen"},
                       {"Kelly", "Liam", "Mary", "Nancy",
        "Owen"} };


cout << names[1][2] << endl;
```

challenge

What happens if you:

- change `names[1][2]` from the original code to `names[2][1]`?
- change `names[1][2]` from the original code to `names[3][0]`?

---

important

IMPORTANT

Note that you must declare the number of elements within the column brackets. You can leave the row brackets empty, but you **cannot** leave the column brackets empty. Also, when you try to print an element that is **outside** of the row/column range, the system will either print random memory data or nothing at all.

2D Array Modification

To modify elements within a 2D array, simply access the element and assign another element to it.

```
string names[3][5] = { {"Alan", "Bob", "Carol", "David",
        "Ellen"},
                       {"Fred", "Grace", "Henry", "Ian", "Jen"},
                       {"Kelly", "Liam", "Mary", "Nancy",
        "Owen"} };

cout << names[1][2] << endl;

names[1][2] = "Harry";
cout << names[1][2] << endl;
```

challenge

What happens if you:

- change all `names[1][2]` within the code above to `names[0][0]`?
- change `"Harry"` within your current code to `"Amy"`?

Iterating a 2D Array

2D Array Iteration

To iterate through a 2D array, we can use two `for` loops, one **nested** inside another. The outer `for` loop is for the rows while the inner `for` is for the columns.

```
int digits[3][3] = { {1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9} };

int row = sizeof(digits) / sizeof(digits[0]); //number of rows
int col = sizeof(digits[0]) / sizeof(int); // number of columns

for (int i = 0; i < row; i++) {
  for (int j = 0; j < col; j++) {
    cout << digits[i][j] << endl;
  }
}
```

Code Visualizer

challenge

What happens if you:

- change all `sizeof(digits[0])` from the code above to `sizeof(digits[1])`?
- remove `<< endl` from the code?

Code Visualizer

Note that all of the columns' lengths are the same, there are 3 columns for each *row*. Therefore, it doesn't matter if we use `digits[0]`, `digits[1]`, or `digits[2]` when calculating the number of elements in each row and column. Also note that using `<< endl` prints the elements vertically while removing it prints the elements horizontally. To print the elements so that the columns stay together but the rows separate, we can try something like this:

```
int digits[3][3] = { {1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9} };

int row = sizeof(digits) / sizeof(digits[0]);
int col = sizeof(digits[0]) / sizeof(int);

for (int i = 0; i < row; i++) {
  for (int j = 0; j < col; j++) {
    if (j == col - 1) {
      cout << digits[i][j] << endl;
    }
    else {
      cout << digits[i][j] << " ";
    }
  }
}
```

Code Visualizer

The `if` conditional forces the elements to be printed with a newline every time the iterating variable reaches the end of the column index. Otherwise, the elements will be printed with a space instead.

2D Array with Enhanced For Loop

Like arrays and vectors, 2D arrays can also make use of the **enhanced** `for` loop.

```
int digits[3][3] = { {1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9} };

for (auto &i : digits) {
  for (int j : i) {
    if ((j == 3) | (j == 6) | (j == 9)) {
      cout << j << endl;
    }
    else {
      cout << j << " ";
    }
  }
}
```

Code Visualizer

You may have noticed that the outer loop contains `for (auto &i : digits)`. Unlike a regular array where we can access the first element by locating just one index, we need two indices in order to access elements within a 2D array. The `&i` creates a **reference** iterating variable that can refer to the 2D array. We type it as `auto` because doing so will cause the system to force the variable to match the 2D array type. In fact, we can always use `auto` to type variables to cause them to match the data that they refer to. For example, we can use `for (auto j : i)` for the inner loop instead of using `int`.

Also note that we cannot use an enhanced `for` loop to **manipulate** array indices. Our iterating variable goes through the 2D array and takes on each element value rather than each element index. This is why we have the conditional statement `if ((j == 3) | (j == 6) | (j == 9))` rather than `if (j == col - 1)`.