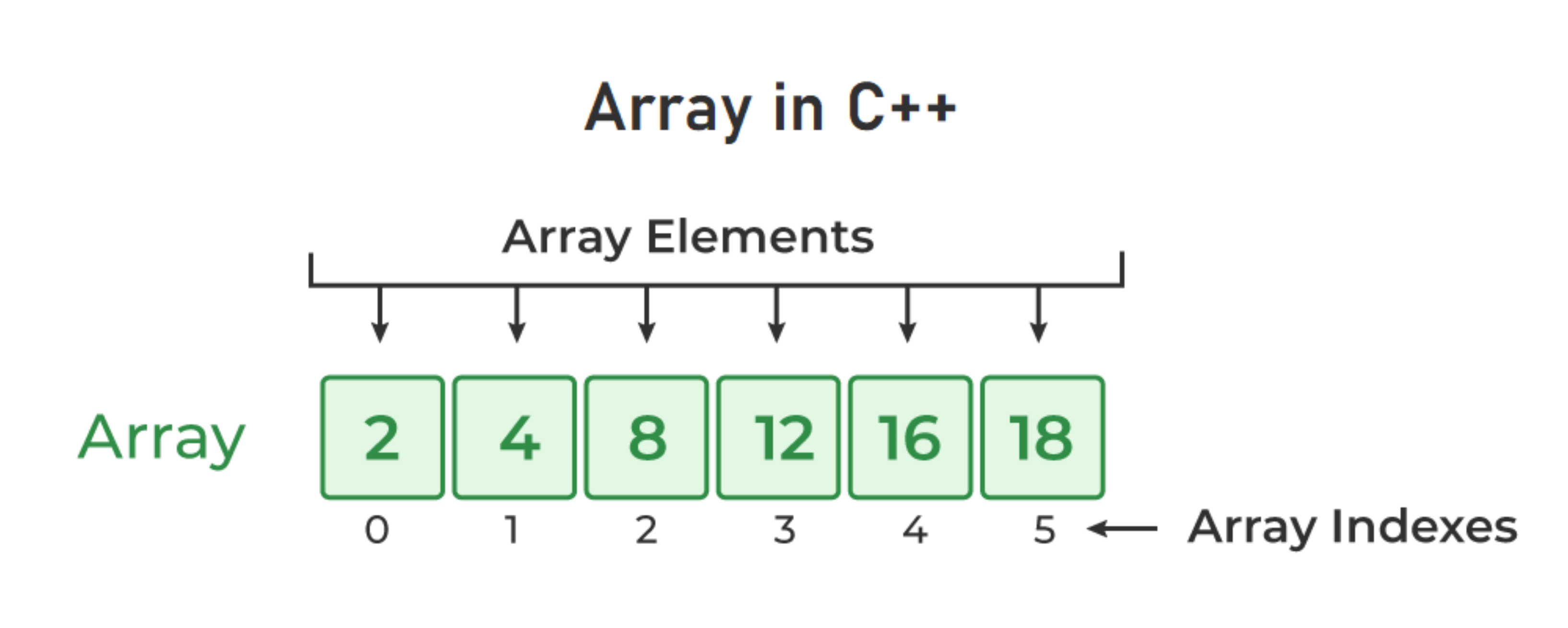
DISCUSSION SESSION
WEEK 5

C++ ARRAYS & VECTORS

An array is a container that is used to store multiple values in a single variable, instead of declaring separate variables for each value.



- Arrays store data (elements) of the **same type** (that is, an array cannot store both integers and doubles, and so on..). Elements are stored in a sequence.
- Arrays are indexed from 0…size - 1
- Each element in an array can be accessed using its index (inside square brackets **[ ]**)
- The size of an array must be **determined at compile-time,** so the compiler knows how much memory to allocate for the array elements. **The size of an array cannot be changed!**

Note: Arrays and Lists are two different things!

Array Declaration

Here are multiple methods to declare arrays:

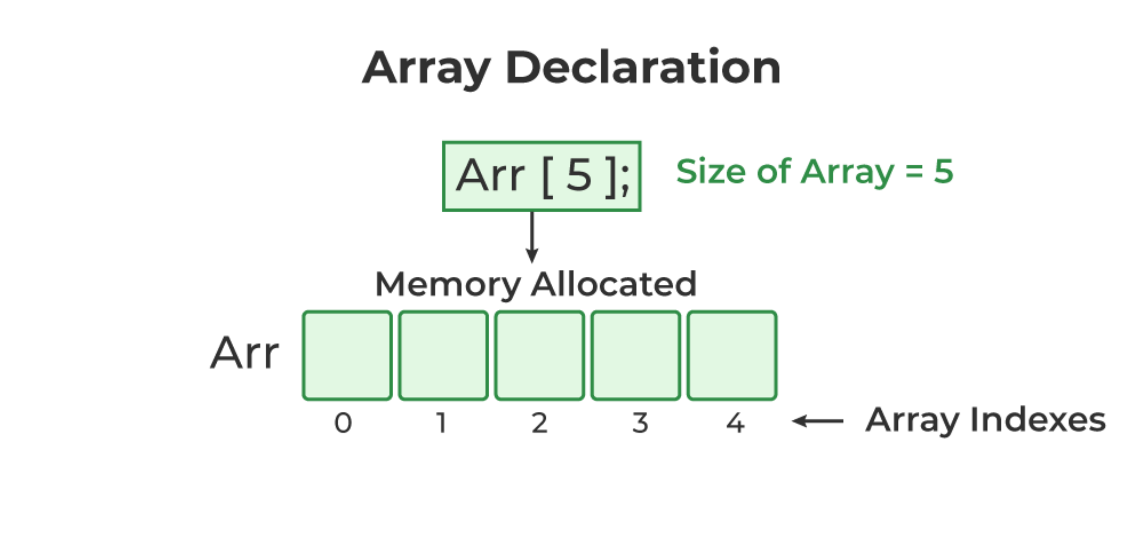
1. Fixed-size arrays: Most basic way to declare an array

- By specifying size directly

```
int myArray[5];
```

- By user-specified size

```
int size = 5;
int myArray[size];
```

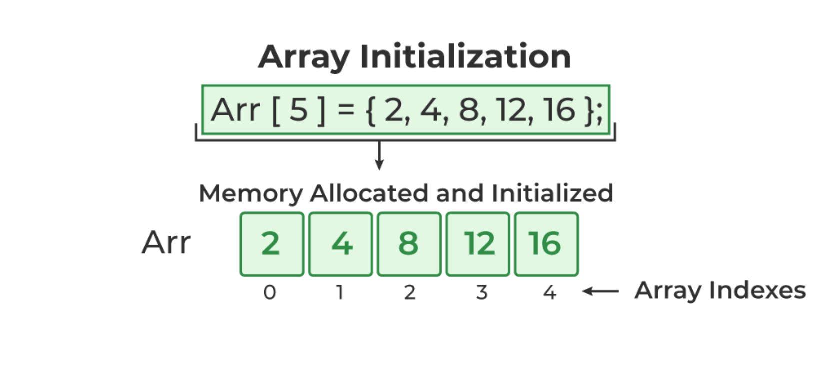


Array Declaration

2. Initialization at declaration: Arrays can be initialized at the time of declaration

- Initializing a size-defined array

```
int myArray[5] = {2, 4, 8, 12, 16};
```



Array Initialization

```
Arr [ 5 ] = { 2, 4, 8, 12, 16 };
```

```
double myArray[4] = {1.0, 2.0};
```

- Automatic size determination

```
int myArray[ ] = {4, 5, 3, 6, 9, 2};
```

Array Accessing/Indexing

An element of an array can be accessed using an index, which is a number that represents the position of the item in the array.

```
int arr[4] = {3, 6, 2, 7};
std::cout << arr[0];    std::cout << arr[1];
std::cout << arr[2];    std::cout << arr[3];
```

If this array had a larger size (say 300), how would we print out all elements of the array? Definitely not 300 print statements! What’s the alternative?

Array Accessing/Indexing

A **loop** can also be used to iterate through the items in an array!

```
int arr[4] = {3, 6, 2, 7};
for(int i = 0; i < 4; i++) {
    std::cout << arr[i] << std::endl;
}
```

Array Element Modification

```
std::string cities[3];
cities[0] = "Boston";
cities[1] = "San Francisco";
cities[2] = "Salt Lake City";
```

Change San Francisco to Phoenix?

```
cities[1] = "Phoenix";
```
✅

Passing Array into Function

- Arrays are automatically passed into functions by reference. So, any changes made to the array within the function will be reflected in the original array.
- In the function parameters, it’s best to use empty brackets and pass in the array size separately as another parameter.

```
void func(int myArr[ ], int arrSize) {
        // some code..
}
void func(int myArr[5], int arrSize) {
        // some code..
}
```

Passing Array into Function

When providing the argument in the function call, use the array name:

```
int main( ) {
    int myArr[7];
    func(myArr, 7);
}
```

Exercise (5 minutes)

In main, declare **myArr** (an integer array of size 5).

Pass the array into a function. Within that function:

- Use a loop to set each element of *myArr* to the value of its index multiplied by 2.
- Print each element of *myArr* separated by a whitespace (using another loop!)
- Now, print out all elements of *myArr* in reverse order!

Expected Output:

```
0 2 4 6 8
8 6 4 2 0
```

VECTORS

Similar to arrays, vectors are a sequence of elements of a single type. However, unlike arrays, **vectors can change in size.** This is because vectors are implemented as dynamic arrays, which means that they can grow and shrink as needed. This makes vectors a very flexible and powerful data structure.

The C++ vector class is very *nice* because it provides us with many methods/functions that we can call on our vector objects/instances.

Vector Declaration

- First, you **must** #include <vector> header in your code

```
        std::vector<int> myVect; // creates an empty vector
```

Note: You absolutely cannot index an empty vector.

```
// Create a vector to store 20 elements
        std::vector<int> myVec(20);
```

Note: Even though this vector is initially created to store 20 integers, you can add more numbers to the vector.

Vector Declaration

- First, you **must** #include <vector> header in your code

```
std::vector<int> myVect = {2, 9, 3, 4, 7, 4};
```

Note: This initialization at declaration method is called an **initializer list** and only works with c++17 and later. So, if you must use it, be sure to pass (at least) a c++17 flag to your compilation process.

Some Vector Class Methods

```
std::vector<double> myVec = {2.5, 3.7, 12.6, 8.2};

myVec.push_back(10.1); // appends 10.1 to the end of myVec
myVec.pop_back( ); // deletes the last element of myVec
std::cout << myVec.size( ); // prints out size of myVec
bool isEmpty = myVec.empty( );
std::cout << myVec.front( ) << " " << myVec.back( ) << std::endl;
std::cout << myVec.at(2) << " " << myVec[2] << std::endl;
myVec.clear( );
```

Passing Vector into Function

```
void func(std::vector<int> myVec) {
    // some code..
}
void func(std::vector<int>& myVec) {
    // some code..
}
```

When providing the argument in the function call, use the vector name:

```
func(myVec);
```

Exercise (5 minutes)

In main, declare **names** (a vector of strings). Pass the vector into a void function by reference. Within that function:

- Add the following names to the vector one at a time: *John, Sarah, Jasmine, Damian, Mai, Ciara*
- Remove the last name from the vector

Back in main:

- Using a loop, print out all names in the vector, one on each line
- Print out the size of the vector
- Delete all contents of the vector

2-DIMENSIONAL ARRAYS & VECTORS

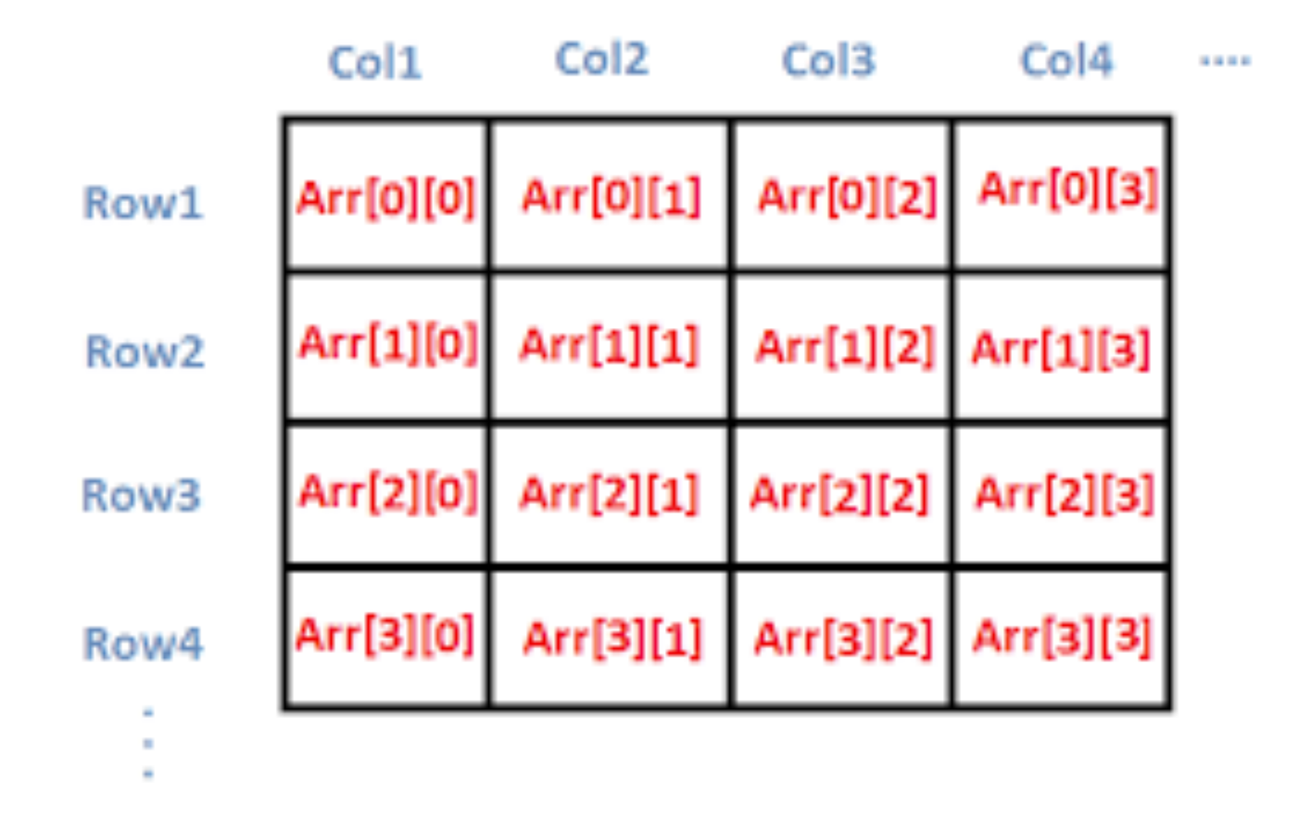
A 2D array is *just* an array of arrays. Similarly, a 2D vector is a vector of vectors. We use multidimensional arrays to store a grid of items, like a chessboard or a spreadsheet.

The general rule of thumb is to use a 2D array if you know the size of the grid at compile time, and to use a 2D vector if you don't.

This is because 2D arrays are more efficient, but 2D vectors are more flexible because they can grow and shrink as needed.

2-DIMENSIONAL ARRAY & VECTOR DECLARATION

```
int myArr[4][4]; // Creates a 4x4 array
```



	Col1	Col2	Col3	Col4	····
Row1	Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[0][3]	
Row2	Arr[1][0]	Arr[1][1]	Arr[1][2]	Arr[1][3]	
Row3	Arr[2][0]	Arr[2][1]	Arr[2][2]	Arr[2][3]	
Row4	Arr[3][0]	Arr[3][1]	Arr[3][2]	Arr[3][3]	
⋮					

2-DIMENSIONAL ARRAY & VECTOR DECLARATION

```
int rows = 3, cols = 4;
int myArr[rows][cols] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
};
std::vector<std::vector<int>> myVec = {
        {1, 2, 3}, {4, 5, 6}, {7, 8, 9}
};
```

```
int rows = 3, cols = 4;
int myArr[rows][cols];
for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
                std::cin >> myArr[i][j];
        }
}
```

```
std::vector<std::vector<int>> myVec = {
        {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}
};
    for(int i = 0; i < ______; i++) {
        for(int j = 0; j < ______; j++) {
            std::cout << myArr[i][j];
        }
        std::cout << std::endl;
    }
```

```
std::vector<std::vector<int>> myVec = {
        {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}
};
    for(int i = 0; i < myVec.size( ); i++) {
        for(int j = 0; j < myVec[0].size( ); j++) {
            std::cout << myVec[i][j];
            std::cout << myVec.at(i).at(j);
        }
        std::cout << std::endl;
    }
```

2-D ARRAY & VECTOR AS FUNCTION PARAMETERS

```
void func(char myArr[ ][5], int rows, int cols) {
    // some code..
}
```

- The second square brackets of array declaration parameter cannot be empty

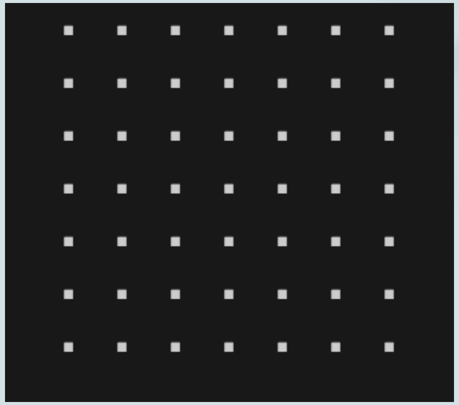
```
void func(std::vector<std::vector<double>>& myVec) {
    // some code..
}
```

Exercise (15 minutes)

1. Create a void print function that takes in a char 2D array and prints out the grid. Use function as needed!

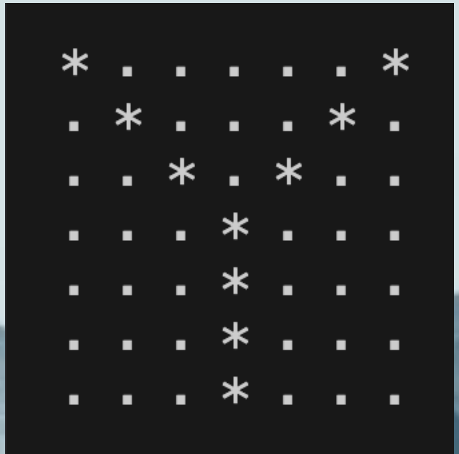
2. In main, declare **myGrid** (a char 2D array of size 7x7).

***** Fill the grid with dots



```
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
```

***** Make a Y-shaped path with asterisks



```
* . . . . . *
. * . . . * .
. . * . * . .
. . . * . . .
. . . * . . .
. . . * . . .
. . . * . . .
```

But first, we are going to make a shape together!