```
//============================
// Section 7 - Arrays
//============================

// What is an array?
Compound data type or data structure
Collection of elements
All elements are of the same type
Each can be accessed directly

// Why do we use them?
Easy to group like data - for example test scores

// Characteristics
- Fixed size
- Elements are all the same type
- Stored contiguously in memory
- Individual elements accessed by index
- First element is at 0
- Last element is at index size - 1
- No bounds checking
- Always must be initialized
- Very efficient
- Iteration is pretty easy

// Declaring and initializing arrays
ElementType ArrayName[constant number of elements];

int testSCores[5]{1,2,3,4,5};
int highScore[100]{2,3}; // Will initialize the first two elements, the rest will be 0's
const double daysInYear{365}; // For initialing below
double hiTemps[daysInYear]{0}; // Initialize all to 0
double hiTemps[daysInYear]{}; // Will also initialize all to 0
int anotherArray[]{0,1,2,3,4,5,6}; // Compiler will determine size based on the number
of elements

// Accessing and modifying Array elements
arrayName[elementIndex]
int testScores[3]; // This will show at location 2 in the array
testScores[3] = 20; // Assign a value to an element in the array

// How does it work?
The name of the array represents the location of the first element in the array
The index represents the offset from the beginning location
C++ will perform a calculation to find the correct element
There's no bounds checking on an array, so you can pull data from a location
that doesn't exist.
printing out just the array name gives the hexidecimal memloc

// Multidimensional arrays
dataType arrayName[x][y];
int movieRating[3][4];

// Example grid multidimensional array
    0   1   2   3   4
0   y   n   y   y   y
1   f   r   r   w   b
2   h   r   e   e   l
3   n   k   z   a   l

Access element e would be at [2][2]

// Initializing
int movieRating[3][4]{
    {0,1,2,3},
    {4,5,6,7},
    {3.5.6.7}
}

```

```
// Declaring and initializing vectors
Use when you don't know how many elements you will have ahead of time

Container in the Standard Template Library
An array that can grow and shrink at execution time
Provides similar syntax as arrays
Very efficient
Does bounds checking
Has built in functions like sort, reverse, find, etc.

// Declaring
#include <vector>

vector<char> vowels;
vector<int> testScores;
vector<char> vowels(5); // Constructor initialization tells compiler we want 5 characters
vector<int> testScores(10); // Again, we want 10 integers

// Two dimensional vectors
vector<vector<int>> movieRatings{
    {1,2,3,4},
    {2,3,4,5},
    {3,4,5,6}
};

// Access two dimensional vectors
movieRatings[2][1];
movieRatings.at(2).at(1);

// Can also use initializer lists
vector<char> vowels{'a' 'e', 'i', 'o','u'};
vector<double> hiTemperatures(365, 80.7); //First value is how many elements, second
what to initialize to

// Characteristics
Dynamic size
Elements are all the same type
Stored contiguously in memory
Can be accessed by position or index
First element at index 0
Last element at index size - 1
[] - No checking to see if you are out of bounds
Has other methods that do bounds checking
Elements initialized to zero by default
Very efficient
Loop to iterate through them


// Accessing elements
testScores[1];
testScores.at(1); // Object oriented way - you can access and update elements this way
testScores.push_back(element); // add to the back of the vector

// What happens if you go out of bounds?
Arrays never do bounds checking
Many methods in vectors do
An exception is thrown and an error message is generated we can read

// Common methods
.at()
.size()
.push_back()
```

```cpp
//============================
// Section 7 Challenge
//============================

#include <iostream>
#include <vector>

int main(){

// Declare 2 empty vectors of integers
    std::vector<int> firstVector{};
    std::vector<int> secondVector{};

// Add 10 and 20 to the first one with push_back()
    firstVector.push_back(10);
    firstVector.push_back(20);

// Display the elements using at() and the size using size()
    std::cout << "First Vector: " << firstVector.at(0) << std::endl;
    std::cout << "First Vector: " << firstVector.at(1) << std::endl;
    std::cout << "First vector
// add 100 and 200 to the second using push_back()
    secondVector.push_back(100);
    secondVector.push_back(200);

// Display the elements using at() and the size using size()
    std::cout << "Second Vector: " << secondVector.at(0) << std::endl;
    std::cout << "Second Vector: " << secondVector.at(1) << std::endl;

// Declare an empty 2d vector
    std::vector<std::vector<int>> twoDimensionalVector{};

// add the first vector to it, then the second one
    twoDimensionalVector.push_back(firstVector);
    twoDimensionalVector.push_back(secondVector);

// display the elements using the at() method
    std::cout << "2d Vector at position 0 0: " << twoDimensionalVector.at(0).at(0) <<
    std::endl;
    std::cout << "2d Vector at position 0 1: " << twoDimensionalVector.at(0).at(1) <<
    std::endl;
    std::cout << "2d Vector at position 1 0: " << twoDimensionalVector.at(1).at(0) <<
    std::endl;
    std::cout << "2d Vector at position 1 1: " << twoDimensionalVector.at(1).at(1) <<
    std::endl;

// Change vector1.at(0) to 1000;
    firstVector.at(0) = 1000;

// Display the elements in the 2d vector again
    std::cout << "2d Vector at position 0 0: " << twoDimensionalVector.at(0).at(0) <<
    std::endl;
    std::cout << "2d Vector at position 0 1: " << twoDimensionalVector.at(0).at(1) <<
    std::endl;
    std::cout << "2d Vector at position 1 0: " << twoDimensionalVector.at(1).at(0) <<
    std::endl;
    std::cout << "2d Vector at position 1 1: " << twoDimensionalVector.at(1).at(1) <<
    std::endl;


// Display the elements in vector 1
    std::cout << "First Vector: " << firstVector.at(0) << std::endl;

// What happened?
// No change in the 2d vector. It must be getting its own copy of the data.
```