You should come away with understanding:

- how arrays are declared
- how to initialize arrays
- how to use for loops to traverse arrays
- how to pass arrays as parameters to functions
- how to declare and use 2D arrays
- how 2D arrays are stored in memory
- how to pass 2D arrays as parameters to functions
- how to process 2D arrays

1) List as many ways as you can find that C++ arrays are similar to python lists.

- They both serve as containers for a collection of other data variables.
- They are both ordered (first item, second item, ... last item)
- They can both access items by index number which starts at 0 and goes to n-1.
- They can both be initialized with values at the time they are created.
- They both allow "nesting" to facilitate the creation of multi-dimensional data structures. That is, we can make an array of other arrays (or a list of other lists).
- They can both be passed to functions as parameters.

2) List as many ways as you can find that C++ arrays are different from python lists.

- C++ arrays are not primitive data types nor are they objects. It is best to think about them as a collection of other datatypes that are stored consecutively in computer memory. C++ arrays do not have methods associated with them. All functions that process C++ arrays are "external" to the array. Python lists are objects. They have data attributes (values) and methods (routines that operate on lists like append, pop, insert, etc). This same functionality can be duplicated in C++, but the programmer needs to write external functions to perform these tasks on an array.
- Since a C++ array is not an object, it does not keep track of its own size. You need an auxiliary variable to keep track of the number of items in the C++ array. Since python lists are objects, they keep track of their length internally which can be accessed via `len(myList)`.
- C++ arrays must have homogenous data -- all the items in an array must have the same data type whereas with python, you can mix different data types in the same list. The reason for this restriction in C++ is that since a C++ array is a collection of consecutive memory locations, each memory location for each item must be the same size (same number of bytes of memory). That way, when we index a particular location in an array, we know how far to jump ahead from the start of the array to access that particular item in memory.
- C++ arrays must be declared with size and data type. A python list is "created" as either an empty list or an initialized list.
- C++ arrays have fixed length which is established when you declare the array. You cannot make the array bigger once it is created. If you need more space in your array,

you would have to create a second array that is larger and copy the values from the first array into the second. Python lists, of course, can be augmented with additional data.

- You cannot create an item in a python list at index 0 and at index 2 without also creating an item at index 1 (there are no "holes" in a python list). In a C++ array, once you create the array, the array locations are all there. You can then "initialize" the values to anything that you like in any order that you like.
- In python for loops, you can iterate over the indices in a python list or the values in a python list. In C++ arrays, you can only iterate over the indices. Later, when we develop STL (standard template libraries) we will see C++ vectors which are objects and function more like python lists.
- Later, when we learn more about pointers and addresses, we will see that an array variable acts like a memory address and can be used interchangeably with addresses in some computing situations. Python lists are not memory addresses since they are objects and they also run inside the virtual machine that is a python interpreter that abstracts away the notion of an address from the python programmer.

**1) Write a declaration below that creates a two-dimensional array of floats with 5 columns and 3 rows.**

```
float     array[3][5];
// note rows comes first, then columns
```

**2) Explain why it is necessary when passing a two-dimensional array as a parameter to a function that you must specify the dimension of the columns (the number of columns) but it is optional to specify the dimension of the rows.**

You must specify the column dimensions of an array that you pass because the compiler must know how many columns a 2D array has in order to properly figure out the address of each spot in the array. (see next question response)

In the above program, we have defined a function named `display()`. The function takes a two dimensional array, `int n[][2]` as its argument and prints the elements of the array.

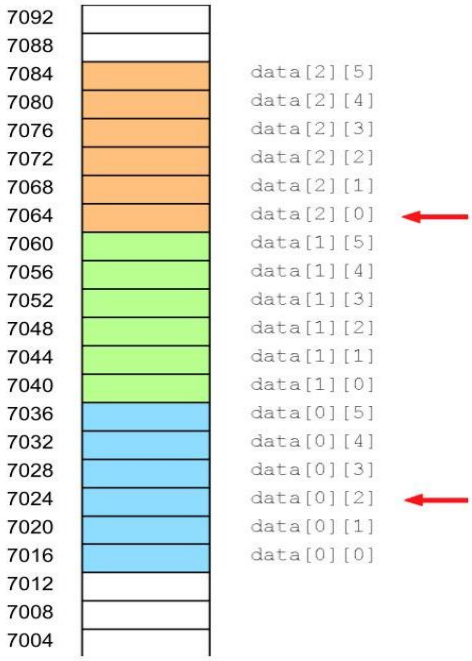
While calling the function, we only pass the name of the two dimensional array as the function argument `display(num)`.

> **Note**: It is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified. This is why we have used `int n[][2]`.

We can also pass arrays with more than 2 dimensions as a function argument.

**3) Below is a declaration for a two-dimensional array of integers. Draw a map of how this array is stored in memory. How many bytes apart are `data[0][2]` and `data[2][0]`? You do not need to submit your drawing but you need to explain the calculations you did to arrive at your byte distance.**

```
int  data [3][6];
```




We can see the memory map above. I have made up the addresses, but what is important is how the array is stored in memory in row-by-row fashion.

The address of any particular element at data[i][j] can be computed as

```
address of data[i][j] =
(i * COLS + j) * sizeof(int) + base address of array
```

Notice that we need the number of columns (COLS) for this calculation but not the number of rows. This is why when we pass in the array we must specify the number of columns but not the number of rows. COLS is needed for this array addressing calculation but ROWS is not.

address of data[2][0] – address of data[0][2] = 7064 – 7024 = 40 bytes of distance

We can also see that there are 10 steps in the model from data[2][0] to data[0][2] and each step is 4 bytes = 10*4 = 40 bytes.

The address of any particular element at data[i] can be computed as

Address of item at index i = base address + (i * size of the type of things in the array)
Address of item at index 3 = 7000 + ( 3 * 4) = 7012

A 2D array is stored in memory as a single contiguous block in row-major order.

**4) Below is a function called miniMax. Write the code for this function. I suggest you test your function in a program and then copy/paste it here.**

```
/*
This function finds and returns the smallest row-maximum value
in a 2D array.
PARAMETERS:
data: a 2D array of integers with ROWS rows and COLS columns.
     ROWS and COLS are global const integers defined elsewhere
RETURN VALUE:
This function computes the maximum integer in each row.  It then
returns the minimum value among all these row-maximums.
*/

int  miniMax ( int data[ROWS][COLS] )
{
   int globalMin;
   int rowMax;

   for ( int r=0; r < ROWS; r++ )
   {
       // initialize to first value in this row
       rowMax = data[r][0];
```

```
        for ( int c=1; c<COLS; c++ )
        {
           // found larger value in this row
           if ( data[r][c] > rowMax )
              rowMax = data[r][c];
        }
        // record globalMin if we find a smaller rowMax
        if ( r == 0 || rowMax < globalMin )
           globalMin = rowMax;
    }
    return globalMin;
}
```

1. When we call a function by passing an array as the argument, only the name of the array is used.

```
display(marks);
```

Here, the argument `marks` represent the memory address of the first element of array `marks[5]`.

2. However, notice the parameter of the `display()` function.

```
void display(int m[])
```

Here, we are just expecting an array of integers. C++ handles passing an array to a function in this way to save memory and time.

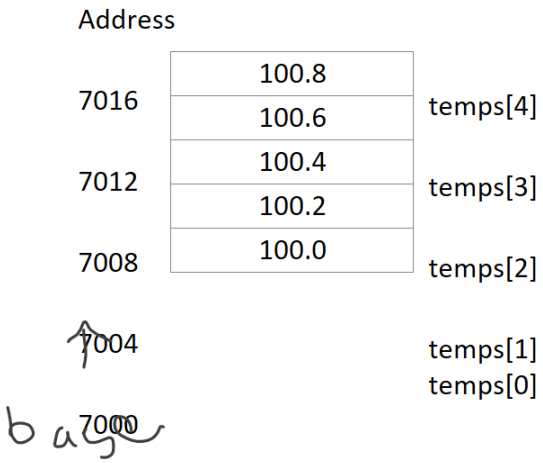
- Kind of like python lists
  - Structured collection of elements
  - Use indices to access specific pieces
    - arr[0];
  - We can go through each thing in them using a for loop
    - We have to use indices to access items in C++ arrays!
- But also not!
  - Can only hold one type of thing, we have to declare the array with the type of thing it can hold
    - int arr[10];
    - float arr2[10];
  - Need to specify maximum capacity of array when we declare it!
  - Not objects! Have NO methods

## Array Example

Monday, February 17, 2025 8:54 AM

```
float temps[5]; // Allocates memory
int m;

for (m = 0; m < 5; m++)
{
   temps[m] = 100.0 + m * 0.2 ;
}
```




Address of item at index i = base address + (i * size of the type of things in the array)

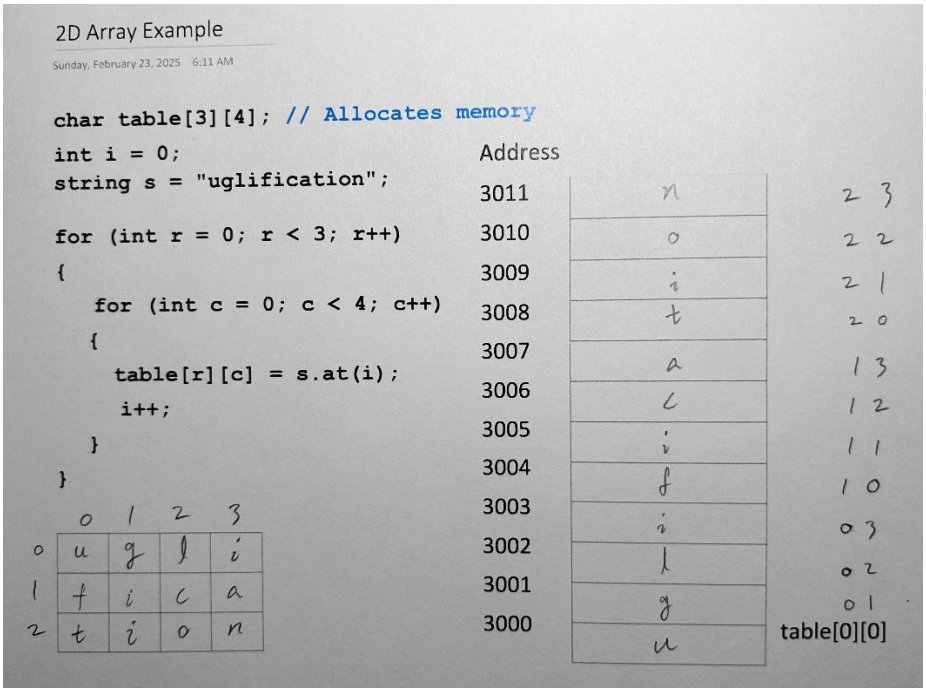
Address of item at index 3 = 7000 + ( 3 * 4) = 7012

Time Complexity

- A measure of how much the time it takes to execute an algorithm grows as the size of the input increases
- How much longer does it take if we have an array with 10 items vs. 5 items, for example?
- Big-Oh
    - In the worst-case scenario, what's the main contributing factor to this growth in time?
    - In terms of input size (n)
    - With arrays, n is the number of items in the array
- O(1)
    - Constant amount of time
    - x< y;
- O(n)
    - Linear time
    - We can multiply some constant by n to get an upper bound on the amount of time it takes to do this operation
        - More in 271!
    - The most important contributing factor is the size of the input
    - n-1 comparisons to check if an array is sorted
- O(n^2)

2D Arrays

- Collection of items
- We can get at them by index
- All those items have to be the same type
- Items are organized in rows and columns
- **Row-major**
    - Rows come before columns
    - float some2DArr[NUM1][NUM2];
- #rows #cols
-
    - some2DArr[0][1] = 3.14;
      row col




To find address of the item in table[x][y]:

- We need to know indices we are looking for (x and y)
  - Say x = 1, y = 3
- Base address
  - 3000
- Type of data (size of that type)
  - Char, size 1 byte

Address of table[x][y]
= base address + (x * num cols * size of thing in array) + y * size of thing in array

```
(i * COLS + j) * sizeof(int) + base address of array
```

Address of table[1][3] = 3000 + (1 * 4 * 1) + (3 * 1)
Address of table[2][0] = 3000 + (2 * 4 * 1) + (0 * 1)

Write a boolean function called isSorted( ) which takes an array of integers and its size as parameters. The function returns false if any larger value in the array comes before a smaller value later in the array. Otherwise it returns true. In other words, isSorted() returns true if the array is sorted in ascending order. You can assume the array has at least one value (it is not empty with size 0 ).

```
bool isSorted ( int array[ ], int size )
{
```

What does the sizeof() function do? Be specific.

In C++, how can we specify that an array passed as a function parameter should not be modified by the function?

**1) [15pts]** Consider the 2D array of integers passed as a parameter to the function below. The row and column dimensions of the array are a global constant M. The function is to search for a 2x2 square within the 2D array where the sum of the four integers is exactly 10. If such a 2x2 square exists, the function returns true. Otherwise it returns false. Write the code for this function.

example:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 10 | 7 | 3 | 4 ... |
| 12 | **5** | **2** | 6 | 12 | 19 | 0 ... |
| 0 | **0** | **3** | 12 | 15 | -5 | 10 ... |
| 22 | 11 | -5 | 10 | 10 | 0 | 2 ... |
| ... | | | | | | |

returns true because there is a 2x2 square of consecutive values that sums to 10 exactly.

```
const M = 20;
bool  isSquare10    ( int array[M][M] ) {
```

Explain why it is necessary when passing a two-dimensional array as a parameter to a function to specify the number of columns but it is optional to specify the number of rows.