Array (std::array first)

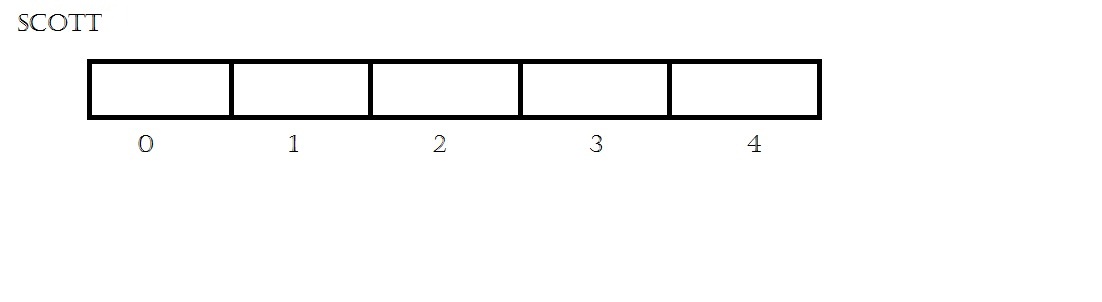
***What is it, how they work?***

An array is a series of numbers/letters/symbols that are the same of the same type (int, float, double, char etc…). Arrays are held in a computer’s memory in a strict linear sequence. An array does not hold any information other than the information that it is assigned. So there is no assigning an array of type *float* and hoping to stick a string in there. Doing so would cause a ‘type mismatch error’ and the program wouldn’t compile. To create an array, the user types into their compiler *(data type) (array name)[length of array]*. It would look like:

*char Scott [5];*

The char is the data type, Scott is the name of the array (you can be as creative as you want with the name) and the [5] is the size of the array. So *char Scott [5]* can hold 5 pieces of data that are the same data type, in this case *char*. Look at the diagram below for assistance.

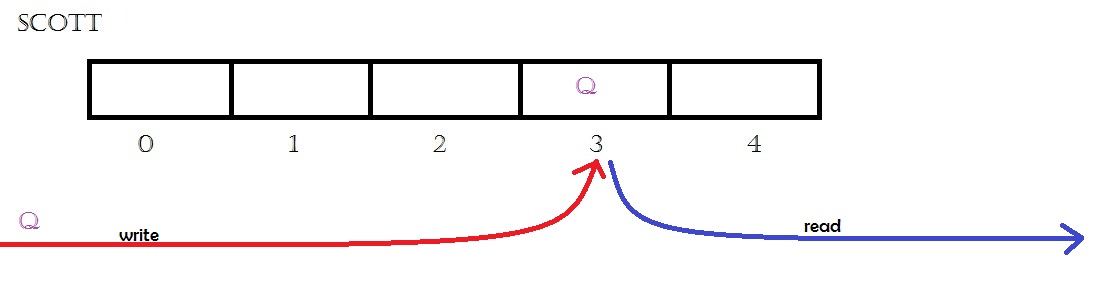
When trying to visualize an array, think of a rectangle with as many open slots in it waiting to be filled as the user defines. In the case of the above example, think of a rectangle with 5 open slots of type char that are waiting for some form of input.



When referring to the individual elements in an array, we start with the number 0 and count upwards. We count [0] as the first element in the array, [1] as the second, [2] as the third, and so on. In order to call certain locations of the array in order to read or write, we state the name of the array and the element we want to call. It should look like this (refer to the diagram to visualize how the compiler views the calling).

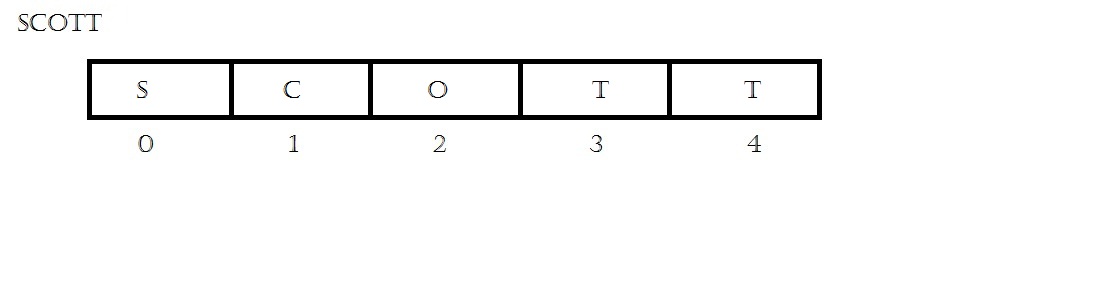
*Scott[3] = q;*

*cout << Scott[3];*

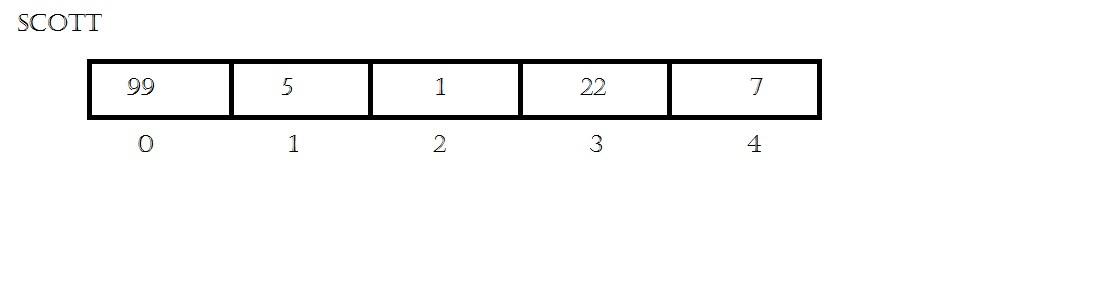


You can also hard code preset values inside the array when you are declaring the array. To do so, you need to enclose the values of the appropriate type in brackets and separate the values with a comma. Below are two examples, one of type *char* and one of type *int*.

*char Scott [5] = {‘ S’, ‘c’, ‘o’, ‘t’, ‘t’};*

**

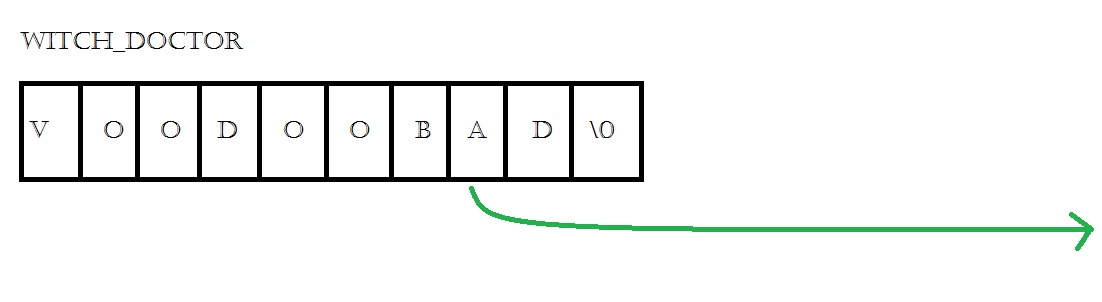
*int Scott [5] = {99, 5, 1, 22, 7};*

**

In the C and C++ language, the Witch Doctor demands payment for using strings and this payment is in the form of a null character. The null character marks the end of the array and takes up the last slot in the array, regardless of the length or user input. If the user calls in a 5 element *char* array and inputs five *char*s, the last one will be cut off and replaced with the null character. The null character looks like \0 (but remember, it only takes up one slot). In order to avoid information stored in an array from being cut off, it is good practice to declare the array at least one element larger than the size you want.

*char witch\_doctor[10] = {‘v’, ‘o’, ‘o’, ‘d’, ‘o’, ‘o’, ‘b’, ‘a’, ‘d’};*

*cout << witch\_doctor[7];*



***Follow the White Rabbit…***

A matrix is the same thing as an array, but is an “array of arrays.” Matrices take arrays to a two dimensional field or more. The way matrices work is the same way as arrays but with more brackets in the declaration. With a three dimensional matrix the first set of brackets are referring to the row, the second set is referring to the columns. The third set of brackets would be the length. For a truly mind boggling experience, try to think of an array in 4 dimensions, and then 5, and then 6. Remember, just because it is hard to visualize doesn’t mean a computer cannot. For a fun exercise, declare a 4 dimensional array and try to find the distance between 2 points on the array.

The declaration of a matrix looks like:

1. *int Rich [3][3] // 2D*
2. *int Rich [10] [10] [10] // 3D*
3. *int Rich [10] [10] [10] [10] // 4D…etc*

A user can input values into a matrix the same way as with an array. Hard coding or individual placements of variables into the matrix are the primary ways of inputting values.

*int main()*

*{*

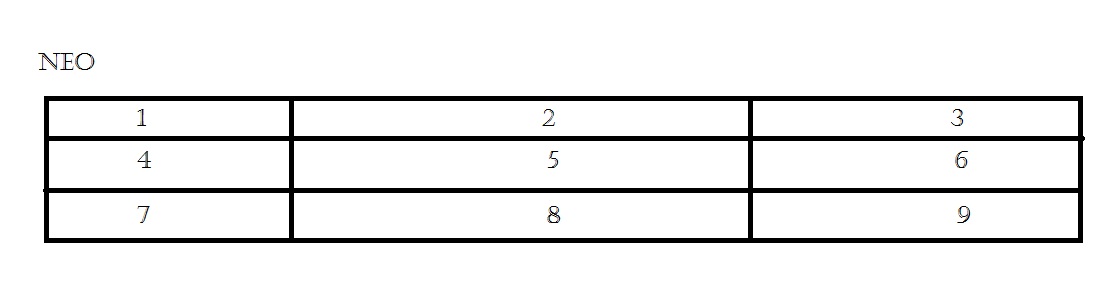
*int neo[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} }; // filling matrix with set numbers*

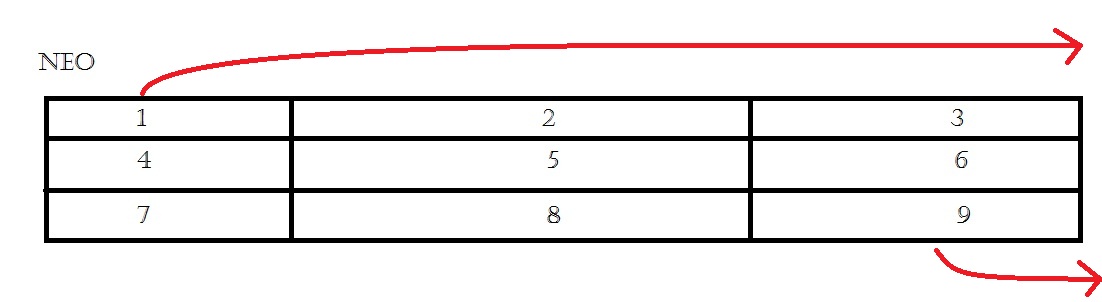
*cout << neo[0][0] << endl <<endl; // first number*

*cout << " " << neo[2][2]; // last number*

*return 0;*

*}*

**

**

The same logic is applied for 3 dimensional and 4 dimensional arrays, but when filling them, be mindful of the order of the input so that when you want to view certain elements in the array you are able to correctly call them and view them.

References:

**Other examples on filling array/matrix**

1) <http://www.cplusplus.com/forum/beginner/43663/>

2) VIDEO <https://www.youtube.com/watch?v=SFGOAKYXfOo>

3) <http://visualcplus.blogspot.com/2006/03/lesson-15-matrixes-and-2d-arrays.html>

**Making a array/matrix inside a pointer**

1) <http://stackoverflow.com/questions/256297/best-way-to-represent-a-2-d-array-in-c-with-size-determined-at-run-time>

2) <http://forums.devarticles.com/c-c-help-52/c-pointer-to-multidimensional-array-11075.html>