# Note 3.3– Why Computer Science Starts Counting at Zero

1. Computer scientists start counting at zero for a number of reasons one of which is that it makes calculating access into an array easier.

1. **One dimensional Arrays**. Consider the following **1D** integer array foo (using base 10 integers):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Value | 54 | -71 | 21 | 42 | -6 | 11 | -16 | 36 | -8 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Address | 4536 | 4540 | 4544 | 4548 | 4552 | 4556 | 4560 | 4564 | 4568 |

* 1. Foo[0] has value 54 and is at bytes 4536, 4537, 4538, and 4539.
  2. We can calculate our way into the array using the index times the offset of the number of bytes per cell. Thus Foo[6] has value -16 and is at location addressOfFirstByteOfFirstCell+(Index\*NumberOfBytesPerCell) = 4536+(6\*4)=4560.
  3. Note this allows us to access any cell in the array in constant time. Note how this differs from a linked list that takes N steps to locate the Nth value.

1. **Two Dimensional Arrays**. We can also map the above 1D array as a 2D array. Consider the 2D integer array Zoo shown below. Note that the first row of the 2D array is the first three values of the above 1D array, the second row is indexes 3, 4, and 5 in the above 1D array, and the third row of the 2D array is indexes 6, 7, and 8 in the above 1D array. This is called row-major order. We could also store an array in column-major order, with the first three values in the 1D array 54, 42, and -16. FORTRAN used column-major order while virtually all other programming languages use row-major ordering. (see <https://en.wikipedia.org/wiki/Row-major_order> <http://stackoverflow.com/questions/32253578/difference-between-row-major-and-column-major> ) For a 3D array, the next plane in the Z dimension (going into this page) would be stored in the 1D array after the first plane.)

|  |  |  |  |
| --- | --- | --- | --- |
| Row/col index | 0 | 1 | 2 |
| 0 | 54 | -71 | 21 |
| 1 | 42 | -6 | 11 |
| 2 | -16 | 36 | -8 |

* 1. Zoo[1,2] is stored at addressOfFirstByteOfFirstCell+( rowIndex \* <NumberOfColumns> + columnIndex) \* numberOfBytesPerCell = 4536+(1\*3+2)\*4 = 4536+20=4556 which has a value of 11.

1. Note that we need the number of columns to calculate our way into a 2D array, but we do not need to know the number of rows. So as long as we trust the coder not to have a bug and to try to access a non-existent cell off the end of the array, a function only needs to have the number of columns defined in order to access cells within a 2D array.
2. In most modern programming languages, arrays know their own size. However, in C++/C and other obsolete languages, that is not the case. With such languages, we need to specify the number of columns in a 2D array in the function header. Further, because accessing a memory cell takes more time that using a constant, that number must be baked into the actual machine code/assembly, so we need to use either a constant or literal to specify the number of columns in the function header. (We just about always use a constant rather than a literal as a basic rule of coding is that we never bake the same number in more than one location in our code as inevitably, we will change one occurrence and not the others, creating a major bug. We also almost always use a constant – not a literal -- as the constant to define array size allows us to tell our fellow coders (and ourselves at a later time) what that number means. 42 is not nearly as useful as arrayASize.
3. We do not need to specify the number of rows as that would only be useful if C++ actually checked to insure our array indexes were actually inside the array. That check would take a few extra machine code instructions and C++ is all about speed at the expense of coding convenience and avoiding bugs and security problems.