Christopher Lawrence

Professor Joseph Jupin

CIS 1057

3 November 16

Technical Analysis of Arrays

In programming, across most languages, a common and vital data structure is an array; a piece of data that holds n elements of data of various types within itself. Arrays group data stored within themselves within main memory instead of grouping individual data types within single cells of memory. When programming it is often more effective and efficient to group data within main memory through the use of arrays and in doing so it often produces a more elegant solution to the problem at hand. Arrays are a part of a higher level of data called data structures and what this essentially means is that the structure alone is not a singular piece of data but rather a structure that relates all of the data within itself to one another. A more formal definition of an array is a collection of two or more adjacent cells in memory. This ultimately means that when data is stored in arrays the data is stored in memory cells directly next to each other. Within this paper, I will delve deeper into the declaration of arrays, multidimensional arrays, processing of arrays, algorithms that can be applied to arrays, and how arrays can be passed.

Arrays of all sorts are generally declared in very similar syntax. This syntax can be represented by datatype name[n]; but if you want to elements to the array or make the array a multidimensional array the syntax will differ. Suppose that an array of 10 single digit integers in ascending order with no duplicates is declared, the declaration of this data structure in C would be represented by: int a[] = {0,1,2,3,4,5,6,7,8,9};. The meaning of the syntax shown is depicted as follows, int determines the data type that the array contains and this is supported by referencing the data declared within braces, which are clearly of the data type int. After the declaration of the data type, the array like any other piece of data must be given a variable name, in this case, it is a. Directly succeeding the name of the array is a pair of brackets [], thus initializing the variable as an array. After the name and variable type has been declared, given our syntax we must now declare the elements of the array. The pieces of data that we want to store within the array must be contained within a pair of braces, {}, and be entered as a comma-separated list. In our example, our array does not have an explicitly defined size through the use of the syntax: [n], so although we have said that we want to declare an integer array of size 10, we could go beyond that size. It is clear that following our previously defined syntax, that if we want to initialize 10 elements to an array, it is contained as a comma-separated list within braces, succeeded by a semi-colon to complete the statement. If you were to declare a multidimensional array of n dimensions, you would need to add n pairs of brackets succeeding the variable name. For example, if an integer array of three dimensions is being declared of a size 3 by 3 by 3, 27 elements, the syntax representation would be int a[3][3][3];.As you can see, the syntax is essentially the same between not only different types but also through multidimensional arrays as well. If you wanted to manually input the elements into a multidimensional array, similarly to how the first example did, you would just have to replicated the process n times over, being sure to encapsulate each dimension of the array appropriately. Given the generally simpler form factor of one-dimensional arrays, it is common for the author of the program to initialize the elements within the array manually, but as arrays grow in size and dimensions it is common to declare the elements within the array through the use of loops. The common approach is to use n amount of for loops for the dimensional complexity of the given array. For example, if I were to initialize a two-dimensional array of characters, also known as a matrix, of size 16 by 16, I would a for loop within a for loop to traverse and initialize individual elements within that array. The code for the operation would be in resemblance to:

Example 1.1

char a[16][16];

int i = 0, j = 0;

for(i = 0; i < 16; i++){

for(j = 0; j < 16; j++){

a[i][j] = ‘A’;

}

}

A lot of new information was covered here in terms of arrays but the information that you should be able to flesh out at this point is that each element in the array a is now the character A. It should be noted that arrays are not limited to be comprised of certain data types, they are a very versatile data type that can be made up of all data types, including data structures such as arrays, which is essentially what multidimensional arrays are.

When processing arrays, it is most common to access the elements within the array through an iterative solution. In Example 1.1, an iterative solution to access the individual elements of a two-dimensional array is shown. The syntax shown accesses each piece data in the row of index i through the use of the inner for loop as the outer for loop allows the inner for loop to work through each row. Within the inner for loop, we can see that the data within the array is accessed through the syntax of a[i][j]. Although this syntax is technically unique to this situation, we can apply the index rule shown in Example 1.1 to all types of arrays. Because the array shown above is two-dimensional array we must use two indexes to access a singular element within the array. A rule that is applicable to all arrays is that I the dimensions of the array is represented by n, n indexes must be called through the use of brackets to access a singular element. Suppose there exists a four-dimensional array defined as int a[3][4][4], the syntax to access a singular element in this array would be a[x][y][z], assuming x is less than 3, y is less than 4, z is less than 4, t is less than 3, and all of them are integers. Suppose a is declared as follows:

int a[3][4][4] = {{{1,2,3,4},

{5,6,7,8},

{9,10,11,12},

{13,14,15,16}},

{{16,15,14,13},

{12,11,10,9},

{8,7,6,5},

{4,3,2,1}},

{{1,2,3,5},

{7,11,13,17},

{19,23,29,31},

{37,41,43,47}}};

If we accessed the piece of data at a[1][3][2]we would be accessing 2. This brings up a key point in the world of arrays, when counting indexes for arrays, the first element is always indexed at 0. Suppose that the expression printf(“%lf”, a[0][0][0]); is called, the value 1 would be printed to the console because it is at the first indexes for the given array, therefore we can conclude that arrays start index counting at 0. The processing of arrays is not only limited to initializing or printing elements of the arrays. Essentially the same methods used in Example 1.1 can be applied to various tasks such as data analysis, data mutation, searching and sorting, and even neural networks, but that goes a bit beyond the scope of this paper.