Chapter 2 Part 2 – Multidimensional Arrays

Introduction

You know the basics of arrays and understand why they are important. However, we focused mostly on linear or 1D array in our earlier discussion. Let us focus on multidimensional arrays now. Multidimensional arrays give us a natural way to deal with real-world objects that are multidimensional, and such objects are quite frequent. Consider an image rendering operation to display a picture on the screen. It is convenient to use a 2D array of pixels to represent the picture to decide how to magnify and align the pixels of the image with the points of the 2D screen. For another example, if you do computation related to heat propagation on a 3D object as part of a scientific study of solid materials' heat conductivity; it is natural to use 3D arrays to represent the solids. Take this second example. Suppose you want to read the current temperature at the <x,y,z> point of a 3D plate variable representing a solid, you can simply do this as follows:

cell_temperature = plate[x][y][z]

You see how convenient it is to write computations using multidimensional arrays! Fortunately, an element access from multidimensional arrays is as efficient as it is convenient. To understand the efficiency, we need to see how languages allocate multidimensional arrays and locate the memory address for a particular multidimensional index.

Multidimensional Arrays in Memory

Interestingly – but not surprisingly – programming languages store multidimensional arrays as liner arrays in the RAM. This makes sense, right? As we know the RAM is a linear array of memory cells. Let us see how it works. Remember that, an array contains elements of a single type and its number of dimensions and dimension lengths are given when you create it. Therefore, a language can follow the simple technique of placing all elements for increasing values of the index along a single dimension in consecutive locations in the memory while keeping the indexes along all other dimensions fixed. After it is done traversing a dimension, then it increases the value of the index along another dimension and restart from the zero index of the previous dimension. Following this process until the indexes along all dimensions become the maximum possible, translates the whole multidimensional array into a linear array. There is just one restriction; a language must follow a fixed dimension ordering rule to store all multidimensional arrays. There are two standards here:

1. Increasing the final dimension's index first and progressively move to earlier dimensions, this is called the Row-Major Ordering. For example, Java and C take this approach.
2. Increasing the starting dimension's index first and progressively move to later dimensions, this is called the Column-Major Ordering. For example, FORTRAN takes this approach.

The following diagram shows the difference between these two approaches for a 2D matrix of 3×3 dimensions. (By the way, row is the vertical and column is the horizontal axis in 2D. You should already know that. Here Dimension 0 represents the row and Dimension 1 represents the column.)

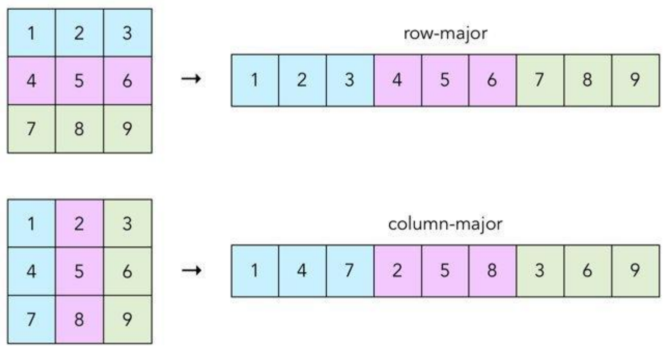

Figure 1: Two Different Ways of Storing Multidimensional Arrays in a Linear Format

There is a great impact of different languages choosing different ordering of dimensions for storing multidimensional arrays. The impact is related to what is efficient in computer hardware and what is not. The way modern computers are build, it is several times more efficient to access data to/from consecutive memory locations than from locations that are distant from one another. As a result, if you iterate the elements of a multidimensional array in an order that is different from how they are stored in memory then your program can be several times slower. Now given the index ordering chosen by Java is radically opposite from how FORTRAN does it, if you simply translate an efficient Java code into an equivalent FORTRAN code then the performance will tank.

The important lesson here is that efficient programming is often language dependent.

The following diagram shows how the row-major approach works for a 3D 3×3×3 cube. From this example, you understand the higher dimension count of the original multidimensional array is not a problem.

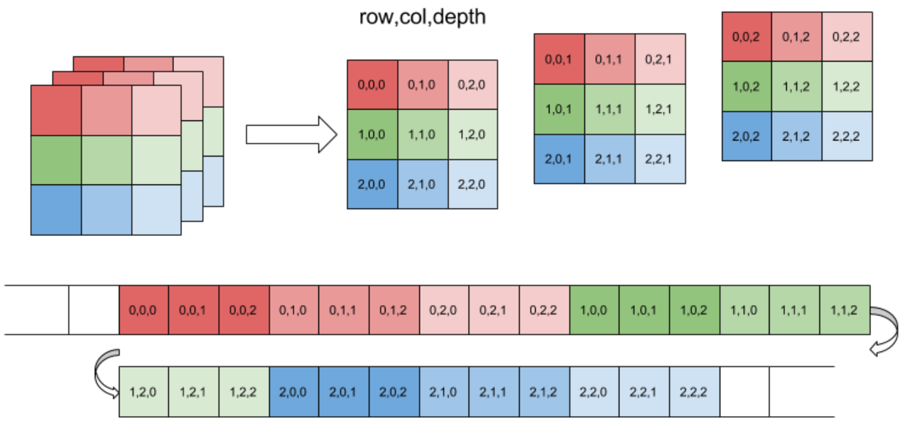

Figure 2: Row Major Ordering of Indexes of a 3D Array

Let us stick to the row-major ordering, as that is more common in application programming. You should be interested to know how to determine the translated linear index of a multidimensional index from the original array. For that, imagine you have a 4D array with dimensions M×N×O×P, named *box,* and the element at the index you are interested is the *box[w][x][y][z]*. Then the index of that element in the linear array is:

$$w \times (N \times O \times P) + x \times (O \times P) + y \times P + z$$

You understand the general rule, right? Basically multiply the upper dimension indexes by the multiplication of the lengths of lower dimensions then add them all. So easy, isn't it?

Do a practice of the reverse operation. Suppose a linear array of length 128 actually stores a 3D array of dimensions 4×4×8. What are the multidimensional indexes of the element stored at location 111? Remember that indexing starts from 0 in each dimension.

Since it is so simple to represent multidimensional arrays as linear arrays in memory, some languages do not support multidimensional arrays as building block data structures at all. They only support linear arrays. The idea is, since it is such a simple computation to go back and forth between multidimensional and linear arrays, let programmers handle the logic of index conversions and keep the language simple. C is a classic example of such a language that does not allow creation of multidimensional arrays dynamically (which means during program runtime).

You have to admit that this is a valid argument.

Solution of the Example Problem:

The dimension of given array is [4][4][8] and length is 128. We need to map the dimension of linear index 111.

The equation we get is:

X * (4*8) + Y*8 + Z = 111 where 0 <= X <= 3, 0 <= Y <= 3 and 0 <= Z <= 7.

Now to find the value of X, Y and Z we need to start repeatedly divide the remainder of the linear index with the multiplication of the lengths of lower dimensions till you reach the last dimension.

So first 111/(4*8) then the quotient is x and then you use the remainder R to do R/8 to get y index. The remainder of that is the z index

Following the process we get,

X = 111//(4*8) = 3 and 111 % (4*8) = 15

Y = 15 // 8 = 1 and 15 % 8 = 7

Z = 7

So, the dimension of linear index 111 is [3][1][7].

Now practice finding the dimension of 96, 107, and 60 of the linear index.