- Dream.In.Code> Programming Tutorials
- > C++ TutorialsPage 1 of 1

Multidimensional Arrays Introduction to Multidimensional Arrays, Pointer Arrays and Vectors Rate Topic: 1 Votes

ccubed
Posted 07 December 2009 - 10:37 AM

Welcome to the wonderful world of Multidimensional Arrays. In this tutorial, we'll discuss Multidimensional arrays using regular array format, vector format and pointer format as well as all manner of neat tricks and basic functions for them. I'm not one for longish type openings, so let's just hit the code.

Part 1: Quick Review of Arrays

This is a quick review of one dimensional arrays before we hit two dimensional arrays.

A one dimensional array is an array that only contains 1 row of information. The number it is initialized with is the number of units in that row, you can also call it column.

So a 1x4 array looks like this

0|1|2|3

Now, to declare that in C++ we have three options.

We can use regular array format.

```
1 int a[4];
```

We can use pointer array format.

```
1 int *a = new int[4];
```

Or we can use a vector.

```
1 std::vector<int> a;
```

Note that we don't need to tell vectors how much information is going in them since they auto resize to fit it.

And that's all for the review. I'm assuming that if you're reading this you know how to iterate through and access any member of these three types of arrays. If you don't, perhaps you should start by reading about arrays.

Part 2: Two Dimensional Arrays

A two dimensional array is something we're all use to. We've all had plane geometry in algebra so the X Y structure shouldn't scare anyone off. Let's get into regular array format first. Also note that Vectors have their own section later on.

Part 2.1: Regular 2D Arrays

A two dimensional array in regular array format.

```
int a[5][5];
```

This code creates a 5x5 array of integers. Now, one important thing to know is how to calculate the number of elements. Quite simply, if I have this declaration.

```
int a[x][y];
```

The number of elements is x*y and the memory required to hold those elements is sizeof(int)* (x*y). Now, this becomes increasingly important later on when we talk about X Dimensional arrays. Now onward to some simple functions of 2 Dimensional Arrays using regular array format.

```
//Our declaration: Assume we know X And Y
int a[x][y];

//Iteration
for( int i(0); i < x; i++ )
    for( int j(0); j < y; j++ )
        //do something to each element here

//Accessing element at 3,3
a[3][3];

//Dynamic Allocation
int b = 3, c = 3;
int d[b][c]; //Yes! This does work

//Special Note: passing to functions
int foo( int d[][3] )
{

    return d[1][1];

}
```

That concludes the basic functions for two dimensional arrays in regular array format. Now, note that when I pass my 2D array to a function, I HAVE to pass the number of columns. It doesn't need the rows, but it needs to know how many columns there are.

Part 2.2: 2D Arrays using Pointers

Now let's hit 2D Arrays using pointers. This is going to be one of those WTF sections, so i'll try to be extra explanatory and go slow.

```
int **a = new int*[height];
for( int i(0); i < height; i++ ) a[i] = new int[width];
```

The above code declares an array of integer pointers to integer pointers. The first thing we do is then make a point to an array of integer pointers. Then we have to loop through all of those pointers

and assign them to an array of ints of size width. What this does is create an array in the size of Height x Width.

Now, let's look at looping through a 2D pointer array.

```
for( int i(0); i < 5; i++ )
    for( int j(0); j < 5; j++)
        cout << list[i][j];
```

Notice how that didn't change the syntax we use? So really the only difference is to remember that it's a pointer to pointers to arrays. So that you really need to use Delete or Free with these things.

Part 2.3: A few notes

I cannot stress enough that there is NO EASY WAY to determine the size of a 2D array without at least knowing either the Rows or the Columns.

Also, it is possible to declare the number of rows when passing 2D arrays. This will not affect how you code, but will force the function to accept an array with only that many columns and that many rows. For example.

```
int foo( int d[3][3] )
{

    return d[1][1];

}
```

The function foo above will ONLY accept 3x3 arrays of integers as its input. This can be useful when limited the size of constructs past to a certain function

Part 3: Three Dimensional Arrays

Assuming you aren't dead yet, I'd go get something to eat and drink then come back.

Done? Great. So, now let's tackle THREE dimensional arrays. Don't explode on me yet.

Part 3.1: 3D Arrays using regular array format

Let's start this section off as usual with declaring an array of integers.

```
int a[5][5][5];
```

The code aboves defines an array, a, that consists of 5 rows, 5 columns and each column contains 5 data items. This could also be used to represent an XYZ Coordinate plane much like a 2D can be used to represent an XY Coordinate plane.

Now, how to iterate through this thing.

```
for( int i(0); i < 5; i++ )
    for( int j(0); j < 5; j++ )
        for( int k(0); k < 5; k++ )
            a[i][j][k];
```

See, not that much harder. All we did was add an extra for loop. Now let's talk about other cases.

```
//Dynamic Allocation
int b = 3, c = 3, d = 3;
int e[b][c][d]; //Yes! This does work

//Special Note: passing to functions
int foo( int e[][3][3])
{

    return e[1][1][1];

}
```

Note that nothing really changed here either except that I had to add in the 3rd dimension explicitly to function calls. This brings up a coding practice in C++. Any dimension after the first has to be explicitly passed in function calls.

Part 3.2: 3D Arrays using Pointers

Now to pointer arrays. Prepare to die on me, but try to pull through okay?

```
int ***a = new int**[Dim1];

for( int i(0); i < Dim1; i++ )
    a[i] = new int*[Dim2];

for( int i(0); i < Dim1; i++ )
    for( int j(0); j < Dim2; j++ )
        a[i][j] = new int[Dim3];
```

Dead yet? No? Good. What that code does is that it declares an array of integer pointers to an array of integer pointers to an array of integer pointers that point to an array of numbers. Basically, the pointers simulate dimensions while the 1st dimension holds all the data. Does anyone see the pattern here yet?

```
//Iteration
for( int i(0); i < Dim1; i++ )
    for( int j(0); j < Dim2; j++ )
        for( int k(0); k < Dim3; k++ )
            a[i][j][k];
```

And that's all for this section.

Part 4: Multidimensional Vectors

Multidimensional Vectors are perhaps the most daunting before being explained. I'll note before we

hit this that multidimensional vectors work much in the same way our pointers do. I'll explain this further as we go along.

Part 4.1: 2D Vectors

Let's set up a simple 2D vector.

```
1  vector< vector<int> > a;
```

What that code does is declare a vector that contains an unlimited number of vectors that contain an unlimited number of integers. That is, each index in a is a vector of ints. Now, doesn't this sound familiar? It kind of looks like what we did using pointers doesn't it? We had to declare an array of integer pointers to an array of integer pointers to an array of ints to accomplish 2D arrays. This is the exact same thing except it resizes itself automatically. You also need to remember that these are VECTORS, meaning that you use them the same as you would a regular vector.

```
01  //iteration
02  for( int i(0); i < a.size(); i++ )
03      for( int j(0); j < a[i].size(); j++ )
04          a[i][j];
05
06  //accessing 2,2
07  a[2][2];
08
09  //size of first row
10  a[0].size();
11
12  //number of rows
13  a.size();
14
15  //passing to functions
16  int foo( vector<vector<int>> a){ return
    a[0][0]; }
```

See, not that much different from a[2][2] except that we need to remember that a[0] is a vector and that a[0][0] is an element of the vector stored at position 0.

I don't think I need to invest anymore time here. I would add a 3D multidimensional vector, but I believe the next part will explain the algorithim enough that you won't need the tutorial.

Part 5: X Dimensional Arrays - THEORETICAL SECTION

Yes, X Dimensional Arrays. Now, I'm not saying 10 here, I'm saying X as in insert number there. I'm going to give you algorithims for creating Arrays of any number of dimensions. Now, I have no idea how many dimensions your particular system would be able to handle, but I can say that the most i've ever tested up to is 9. Now, I don't exactly expect that anyone would ever NEED more than 3 or 4 dimensions, however, I think that knowing an algorithim is better than learning one way of doing things so i'll explain the basic algorithim for each type of array.

Part 5.1: X Dimensional Arrays - Regular Array Format

I think that this one is pretty obvious, but let's examine. If we want X Dimensions we have to first have the size of each dimension and then do this.

```
1 <type> <identifier>[<dimension size>]
  [<dimension size>]...;
```

Here's the iteration algorithim.

If we have an X dimension array, we need X loops to properly traverse the entire array. I will not show psuedo code for this as you should alreay understand what I mean if you've gotten this far.

Now, for passing to functions, as I said above, you MUST pass the size of EVERY dimension AFTER the first.

And that concludes the algorithims for X Dimensional Arrays in Regular Array Format.

Part 5.2: X Dimensional Arrays - Pointer Array Notation

This one gets a littler harder, but it's really quite simple once you get it. I'll try to explain it the best I can.

If we want an X Dimensional array, we need to declare it like so.

```
1 int (* - Repeat it X times)a = new int(*
  - repeat it x-1 times)[Dim1];
2
3 for( int i(0); i < Dim1; i++ ) a[i] = new
  int(* - repeat it x-y times)[Dim2];
4 //A bunch of for loops
```

Now, to explain that. We need to type <type>, then * X times, then the identifier and then we set it equals to new int, then we write * x-1 times, then we write [] and put our first dimension size into them.

Okay, now we have to loop through EACH level down to the last one and assign to an array of pointers. To do that we do a for loop for the first dim, then we set each element, that is a[i] to new int and, this is important, the * decreases by 2 the first time, 3 the third time, etc. So we keep removing 1 * each time we make a new int array until we have 0 *. We also have to do each level individually like we did for the 3D array so this method is LARGELY inefficient. I hope that makes sense, but otherwise, there's not an easier way to explain it.

After that everything stays the same. iteration is done through X loops and passing to functions requires the same as a regular array does. I won't spend anymore time here because I really feel it's too inefficient to use as a method past 2.

Part 5.3: X Dimensional Arrays - Vectory Style

Oh yeah, now we get to the easy part. What's easy about it you ask? Well, once you understand a 2D vector, you understand any number of Dimensions in a vector. So let's jump into it.

To have an X Dimensional Array in a vector we

need to have this format.

```
1 vector< vector< vector<(So on and So Forth to desired amount)<type>>>(so on and so forth);
```

Now, what that essentially means is that our X dimensional array is really X vectors of Vectors of ints down to the last one which really contains the data.

Iteration and functions are the same here as they have been so I won't add anything there since this section is really meant to be theoretical.

Part 6: Closing Statements

I hope that this has cleared up arrays for you. I'm fairly sure that Part 5 was where you stopped and decided to skip down to 6 and that's fine. Part 5 is mainly for those who like to ask, well what if it was X Dimensions. That being said, I need to draw one final Distinction for you to help you better understand arrays.

Now, take a 2D array. What you need to understand is that the 1st dimension is simply for a cell. For example, if you had an excel document, the 1st dimension would be the number out to the left. The 2nd dimension is the DATA. That is, in an excel document, it'd be the Number or name next to the number to the left. Now, another distinction is that every dimension before X - Where X is the number of Dimensions - is simply a placeholder. The actual DATA exists always in the last dimension.

I would draw pictures for this, but I'll just go with painting a text description in your head. Assume that ALL Dimensions are initialized to 5.

A 3D array is actually an array of 5x5 matrices.
A 4D array is an array of arrays of 5x5 matrices.
A 5D array is an array of arrays of arrays of 5x5 matrices

Etc and so on and so forth.

That's all. You've graduated from the Multi-Dimensional Array school.

Replies To: Multidimensional Arrays

DudewhereismyCar
Posted 17 June 2011 - 07:51 PM

Nice, to the point, explanations. Thanks. One question though; In the for loop headers the counter is initialized as, int i(0): Is this a C++ convention or from a different language?

qasimbilal
Posted 29 June 2011 - 11:24 PM

Yes it is confusing a little bit because **int i(0)** is used in loops. Can someone explain it??

Deca
Posted 09 July 2011 - 02:34 PM

thanks

tecnoupdates
Posted 01 September 2011 - 04:04 AM

thanks dude

PlasticineGuy
Posted 01 September 2011 - 10:52 PM

int i(0); calls the type int's constructor. It's just like any C++ class.

ccubed
Posted 24 December 2011 - 04:40 PM

Having now seen these comments, the answer.

Int i(0) is just another way of initialization. Take the following code.

```
1 Int i(0);
2 Int i=0;
```

All of these do the same thing, assign i the value of 0.

Related C++ Topics[beta]

- Understanding Dynamic Multidimensional Arrays
- Tic Tac Toe With Multidimensional Arrays
- Multidimensional Arrays As Parameters To Functions.
- Multidimensional Arrays In C
- Multidimensional Arrays
- Returning Multidimensional Arrays
- Question Concerning Classes And Multidimensional Arrays
- Strange Multidimensional Arrays - Unexplainable Behaviour

Multidimensional Arrays Contain Garbage?

Multidimensional Arrays And Pointers - Having Trouble Creating A Pointer To A Multidimensional Array.