ColdFusion Lists, Arrays, and Structures

Jeff Peters

Proton Arts – Manassas VA USA

ColdFusion Lists, Arrays and Structures

FIRST EDITION: February 2004

08 07 06 05 04 7 6 5 4 3 2 1

ISBN 0-9752647-0-2

Interpretation of the printing code: The rightmost double-digit number is the year of the book's publication; the rightmost single-digit number is the number of the book's publication. For example, the printing code 04-1 shows that the first publication of the book occurred in 2004.
Published in the United States of America

Trademarks

All terms mentioned in this book that known to be trademarks or service marks have been appropriately capitalized. Proton Arts cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark. ColdFusion is a registered trademark of Macromedia, Inc. Fusebox is a registered trademark of Fusebox, Inc. BlueDragon is a trademark of New Atlanta Communications.

Warning and Disclaimer

This book is designed to provide information about ColdFusion programming. Every effort has been made to make this book as complete with respect to its topic and as accurate as possible, but no warranty of fitness is implied.
The information is provided on an as-is basis. The author and Proton Arts shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs that may accompany it.

This book was written and set with OpenOffice.org 1.1.0

the array. This effectively transforms the array into a string variable with an empty string as its value:

```
<cfset aryMenuItems = "">
```

or

```
<cfscript>
  aryMenuItems = "";
</cfscript>
```

If we now test the variable aryMenuItems with the IsArray() function, the result we'll get is **NO**.

Multiple Dimensions

Arrays in two or more dimensions are not much more complex than arrays in one dimension. The trick to the whole game is to recognize that a multi-dimensional array is simply a nested set of arrays. To clarify, a two-dimensional array is an array whose elements are themselves arrays.

The nature of multi-dimensional arrays might become a bit clearer if we examine how arrays are created. The ArrayNew() function takes one argument, which is an integer from 1 to 3. This argument specifies the number of dimensions to be used for the new array. So, to define a two-dimensional array, the typical syntax looks like this:

```
<cfset aryTeammates = ArrayNew(2)>
```

or

```
<cfscript>
  aryTeammates = ArrayNew(2);
</cfscript>
```

Let's say we need to store three sets of data in our new array (three values in the first dimension). If we knew this in advance, we could define the array this way:

```
<cfset aryTeammates = ArrayNew(1)>
<cfloop from="1" to="3" index="i">
  <cfset aryTeammates[i]= ArrayNew(1)>
</cfloop>
```

The end result is a two-dimensional array that has three elements in the first dimension. As you can see, this syntax, though more cumbersome than the first method, clearly illustrates the nested nature of multi-dimensional arrays.

Why is this important? The ArrayNew() function limits array creation to three dimensions. If we know that a multi-dimensional array is just a nested group of arrays, we can create arrays in as many dimensions as we want.

In turn, if we have the ability to manipulate arrays in any number of dimensions without the restrictions of the ArrayNew() function, we've removed one of the arbitrary limitations of CFML. The next sections are essentially duplicates of the preceding Array How-To, with examples for multi-dimensional arrays.

Array How-To (Multiple Dimensions)

The key key to dealing with multi-dimensional arrays is very simple: a multi-dimensional array is just a set of nested one-dimensional arrays, no matter how many dimensions it has.

To put it another way, the contents of every element of a multi-dimensional array is another array, with the exception of the last (rightmost) dimension. The following illustration shows a two-dimensional array that stores values for an x,y coordinate system. Note that only the elements in the second dimension contain numbers; the elements of the first dimension contain arrays.

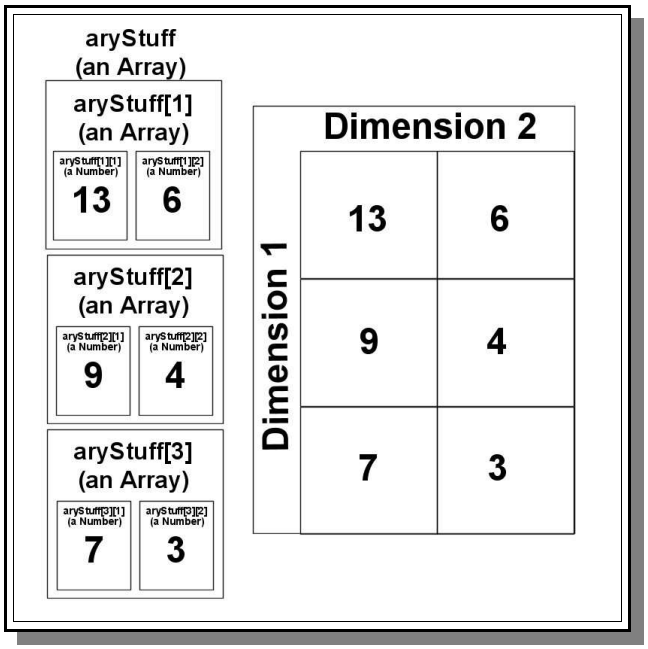

Illustration 3 - Two Views of a Two-Dimensional Array

This is the case regardless of how many dimensions the array has—every element in each dimension is an array, except for the last dimension. Knowing this, we can apply any of ColdFusion's array functions to any element in a multi-dimensional array, with the exception of the elements in the last dimension. So, in the case of the example in Illustration 1, the following code is a valid way to add a fourth set of coordinates:

```
<cfset aryStuff[4] = ArrayNew(1)>
<cfset aryStuff[4][1] = 17>
<cfset aryStuff[4][2] = 8>
```

With all that said, we can move on to look at specifics for creating, adding and deleting elements, and looping with multi-dimensional arrays. Then we'll wrap up this chapter with some examples of arrays in action. We'll start with a look at a speedy method for creating multi-dimensional arrays.

Create an Array (Multiple Dimensions)

Keeping in mind that multi-dimensional arrays are simply arrays of arrays, the creation process is just the one-dimensional process repeated for as many dimensions as we need. The process can be implemented very nicely with the use of loops.

As an example, let's say we need to create an array to store the temperature for the first 20 days of each month for the last 10 years. This means we need a three-dimensional array (days by months by years) to manipulate the data.

We can create this multi-dimensioned array very quickly with a looping technique. Since we know that we're going to have 10 years, 12 months in each year, and 20 days in each month, the array can be created with the following code (we'll default each value to 0 to make sure all our numeric manipulations work):

```
<cfset aryTemperatures = ArrayNew(1)>
<cfloop from="1" to="10" index="thisYear">
  <cfset aryTemperatures[thisYear] = ArrayNew(1)>
  <cfloop from="1" to="12" index="thisMonth">
    <cfset aryTemperatures[thisYear][thisMonth] = ArrayNew(1)>
            <cfloop from="1" to="20" index="thisDay">
                  <cfset aryTemperatures[thisYear][thisMonth][thisDay] =
0>
            </cfloop>
  </cfloop>
</cfloop>
```

Of course we could also have created this specific array with the syntax `<cfset aryTemperatures = ArrayNew(3)>` and populated it with zeros through nested loops. Though true for this example, the idea is to see that we can create any necessary number of dimensions through the use of loops; we're not constrained to the limitation of 3 dimensions imposed by the `ArrayNew()` function.

Add Elements To an Array (Multiple Dimensions)

Adding elements to a multi-dimensional array is only slightly more complex than adding elements to a one-dimensional array. The thing to keep in mind

is that we only store actual data in the last dimension. Adding an element to any dimension other than the last one calls for adding an array.

So, to add an element to our aryTemperatures array (let's say we want to store a temperature for the twenty-first day of January of the first year), we could just do this:

```
<cfset aryTemperatures[1][1][21] = 34>
```

Depending on what we intend to do with our data, though, this might cause a problem. We now have 21 elements in January of the first year, and only 20 elements in each of the other months. Since we're setting default values to zero, we should create a 21st day for each month, and set its value to zero, then set the value of the first January 1st to 34:

```
<cfloop from="1" to="10" index="thisYear">
  <cfset aryTemperatures[thisYear] = ArrayNew(1)>
  <cfloop from="1" to="12" index="thisMonth">
    <cfset aryTemperatures[thisYear][thisMonth][21] = 0>
  </cfloop>
</cfloop>
<cfset aryTemperatures[1][1][21] = 34>
```

So adding elements to a multi-dimensional array isn't too tough, as long as we keep in mind the idea that we're operating in a system of dimensions. Deleting elements, on the other hand, can get to be a bit tricky.

Delete Elements From an Array (Multiple Dimensions)

As noted in the Delete Elements From an Array section above, when you delete an element of an array in ColdFusion, the array closes up around the deleted element. That is, each element to the right of the deleted element shifts one step to the left, acquiring a new index number that is one less than its index number prior to the deletion.

This behavior makes sense from the point of view that arrays are designed to enable looping behaviors in code; we don't want holes all over the place to cause problems when we're running loops. However, it does mean that we