Understanding Arrays

> In Director, it is VERY common to have extremely deeply nested arrays,
> often with tens of thousands of items dispersed across many layers of
> array nesting. These nested arrays can contain both associative arrays,
> and linear lists, and those in turn can contain either as well, and so
> on. Contents of these lists are easily target-able, and very rapidly
> accessible.
>
> How is this done in Transcript, if at all? If not, what is the logical
> replacement? In some cases, I've gone to generating stacks and using
> cards to organize such data, but would prefer a variable structure for
> the speed, especially considering that the user never has to actually
> see the data in the stored/working form.

The first thing you need to "grok" is that the way Rev uses associative arrays and making what I call "string assumptions" can allow the arrays to mimic true multidimensional arrays, even though under the hood, they are considered to be single-dimension string-based arrays.

Consider:

```
put "hello" into myAlert[greeting]
```

In the case above, "greeting" is considered to be a string so long as no variable named "greeting" has been previously defined. So it is the equivalent of:

```
put "hello" into myAlert["greeting"]
```

Now for multi-demensional arrays, let's use a tic-tac-toe board. Many beginning tutorials on multidimensional arrays use this as an example. In languages like Flash, you'd refer to the upper left square like this (I'm using "cell" as the array variable):

```
cell[1][1]
```

In Rev, you would refer to it as:

```
cell[1,1]
```

But what's really happened is that Rev has created a single key using the *string* "1,1" in the array. To onlookers, it looks like a two-dimensional call, but it really isn't. So supposed you wanted to "clear the board" in tic-tac-toe using Rev. You might do something like this:

```
repeat with x = 1 to 3
  repeat with y = 1 to 3
    put "" into cell[x,y]
  end repeat
end repeat
```

Now at first you might think that the call to

```
put "" into cell[x,y]
```

would be interpreted as

```
put "" into cell["x,y"]
```

(i.e. rewriting a single array element every time throught the loop instead of creating all 9 array elements), but Rev's smarter than that... it knows that if what's in the [] is comma-delimited, AND the items within the [] are variables, it just substitutes the variables.

However, what REALLY happens, is that instead of Rev creating a 2-dimensional array, it creates a 1-dimensional array with 9 elements:

```
cell[1,1]
cell[1,2]
cell[1,3]
cell[2,1]
cell[2,2]
cell[2,3]
cell[3,1]
cell[3,2]
cell[3,3]
```

Now... how does that apply related to Director? Well I'm a bit rusty on Director lists (haven't used Director for a couple of years), but based on the MX docs and my rusty memory, here's how it applies...

LINEAR LISTS

The simple linear lists in Director are straightforward - compare:

Director

```
set myList = ["Troy","Ken","Richard"]
put getAt(myList, 2)

     RESULT: Ken
```

Revolution

```
put "Troy,Ken,Richard" into myList
split myList by ","
put myList[2]

     RESULT: Ken
```

(Of course in this simple example, in BOTH environments, it might just be easier to "get item 2" of the string, but I digress.)

More complex linear lists compare this way:

Director

```
set myList = [["Troy","Ken","Richard"],["Kevin","Jan","Judy"]]
put getAt(myList,2)

     RESULT: ["Kevin","Jan","Judy"]

put getAt(getAt(myList,2),1)

     RESULT: Kevin
```

Revolution

```
put "1,Troy,Ken,Richard"&cr&"2,Kevin,Jan,Judy" into myList
split myList by CR and ","
put myList[2] into tempArray

     RESULT: Kevin,Jan,Judy

split tempArray by ","
put tempArray[1]

     RESULT: Kevin
```

But note that this is NOT usually something you'd do in Rev... Director is designed to be able to make arrays of arrays, and although Rev can simulate this, it is not the most efficient, nor is it the best approach. One reason is that if you'll notice, I had to add line numbers before each line in **myList** so that when I used "split" I could reference the array by number. Why? Because remember that no matter what it may appear to be, Rev uses associative arrays ONLY - **myList[1]** is NOT numeric... it is creating a *string* called "1" that is associated with the value that you give it.

When you use "split" with only one delimiter (like in my first example), Rev *automatically creates* the string keys for the array and puts the index number in them. So when you do this:

```
put "Troy,Ken,Richard" into myList
split myList by ","
```

What you are really doing is having Rev say: "for each item in myList I will create a key for the array using a string that corresponds to the item number of the item I'm examining, and then set the value of that key to the string of the item I'm examining." So it's basically doing (I'm using quotes for clarity, but in practice you wouldn't use the quotes):

```
put "Troy" into myList["1"]
put "Ken" into myList["2"]
put "Richard" into myList["3"]
```

Now when you split on *two* delimiters, Rev first uses the first delimiter to get a "chunk" of data, and then based on the second delimiter it uses the first "item" of each chunk as the string for the key of the array, and sets the value of that key to the rest of the "items" of that chunk. So this:

```
put "1,Troy,Ken,Richard"&cr&"2,Kevin,Jan,Judy" into myList
split myList by CR and ","
```

Is the same as this (what Rev's doing under the hood):

```
put "1,Troy,Ken,Richard" into tChunk1
put "2,Kevin,Jan,Judy" into tChunk2
put item 1 of tChunk1 into tKey1
put item 1 of tChunk2 into tKey2
delete item 1 of tChunk1
delete item 1 of tChunk2
put tChunk1 into myList[tKey1]
put tChunk2 into myList[tKey2]
```

If I hadn't put the numbers in front of each line in myList, this is what I would have gotten instead:

```
put myList["Troy"]

     RESULT: Ken,Richard

put myList["Kevin"]

     RESULT: Jan,Judy
```

It is my experience that combining regular text parsing chunk expressions with arrays is the most useful. So in the second comparison above, I would do this:

```
put "Troy,Ken,Richard"&cr&"Kevin,Jan,Judy" into myList
split myList by CR
put item 1 of myList[2]
```

```
RESULT: Kevin
```

Of course this means looking at your data differently when you prepare to store it, but I'm sure you get the idea.

PROPERTY LISTS

These are actually more applicable in Rev since for Director these really are doing associative instead of numeric arrays. But the same approach with Rev holds true - rather than storing lists of lists (or arrays of arrays), you get creative in how you manage them.

Director

```
set customer = [#name:"Ken",#age:40,#gender:"male"]
put customer.age    -- or put getProp("customer","age")
```

```
RESULT: 40
```

Revolution

```
put "name,Ken" & cr & "age,40" & cr & "gender,male" into customer
split customer by cr and ","
put customer["age"]
```

```
RESULT: 40
```

Director

```
set customers = [#good:[#name:"Ken",#age:40],#bad:[#name:"James",#age:35"]]
put customers.good.name
```

```
RESULT: Ken
```

Revolution

```
put "name,1" & cr & age,2" into tLookup
split tLookup by cr and ","
put "good,Ken,40" & cr & "bad,James,35" into customers
split customers by cr and ","
put item tLookup["name"] of customers["good"]
```

```
RESULT: Ken
```

Keep in mind that a lot of this depends on how you assign the data to the array. You could store the data such that:

```
customers["good.name"] = "Ken"
customers["good","name"] = "Ken"
```

So an alternative storage approach (while longer) may make it easer for retrieval:

Store

```
put "name,age" into tProps
put "good,Ken,40" & cr & "bad,James,35" into tCustData
split tCustData by cr and ","
repeat for each line tKey in (the keys of tCustData)
  repeat with x = 1 to the number of items of tProps
    put item x of tCustData[tKey] into customers[tKey,(item x of tProps)]
  end repeat
end repeat
```

Retrieve

```
put customers["good","name"]
```

```
RESULT: Ken
```

I hope this helps with your understanding of arrays in Rev... personally I think that if we had a simple method of being able to assign data into an array like the one I'm referring to above, it would help speed things up and make it easier to work with. It's not that we need (IMHO) a way of being able to *actually* put arrays into arrays, but we need something that can allow us to *simulate* putting arrays into arrays, and allow us to retrieve it. IMHO, what's the difference between:

```
put customers.badcredit.state.california
```

and

```
put customers["badcredit","state","california"]
```

except a few extra characters to type? Would we *care* that Rev is storing the value of this element in the *actual* array customers["badcredit,state,california"]? I don't think so.

So we need something that will allow us to slice and dice the data quickly so that it can go into this kind of an array. Something along the lines of the way Director does it, but perhaps an extension of the split command:

```
split <var> {by|using|with} <primaryDelim> [and <secondaryDelim>] [and <propDelim>]
```

So this would work:

```
put "good,[name:Ken,age:40]" & cr & "bad,[name:James,age:35]" into customers
split customers by cr and "," and ":"
```

```
put customers["good","name"]

     RESULT: Ken
```

I'm not sure how this would extend to more than one array in another array... perhaps that's a useful discussion for an enhancement to Rev?

Anyway, now I'm rambling... I hope this has been helpful to both you and anyone else paying attention. ;-)

Posted 7/11/2004 by Ken Ray to the Use-Revolution List

Addendum By Dar Scott

Cool essay, Ken.

While we are at it I'll mention a couple details concerning arrays.

As Ken mention, the subscript is an expression that is viewed as a string.

Here is a gotcha for numeric subscripts. The subscripts in a[0.0+0.0] and a[0.0] do not point to the same entry, because the first use a subscript of "0" and the second uses a subscript of "0.0".

Another for array subscripts. Null cannot be a subscript.

There are some things to watch for in array values. Array elements cannot (yet!) be arrays. The result of arithmetic retains full precision when saved as an element (numberFormat is not applied at saving) and is not coerced to a string. When converted to a string later, numberFormat at that time is used.

Oh. There is one very positive thing. The best I can tell, you can put any string into an array element, including binary (arbitrary byte) values such as images. (The best I can tell, only keys cannot take arbitrary binary data.)

Posted 7/11/2004 by Dar Scott to the Use-Revolution List

Print this tip

News and Rumors Products Services Developer Resources Contact STS About STS