Working With Arrays

Doug Hennig

This month's column examines some best practices for working with arrays, including a library routine that does a better job of searching an array than ASCAN and one that cleans up by closing all tables opened by a routine.

Although it seems like arrays have been part of FoxPro forever, that isn't the case. In the "old" days, we used to fake arrays by creating a set of variables such as ARRAY1, ARRAY2, ARRAY3, etc. and then process them in sequence using macro expansion of a counter variable (we also had to walk 20 miles to get to work, too). Fortunately, real arrays were added to the language some time ago, along with an impressive array (pun intended) of functions that create, update, and process arrays. Every release of FoxPro has added additional array functions, and VFP is no exception. Some of the new functions, such as AUSED() and ADBOBJECTS(), allow you to process tables and databases objects much easier than before.

This month, we're going to take a look at some best practices for working with arrays. We'll look at some ideas for dimensioning arrays, searching arrays, and processing arrays.

Dimensioning Arrays

Although both DIMENSION and DECLARE can be used to dimension an array, I prefer to use DIMENSION. My main reason falls under the "you can teach an old dog new tricks, but you can't make him drink" category: DIMENSION has been in the language longer and I'm used to it. However, another reason is because in Visual FoxPro, DECLARE is overloaded; it's also used to define the DLLs that an application uses. In VFP, you can also dimension an array using the LOCAL or LOCAL ARRAY statements.

Frequently, you'll need to dimension an array multiple times. For example, you might be processing a table, adding one row to an array for each record you process. I learned the hard way that the following is the wrong way to do this:

```
dimension laArray[1, 3]
lnRows = 0
scan for <some condition>
    lnRows = lnRows + 1
    dimension laArray[lnRows, 3]
* store something in the array
endscan
```

What's wrong with this picture? Notice that laArray is dimensioned to 3 columns in two places. What if you need to change the number of columns later? In this simple example where the two statements are only a few lines apart, it might not be hard to remember to change both places, but what if the statements are in different subroutines or methods?

A better idea is to either store the number of columns in a variable or #DEFINE constant and use that variable or constant in the DIMENSION statement, or redimension the array using the current number of columns in the array as returned by the ALEN() function. The second DIMENSION statement in the code above would thus be better written as either of the following:

```
dimension laArray[lnRows, lnCols]
    && lnCols is previously initialized as 3
dimension laArray[lnRows, alen(laArray, 2)]
```

By the way, notice that I use square brackets rather than parentheses for the array dimensions. This makes it easy to distinguish between an array and a function. For example, it's obvious in the first line of code below that MyItems is an array, but is MyTables in the second line a function or an array?

```
lcItem = MyItems[lnRow]
lcTable = MyTable(lnTable)
```

Searching Arrays

ASCAN() is a function that's been with us for quite some time. ASCAN() searches an array for a particular value and returns the element number where the value was found, or 0 if it wasn't found. While ASCAN() is very fast, it has some shortcomings, especially regarding two-dimensional arrays:

- While ASCAN() can accept parameters specifying a range of elements to search, there's no way to tell it to just search a particular column in a two-dimensional array. If it's possible the value you're searching for could occur in more than one column but you're only interested in finding an instance in a particular column, too bad.
- ASCAN() respects the setting of EXACT, as it should. However, while I usually have EXACT set OFF so I don't need to specify an exact match for string comparisons, I find array searches normally need to be exact. This means saving the current setting of EXACT, setting it ON, using ASCAN(), then restoring the former setting. Forgetting to do all this usually results in something not working correctly (it's especially disastrous if you forget to set EXACT back again, since the symptom of that problem invariably occurs far from its actual source).
- As I mentioned above, ASCAN() returns the element number where the value was found. In the case of two-dimensional arrays (which I seem to work with more frequently than one-dimensional), it's usually the row number, not the element number you want, so you need an additional call to ASUBSCRIPT() to get the row number. Ah, but don't forget to check the return value from ASCAN() first; if it's 0 because the value wasn't found, ASUBSCRIPT() doesn't return 0 for the row, but instead actually blows up, scattering charred bits of your computer all over the desktop (perhaps that's an exaggeration).

Because of these limitations, I created a function called ArrayScan. This function works in FoxPro 2.x as well as VFP. It accepts four parameters:

- the array (passed by reference using @) to search
- the value to search for
- the column to search (optional: if it isn't specified, column 1 is searched)
- the occurrence to search for (optional: if it isn't specified, the first occurrence is located)

ArrayScan returns the row the value was found in if it was found, or 0 if it wasn't found.

Here's an example of using ArrayScan. Suppose you have an array containing information on a selected group of patients. Several columns in the array contain logical values, and you want to search for the occurrence of .T. in the third column only. You can't use ASCAN() since it would find .T. in any column. Here's what the code might look like:

```
lnRow = ArrayScan(@laPatients, .T., 3)
if lnRow > 0
    ? laPatients[lnRow, 1]    && display the name
endif lnRow > 0
```

Here's the code for ArrayScan (included in ARRYPROC.PRG on the source code disk). This function works in both FoxPro 2.x and VFP.

```
parameters taArray, ;
    tuValue, ;
    tnColumn, ;
    tnOccur
external array taArray
private lnColumn, ;
    lnOccur, ;
    lnRow, ;
    lnStartElement, ;
    lnFound, ;
    lnElement, ;
    lnCol
lnColumn       = iif(type('tnColumn') = 'N', ;
```

```
    tnColumn, 1)
lnOccur        = iif(type('tnOccur')  = 'N', ;
    tnOccur,  1)
lnRow          = 0
lnStartElement = 1
lnFound        = 0

* Use ASCAN to find the value in the array, then
* determine if it's in the correct column. If not,
* change the starting element number and try again.

do while .T.
    lnElement = ascan(taArray, tuValue, ;
        lnStartElement)
    if lnElement <> 0
        lnCol = asubscript(taArray, lnElement, 2)
        if lnCol = lnColumn and ;
            (type('tuValue') <> 'C' or ;
            taArray[lnElement] == tuValue)
            lnFound = lnFound + 1
            if lnFound = lnOccur
                lnRow = asubscript(taArray, lnElement, 1)
                exit
            endif lnCol = lnColumn ...
        endif lnCol = lnColumn ...
        lnStartElement = lnElement + 1
    else
        exit
    endif lnElement <> 0
enddo while .T.
return lnRow
```

Closing Opened Tables

One of the key principles in writing generic code is that you must clean up after yourself; if you changed an environment setting, you must change it back or other routines could stop functioning correctly since they didn't expect the setting to be changed. This includes tables; if your code needs to open a table, it should close the table before terminating. DataEnvironments in forms and reports make this a breeze, but for classes or procedural code which can't use DataEnvironments, you must handle it yourself. "What's the big deal?", you're probably wondering. "I'll just set a logical variable based on whether I opened the table or not. At the end of the routine, if the variable is .T., I'll close the table". This can be tedious if there are lots of tables involved, but that's not too much of a hassle. The gotcha occurs with views. If the generic routine opens a view, closing the view at the end of the routine doesn't completely clean up what the routine has done. After all, opening a view causes all the tables the view is based on to be opened as well. Not only do you have to check if the view is open at the start of the routine and close it if so at the end, you have to so the same for all the tables it's based on. Of course, you don't want to hard-code all those tables names (what if the view definition changes or is passed in as a parameter?), so that means using DBGETPROP() to get a list of tables involved in the view into a string, parsing the string, checking if each table is open, and then repeating the process at the end of the routine to close the tables. As well-known developer Steven Black often says in response to such a design, "Bleh".

Here's a better way. The VFP AUSED() function puts the alias of all open tables into an array. So, the idea is to use AUSED() at the start of the routine to store a "snapshot" of which tables are open before you do anything, then at the end of the routine compare the current set of open tables against that snapshot and close anything that wasn't open at the start. Because I do this type of thing fairly frequently, I created a routine called CloseOpenedTables that handles the work. CloseOpenedTables expects two parameters: an array (passed by reference using @) containing the starting "snapshot" and optionally the DataSessionID of the current form if it uses a private datasession.

Here's an example of using CloseOpenedTables:

```
local laUsed[1]
= aused(laUsed)
* do some stuff here which opens tables
= CloseOpenedTables(@laUsed)
```

Here's the code for CloseOpenedTables (included in ARRYPROC.PRG on the source code disk). Notice it uses ArrayScan to ensure a table alias is properly located in the array:

```
lparameters taTables, ;
    tnDataSession
local lnDataSession, ;
    laTablesOpen[1], ;
    lnTables, ;
    lnI, ;
    lcTable

* Ensure we were passed an array.

if type('taTables[1]') <> 'C'
    wait window 'taTables must be an array.'
    return .F.
endif type('taTables[1]') <> 'C'

* If the datasession was passed, save the current
* one and set the datasession to it.

if type('tnDataSession') = 'N' and ;
    tnDataSession > 0
    lnDataSession = set('DATASESSION')
    set datasession to tnDataSession
endif type('tnDataSession') = 'N' ...

* Get an array of currently open tables and go
* through them one at a time. If the table wasn't
* open before, close it.

lnTables = aused(laTablesOpen)
for lnI = 1 to lnTables
    lcTable = laTablesOpen[lnI, 1]
    if not empty(lcTable) and ;
        ArrayScan(@taTables, lcTable) = 0
        use in (lcTable)
    endif not empty(lcTable) ...
next lnI

* Restore the former datasession if necessary.

if type('tnDataSession') = 'N' and ;
    tnDataSession > 0
    set datasession to lnDataSession
endif type('tnDataSession') = 'N' ...
return
```

Stupid Array Tricks

Here are a bunch of little array ideas I use all the time:

- The Project Manager can tell the difference between an array and a function in a routine only if the array is defined in that routine. If you define an array and then pass it to a subroutine, the Project Manager will whine that it can't find a function named <array name> that's called from the subroutine. The EXTERNAL ARRAY <array name> command tells the Project Manager to stop complaining.
- While EXTERNAL ARRAY works with programs and objects, there's no way to do it with reports. If you create an array and then run a report that references the array, the Project Manager will complain that it can't find a function with the array name every time you build the project. The way around this problem is to define a function with no code using that name in some PRG file that'll never be called. I have a program called LIBRARY.PRG that just contains the names of arrays defined as functions, such as:

```
function laFields
```

```
function laTables
```

Since LIBRARY.PRG is included in the project but is never called from anywhere, it only serves to fool the Project Manager into thinking these arrays are defined as functions.

- You can't pass a array that's a property of an object to a function. The first statement below will only pass the first element in the array and the second will give an error:

```
= MyUDF(This.aArray)
= MyUDF(@This.aArray)
```

Instead, use ACOPY() to copy the array property to a local array, then pass the local array to the function. If necessary, such as if the function modifies the contents of the array, use ACOPY() to copy the local array back to the array property. Note that if the dimensions of the local array are changed, you need to redimension the array property before using the second ACOPY() or you'll get an error.

```
local laArray[1]
= acopy(This.aArray, laArray)
= MyUDF(@laArray)
dimension This.aItems[alen(laArray, 1), ;
    alen(laArray, 2)]
= acopy(laArray, This.aArray)
```

- Notice in the code above that laArray is declared local and is dimensioned to one element. Why bother with the "[1]" since the ACOPY() command will dimension laArray properly anyway? If you leave it out, VFP will give an "laArray is not an array" error. Unlike PRIVATE, which doesn't really create a variable but only "hides" any existing variable of the same name, LOCAL actually creates the variable. The variable is created as a logical unless you specify a dimension, in which case it's created as an array. Most of the VFP array functions will create an array if the variable doesn't exist, but will give an error if the variable does exist but isn't an array.
- Since I most frequently use arrays as the source of data for combo boxes and list boxes, I've added a custom property called aItems to my base combo box and list box classes and set their RowSourceType to 5-Array and RowSource to This.aItems. If a specific control created from these classes isn't based on an array, I can easily change RowSourceType and RowSource to the desired values, but the rest of the time I simply populate This.aItems as necessary in the Init() method of the control. This makes the control more self-contained than if it's tied to an array property of the form.

Conclusion

FoxPro has an impressive set of array functions, but sometimes you need more capabilities than are built in. Fortunately, routines like ArrayScan and CloseOpenedTables are easy to write and use whenever you need additional features.

Next month, we'll examine some ideas to put in your subclasses of Visual FoxPro base classes. Some of these ideas are straightforward, such as setting the BackStyle property of a Label subclass to Transparent so the background of the container shows through. Others may be less obvious, such as having the Error method of an object look up its containership hierarchy to find the first parent with error handling code. We'll look at some custom properties and methods you can add to provide more automatic functionality to your applications.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Data Dictionary for FoxPro 2.x and Stonefield Database Toolkit for Visual FoxPro. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America. CompuServe 75156,2326.