April, 2002

Hooray for Arrays!

With some VFP 7 enhancements, arrays are more useful than ever

I've been programming computers for a long time, and I've worked in a number of different programming languages. One feature offered by virtually every language I've used is arrays, ordered collections of elements that let you address a bunch of items by a single name.

FoxPro, of course, is no exception and, in the years I've been using it, the capabilities of FoxPro's arrays have been enhanced repeatedly. There are quite a few functions for managing arrays, lots of ways to move data between arrays and tables, and a whole bunch of functions that collect some information and put it into an array.

VFP 7 includes some key additions to array functionality, and several more "dump it into an array" functions. I'll take a look at both kinds of enhancements in the article to show you why arrays are more valuable than ever.

One exciting new capability is returning entire arrays from methods. Since I covered that in the March ADVISOR Answers column, I won't go over it again here.

Sorting and Searching

Sorting and searching within arrays are pretty fundamental operations. In fact, the first Fox article I ever published showed how to sort and search efficiently in FoxBase+ arrays. Then, in FoxPro 2, the ASort() and AScan() functions entered the language and we didn't have to worry about how to do these tasks ourselves.

However, until VFP 7, both of these functions had key weaknesses. Both were case-sensitive, making it difficult to work with character data. In addition, AScan() could search a subset of the array, but you couldn't limit it to a particular column. In VFP 7, these issues, along with a couple of others, are addressed and make ASort() and AScan() more useful than ever.

ASort() has only one change, so let's look at it first. There's one new parameter-a flag that indicates whether the sort is case-sensitive or case-insensitive. The complete syntax is:

```
nSuccess = ASORT( ArrayName [, nStart [, nElements
```

```
[, nOrder [, nFlag ] ] ] ] )
```

As in older versions, the nStart and nElements parameters indicate what part of the array is sorted. nStart serves double-duty here. Not only does it indicate the first element to include in the sort, but the array is sorted on the column that element resides in . nElements also has two interpretations; its meaning depends on whether the array is one-dimensional or two-dimensional. For a one-dimensional array, nElements indicates the number of elements (items) to be sorted, beginning with nStart. For a two-dimensional array, nElements is interpreted as the number of rows to sort, starting with the row that contains element number nStart.

nOrder, which determines whether the sort is ascending or descending, is pretty counter-intuitive. Omit it or pass a positive number to sort in descending order; pass 0 or a negative number for an ascending sort.

Finally, we come to the new parameter. It's actually quite simple. Omit the parameter or pass 0 for the same old case-sensitive sort. Pass 1 for a case-insensitive sort.

But what if you want to do a case-insensitive sort of the entire array? What do you do about the parameters between ArrayName and nFlag? While you can figure out the right values and pass them, there's an easier way. Pass –1 for any of those parameters (nStart, nElements or nOrder) and you get the default behavior for that parameter.

To set up some test data, we'll use another new feature of VFP 7. ADIR() now has an optional parameter that lets you retrieve file names in their original case. This line fills an array with information about the files in the Tools\Filer subdirectory of the VFP home directory:

```
nFileCount = ADIR( aFileList, HOME()+"Tools\Filer\*.*", ;
                   "", 1)
```

In my installation, that directory has only 4 files, and the first column of the array contains the following:

```
filer.dll
Filer.ico
filer.sct
filer.scx
```

(By coincidence, these files are already sorted as we want them, but in an application, we can't count on that.)

If we sort without the nFlag parameter, like this:

```
ASORT( aFileList )
```

the file "Filer.ico" comes to the top of the list because of the capital "F". To get the desired results, add the new parameter, like this:

```
ASORT( aFileList, -1, -1, -1, 1 )
```

AScan(), for searching within arrays, has two new parameters, but they add a whole bunch of new capabilities. Here's the updated syntax:

```
nResult = ASCAN( ArrayName, uSearch [, nStart [, nElements
                 [, nColumn [, nFlags ] ] ] ] )
```

The second parameter, uSearch, is the item you're searching for. As with ASort(), nStart and nElements determine which portion of the array is searched; you can pass –1 to indicate the default values.

The new nColumn parameter provides the ability to search within a single column. If you want to search only in the name column of the aFileList array created above, you can use code like this, adding the column number to the parameter list:

```
cFileToFind = "filer.scx"
nFoundWhere = ASCAN( aFileList, cFileToFind, -1, -1, 1 )
```

The nFlags parameter is, as its name suggests, an additive, bit-oriented flag. It provides three new capabilities: case-insensitive searching, control over exact searching, and returning the row rather than the element number.

The case-sensitivity flag uses bit 0 (the right-most bit), so you add 0 to keep the default case-sensitive search or 1 for case-insensitive search.

The two bits to the left of the first flag are involved in the process of determining EXACT matches. By default, AScan() follows the SET EXACT setting in effect. The new flags let you decide whether to do that or to control EXACTness locally, just for this function call. Bit 2 determines whether we're controlling EXACTness locally – add 0 to leave SET EXACT in charge, or 4 to control it locally. Bit 1 indicates which EXACTness setting we want to use locally – add 0 for EXACT OFF or 2 for SET EXACT ON. The setting of bit 1 matters only when bit 2 is set, that is, when you add 4 to the flag.

By default and in all earlier versions of VFP, AScan() returns the element number of the found item. In a two-dimensional array, you

usually want to know which row the item is in, rather than its actual element number. Often, you'll want to look up another item in the same row. With AScan()'s new ability to search in a particular column, this use of the function is likely to be even more common. Moving left again, bit 3 of nFlags determines what the function returns – add 0 for the element number or 8 for the row number.

All in all, there are 16 possible values for nFlags. (Because of the interaction between the two bits used for EXACTness, there are actually 12 different functionalities.) Table 1 shows the various options.

Table 1. Fine-tuning AScan(). The nFlags parameter makes AScan() do what you want.

nFlags Value	**Meaning**
0 or 2	Case-sensitive, use SET EXACT setting, return element number
1 or 3	Case-insensitive, use SET EXACT setting, return element number
4	Case-sensitive, EXACT OFF, return element number
5	Case-insensitive, EXACT OFF, return element number
6	Case-sensitive, EXACT ON, return element number
7	Case-insensitive, EXACT ON, return element number
8 or 10	Case-sensitive, use SET EXACT setting, return row number
9 or 11	Case-insensitive, use SET EXACT setting, return row number
12	Case-sensitive, EXACT OFF, return row number
13	Case-insensitive, EXACT OFF, return row number
14	Case-sensitive, EXACT ON, return row number
15	Case-insensitive, EXACT ON, return row number

For example, if you want to find the row that contains the exact filename "Filer.SCX", but without worrying about case, you'd use this code:

```
nMatchRow = ASCAN( aFileList, "Filer.SCX", -1, -1, 1, 15)
```

On the other hand, if you want to find the row containing the first item in the array that has "filer" in its name, regardless of case, use:

```
nMatchRow = ASCAN( aFileList, "filer", -1, -1, 1, 13)
```

The new capabilities of AScan() make a lot of tasks easier and cleaner. There are a couple of more detailed examples later in this article.

Tell Me About Yourself

VFP has a large group of functions that collect some information and put it into an array. In VFP 7, the group is even larger with six new "A" functions. Table 2 shows the new functions and explains each one's purpose.

Table 2. Fill 'er up–These new functions each fill an array with specified data.

Function	**Resulting array contains a list of:**
ADLLs()	API functions currently declared.
ALanguage()	Elements of the VFP programming language.
AProcInfo()	The components of a specified program file.
ASessions()	Active data sessions.
AStackInfo()	The current call stack.
ATagInfo()	Index tag information for a specified table.

Like the other functions of this type, all of these return the number of rows (for two-dimensional arrays) or items (for one-dimensional arrays) in the result.

Tracking API Functions

ADLLs() takes a single parameter–the array to fill. The resulting array has one row for each currently declared API function. There are three

columns in the array: the name of the API function, the alias with which it was declared, and the path and filename of the containing library.

The new features of AScan() combine nicely with ADLLs() to write a new function that returns the alias with which a particular function was declared, or the empty string if that function isn't currently declared. Here's the function, included on this month's Professional Resource CD as DLLAlias.PRG:

```
*=================================================
* Program:            DLLALIAS.PRG
* Purpose:            Return the alias with which a
*                     function was declared.
* Author:             Tamar E. Granor
* Copyright:          (c) 2001, Tamar E. Granor
* Last revision:      12/18/01
* Parameters:         Name of the function to look up
* Returns:            The alias with which the function
*                     was declared; the empty string,
*                     if the parameter is no good
*                     or the function isn't declared
*=================================================
LPARAMETERS cFunction

ASSERT VARTYPE(cFunction) = "C" AND NOT EMPTY(cFunction) ;
  MESSAGE "DLLAlias: Must pass function name"

IF VARTYPE(cFunction) <> "C" OR EMPTY(cFunction)
  RETURN ""
ENDIF

LOCAL nDLLCount, aDLLList[1], nItem, cAlias

nDLLCount = ADLLS( aDLLList )
nItem = ASCAN( aDLLList, cFunction, -1, -1, 1, 14)
IF nItem = 0
  cAlias = ""
ELSE
  * Grab the defined alias for the function
  cAlias = aDLLList[ nItem, 2 ]
ENDIF

RETURN cAlias
```

To call DLLAlias(), pass the name of the API function you're interested in. Keep in mind that API functions are case-sensitive, so the search performed in DLLAlias() is also case-sensitive. Here's an example call:

```
* Most likely, you'd have a line like this earlier
* in your code:
DECLARE INTEGER GetSysColor IN Win32API INTEGER nIndex
```

```
* Then, to determine the alias:
cGSCAlias = DLLAlias( "GetSysColor" )
```

Listing Language Elements

ALanguage() extracts various kinds of language elements and puts them into an array. The function takes two parameters: the array to fill and a number indicating which type of language elements to find. Table 3 shows the possible values of the second parameter and the results.

Table 3. What's in a Language?–ALanguage() puts various language elements into an array.

Second parameter	Result
1	One-dimensional array with a command in each element
2	Two-dimensional array with one row per function. The first column contains the function name. The second column has three pieces of information, in the form "M9-9". If the letter "M" is present, the function name can't be abbreviated. The first number indicates how many parameters the function requires. The second number is the total number of parameters the function accepts. For example, for ACOPY(), the second column contains the string "2-5".
3	One-dimensional array with a base class in each element.
4	One-dimensional array with a DBC event in each element.

I haven't found a use for this function yet.

Auditing Program Files

AProcInfo() is one of several new functions related to the Document View and Task List tools. It accepts two or three parameters: the array to populate, the name of the file to examine, and optionally, a number

indicating what information to extract from the file. Table 4 shows the possible values for the third parameter, and the results in each case.

Table 4. What's in a program file?–AProcInfo() lets you find various components of .PRG files.

Third parameter	Number of columns in results	Contents of array
0 or omitted	4	Name, line number, type of item and indentation for each procedure/function, class, method or compiler directive in the file.
1	4	Class name, line number, parent class name, and OLE public status for each class in the file.
2	2	Name (in the form "class.method") and line number for each method in the file.
3	3	Name, line number and type of item for each compiler directive in the file.

Although Document View works with all kinds of code, whether it's in program files, class libraries, forms or stored procedures, AProcInfo() works only for program files (.PRG). The information it provides gives you a good picture of the file contents, and is sufficient to use the new EditSource() function to open the file to a particular location.

You might use the data collected by AProcInfo() in a documentation utility. As a starter, you could use ASort() to put the items in alphabetical order like this.

```
* cFile = file name to audit
nItems = APROCINFO( aFileItems, cFile )
* Sort on name, case-insensitive
IF nItems > 0
   ASORT( aFileItems, 1, -1, 0, 1 )
   * More processing
ENDIF
```

Your code might then dump the data into a table or cursor, or perhaps process it one at a time, sending different item types into different tables.

Listing Data Sessions

Certain situations call for code that goes through each active data session, and processes the tables open in that session. For example, when an error occurs that's serious enough to shut down the application, you want to minimize the loss of data. There are also times when you want to close a table everywhere it's open and be able to reopen it. ASessions() makes both of these scenarios possible.

At first glance, the function results seem trivial. There's one item in the array for each active data session, containing the data session number. In simple cases, that means that element 1 contains 1, element 2 contains 2, and so forth. However, the way data session numbers are handed out means the results of ASessions() aren't always just numbers in order.

Data session numbers are assigned in ascending order as data sessions are opened. However, when a data session is closed, other active sessions are not renumbered. (Good thing.) That means there can be holes in the sequence. (In addition, available numbers are re-used, so if data session number 3 is closed, the next new data session created gets the number 3.)

Since there can be arbitrarily many active data sessions, looping through all possible data session numbers doesn't work, and the possibility of holes in the sequence means that you can't just loop until you find an unused data session number. ASessions() solves both these problems. You're most likely to use it in code that looks something like this:

```
LOCAL nSessions, nWorkAreas, aSessionList[1], aCursors[1]
LOCAL nSession, nOldSession, nCursor, nOldWorkArea

* Get a list of sessions
nSessions = ASESSIONS( aSessionList )
nOldSession = SET("DATASESSION")

FOR nSession = 1 TO nSessions
   SET DATASESSION TO nSession

   * Get a list of open tables
   nWorkAreas = AUSED( aCursors )
   nOldWorkArea = SELECT()
   FOR nCursor = 1 TO nWorkAreas
      * Do something to the table in this work area
   ENDFOR
   SELECT (nOldWorkArea)
ENDFOR
```

```
SET DATASESSION TO nOldSession
```

The "something" in the inner loop might be reverting changes and closing the table, checking whether the work area contains a particular table and doing something in that case, and so forth.

Tracking the Program Stack

AStackInfo() is primarily a debugging aid. It provides programmatic access to information that's available in the Debugger. The array created by AStackInfo() has six columns, as shown in Table 5.

Table 5. What's running?–AStackInfo() fills an array with the program stack, containing this information for each active routine.

Column	Contents
1	The stack level.
2	The name of the file executing at this level.
3	The name of the module or object executing at this level.
4	The name of the source code files for the code executing at this level.
5	The line number executing at this level.
6	The line of code executing at this level.

The most obvious use for this function is in error handlers, where you can collect the call stack data and store it in an error log. The task of figuring out what went wrong is much easier with complete stack information. You might do something like this:

```
* Collect stack info
LOCAL aProgStack[1], nLevels, nLevel, cStackInfo
nLevels = ASTACKINFO( aProgStack)
cStackInfo = ""
FOR nLevel = 1 TO nLevels
   cStackInfo = cStackInfo + "Stack level " + ;
                     TRANSFORM( aProgStack[nLevel, 1]) ;
                     + ": "
   cStackInfo = cStackInfo + aProgStack[nLevel, 2] + ", "
   cStackInfo = cStackInfo + aProgStack[nLevel, 3] + ", "
   cStackInfo = cStackInfo + aProgStack[nLevel, 4] + ", "
```

```
   cStackInfo = cStackInfo + "Line " + ;
                       TRANSFORM( aProgStack[nLevel, 5] ) ;
                       + ": "
   cStackInfo = cStackInfo + aProgStack[nLevel, 6] + CHR(13)
ENDFOR
```

Then, you can store the accumulated string into a memo field in the error log.

AStackInfo() does have one problem. When you call it from a form or class within an .APP or .EXE, the second column contains the name of the .SCT or .VCT for the form or class. It should contain the .APP or .EXE name.

Collecting index information

The final new "A" function, ATagInfo() puts all the data you need to recreate the index tags for a table into an array. The array has 6 columns, as shown in Table 6.

Table 6. What's in a tag?–The array created by ATagInfo() contains the information you need to recreate index tags.

Column	Contents
1	Tag (or IDX) name
2	Tag type ("PRIMARY", "CANDIDATE", "REGULAR", "UNIQUE")
3	Key expression
4	Filter expression
5	"ASCENDING" or "DESCENDING"
6	Collate sequence

In addition to storing metadata to allow tags to be recreated, ATagInfo() also makes it much easier to determine whether a tag with a particular key exists. In earlier versions of VFP, this task requires brute force. With ATagInfo() and the new parameters to AScan(), it's a breeze. This function is available on this month's PRD as KeyExists.PRG.

```
*=========================================================
* Program:           KEYEXISTS.PRG
* Purpose:           Determine whether a table has a
*                    specified key expression
* Author:            Tamar E. Granor
* Copyright:         (c) 2001, Tamar E. Granor
* Last revision:     12/20/01
* Parameters:
*    cKeyExpr = the key expression to look for
*    cIndexFile = the index file to search in (optional)
*    uWhatTable = the alias or work area to search
*                 in (optional)
* Returns:          .T., if the key exists;
*                   .F., if parameters are bad
*                        or the key doesn't exist
*=========================================================
LPARAMETERS cKeyExpr, cIndexFile, uWhatTable

LOCAL ARRAY aTagList[1]
LOCAL nTagCount, nResult

* Check key expression parameter
IF VARTYPE( cKeyExpr ) <> "C" OR EMPTY(cKeyExpr)
   ERROR 11
   RETURN .F.
ENDIF

* Check index file parameter
DO CASE
CASE VARTYPE( cIndexFile ) = "L" AND NOT cIndexFile
   * default to all open indexes
   cIndexFile = ""
CASE VARTYPE( cIndexFile ) <> "C"
   ERROR 11
   RETURN .F.
OTHERWISE
   * all is well. No change needed
ENDCASE

* Check table parameter
DO CASE
CASE VARTYPE( uWhatTable ) = "L" AND NOT uWhatTable
   * default to current table, if anything is open
   IF NOT EMPTY(ALIAS())
      uWhatTable = ALIAS()
   ELSE
      ERROR 52
      RETURN .F.
   ENDIF
CASE VARTYPE( uWhatTable ) = "C"
   * check that the specified alias is in use
   IF NOT USED( uWhatTable )
      ERROR 13
      RETURN .F.
   ENDIF
```

```
CASE VARTYPE( uWhatTable ) = "N"
   * check that the specified workarea is in use
   IF NOT USED( uWhatTable )
      ERROR 52
      RETURN .F.
   ENDIF
OTHERWISE
   * Not a valid type
   ERROR 11
   RETURN .F.
ENDCASE

* If we get this far, we have good parameters
* So get a list of tags
nTagCount = ATAGINFO( aTagList, cIndexFile, uWhatTable )

* Now search for specified expression
IF nTagCount > 0
   nResult = ASCAN( aTagList, NORMALIZE(cKeyExpr), -1, -1, 3, 6 )
ELSE
   * There was no such index file as specified
   nResult = 0
ENDIF

RETURN nResult > 0
```

You might want to modify the function to return the name of the tag that has the specified key rather than just a logical value indicating whether it exists.

Tell Me *More* About Yourself

In addition to all the new "A" functions, several existing functions that fill arrays have been enhanced in VFP 7. The details are beyond the scope of this article, but be sure to check Help for ADir(), ALines(), AMembers(), and ANetResources() to see how they've been improved.

Better and Better

VFP 7 continues the trend that started in FoxPro 2.0 of making arrays more and more useful in each version of VFP. If you haven't worked much with arrays, spend some time with the relevant Help topics or take a look at Miriam Liskin's article in the April '98 issue and I'm sure you'll see a number of ways you can take advantage of them in your applications.