Session FT-
You Need Arrays

Tamar E. Granor, Ph.D.
Email: tamar_granor@compuserve.com

Overview

Visual FoxPro's arrays are amazingly useful and powerful. This session will examine the functions that let you manipulate arrays, the many functions that use arrays to provide information about both VFP and the broader environment in which an application is operating, and explore the many ways arrays can be used in application development and in building developer tools. It will also address the performance of arrays and look at issues in using arrays in COM development. Special attention will be paid to changes in VFP 7 that affect arrays. This session assumes familiarity with VFP fundamentals.

Array Basics

An *array* is an organized collection of data. It lets you call a number of variables by a single name. Each item in the array, called an *element*, is identified by its position. While some programming languages require all elements of an array to be of the same type, in FoxPro, there is no such rule. Every array element can be of a different type.

Individual elements of an array are referenced by following the name of the array with either square brackets or parentheses, enclosing the position of the element. (These notes use the square bracket notation.) Visual FoxPro (and FoxPro before it) offers one-dimensional (1-D) and two-dimensional (2-D) arrays. To specify an element of a 2-D array, the row and column positions are separated with a comma.

Internally, all FoxPro arrays are represented as one-dimensional, which means that you can access the elements of a 2-D array as if it were 1-D. Two-dimensional arrays are stored in *row-major order*. That is, all the elements from the first row are stored before the elements from the second row, and so forth. (The alternative to row-major order, used by some programming languages, is *column-major order*, in which all the elements of the first column are stored before all the elements of the second column.) So, when you create a 6x3 array in VFP, it can be seen as a one-dimensional array with 18 elements. The first 3 are those from row 1, then the next three are those from row 2, and so on.

Internally, a 1-D array is seen as a single row of elements, so while you can access the elements as if it were 2-D without error, it's not terribly meaningful. In fact, it can be confusing, because changing the row index doesn't affect the result. VFP only looks at the column index in this case.

Defining arrays

Several commands let you create arrays. The identical DIMENSION and DECLARE commands are designed specifically for array definition. They create the new array as a private variable (except when used from the Command Window, where they create public variables). For example:

```
DIMENSION aTest1D[3], aTest2D[7,5]
```

creates two arrays. aTest1D is a one-dimensional array with 3 elements, while aTest2D is a two-dimensional array with 7 rows and 5 columns.

To create local arrays, use the LOCAL keyword, but specify the dimensions of the array. There's an optional ARRAY keyword for those who prefer to be explicit. Without the ARRAY keyword, array and scalar (non-array) variables can be created with a single LOCAL statement. When the ARRAY keyword is specified, only arrays can be defined with that LOCAL statement. For example, the following commands are all valid:

```
LOCAL ARRAY aLocal[3]
LOCAL cName, aLocal2D[5,2]
LOCAL aLocal1D[25], nValue
```

but these are not:

```
LOCAL dToday, ARRAY aNoGo[17]
LOCAL ARRAY aNoGo[17], dToday
```

The PUBLIC command, not surprisingly, creates public variables, and like LOCAL, can be used to define arrays. It has the same optional ARRAY keyword and the same restrictions on its use. Think twice, however, about creating any public variables, arrays included. Public variables can lead to hard-to-find errors. In general, the best idea is to make all variables local unless you have a specific need for a larger scope.

Arrays are limited to 65,000 elements. The number of elements in a two-dimensional array is computed by multiplying rows by columns. When you attempt to create an array with more than 65,000 elements, you get the error message "Too many variables."

Creating array properties

When you need an array to be available to multiple methods of an object (whether it's a form, a control or some other kind of object), the best solution is to make the array a property of that object. Defining array properties is simple, once you know how.

In coded classes, to add an array property, use DIMENSION or DECLARE in the definition portion of the class. For example, this (fairly useless) class has a three-element array and a method to report the type of the specified element of the array's:

```
DEFINE CLASS ArrayDemo AS Custom

DIMENSION aMyArrayProperty[3]

PROCEDURE ArrayElementType
LPARAMETERS nElement

* Report the element's type
WAIT WINDOW TYPE("This.aMyArrayProperty[nElement]")
RETURN

ENDDEFINE
```

In the Class Designer or Form Designer, use the New Property dialog to add an array property. The secret is to specify the dimensions for the array in the dialog. Figure 1 shows the creation of an array property in the New Property dialog.

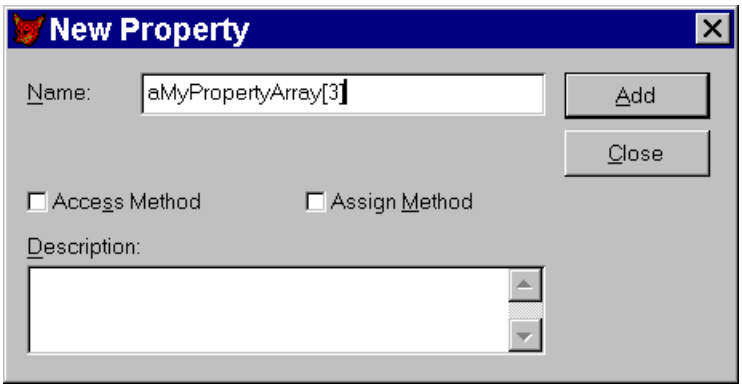

Figure 1 Adding array properties – The New Property dialog lets you add array properties like any other, but you must specify the dimensions of the array.

Redefining arrays

FoxPro allows you to redefine an array once it's been created. Use DIMENSION or DECLARE and specify the new size of the array. Any data already in the array remains in the same element. If you change a 1-D array to a 2-D array, the existing data fills the cells in row-major order. This makes sense if you remember that internally, all arrays are one-dimensional.

If you resize an array so that it's larger than it was before, the new elements are added at the end. The new elements have an initial value of .F. Similarly, if you resize an array so that it's smaller than it was before, the extra elements are removed from the end.

Here are some examples:

```
LOCAL aExample[ 17 ]  && results in a 17-element array
DIMENSION aExample[6, 3] && adds an element at the end and makes the array 2-D
DIMENSION aExample[6, 2] && reduces the array to 12 elements. The last 6 (the
                         && last two rows in the previous arrangement) are
                         && removed and the remaining elements are
                         && redistributed to fill 2 columns.
```

Assigning array data

Array elements are accessed like any other variable, except that you must specify the particular element you want. So, for example, to save "This is a test" to the third element of aTest, use:

```
aTest[3] = "This is a test"
```

It's also possible to store the same value to all elements of an array with one command. To set the entire array aTest to "This is a test", use:

```
aTest = "This is a test"
```

However, there's a trap lurking here. Setting all elements of an array with a single assignment statement works only if SET COMPATIBLE is OFF or FOXPLUS (which are identical settings). If COMPATIBLE is set to ON or DB4, a statement like the last example destroys the array and creates a new variable with the same name and scope, assigning it the specified value. (Note that this trap affects only variables, not array properties.)

While it's easy enough to avoid changing SET COMPATIBLE programmatically, there's one further trap. In the General page of the Options dialog (figure 2), there's an innocuous looking checkbox labeled "dBASE compatibility". Checking this box is the same as issuing SET COMPATIBLE ON.

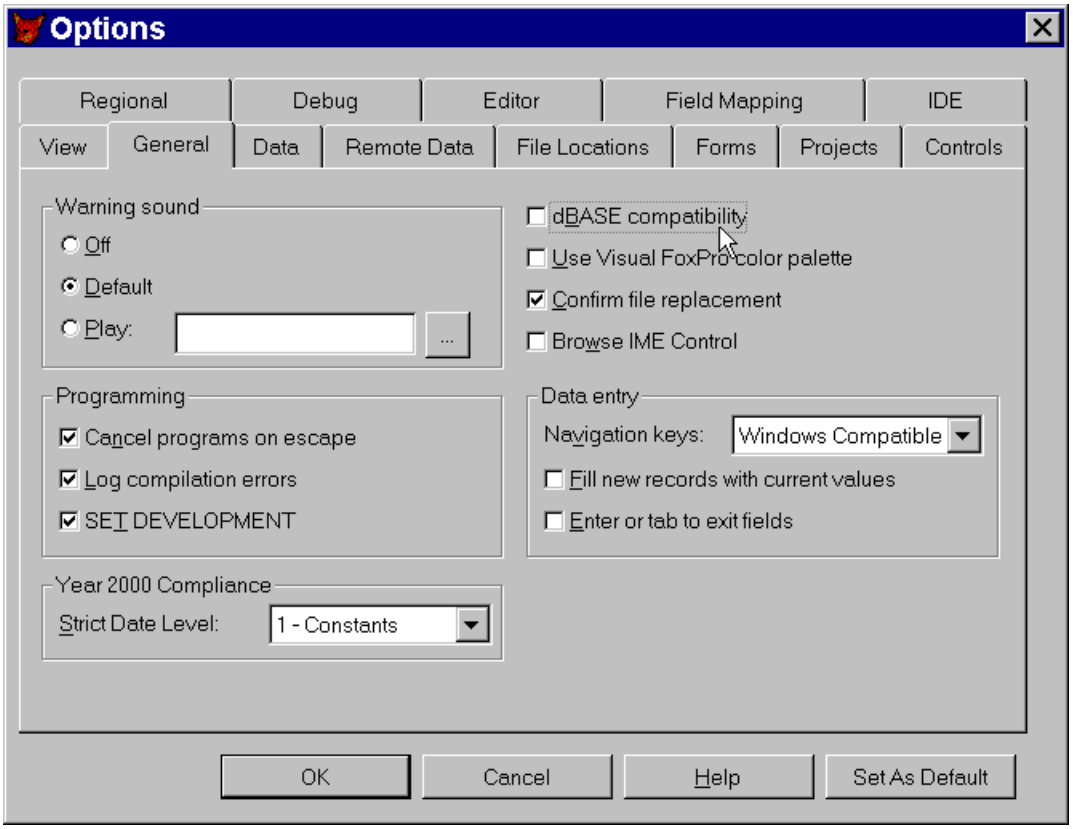

Figure 2 Avoiding SET COMPATIBLE ON – The selected checkbox in this page of the VFP's Options dialog changes the setting of COMPATIBLE, which affects assignment statements to entire arrays (and a host of other things).

There's one further consideration when working with array properties. As the ArraySize method in the *Creating Array Properties* section shows, to refer to an array property, you need to provide the appropriate object reference. In a method of the class to which the property belongs, using THIS is sufficient.

Checking for an array

To check whether a particular variable is an array, use the TYPE() function and examine the first element. If the variable is an array, TYPE() returns the type of that element. If the variable is not an array, TYPE() returns "U" for "unknown." For example:

```
LOCAL aArray[4]
? TYPE("aArray[1]")  && returns L because newly created elements contain .F.
? TYPE("cScalar[1]") && returns "U" because cScalar is not an array
```

Moving data between arrays and tables

Several commands are designed to provide quick ways of transferring data from tables to arrays and back. (In this context, "table" can usually mean a table, view or cursor.)

Starting with the most commonly used of these command, arrays are one of the possible destinations for the SQL SELECT command. If the specified array doesn't exist, it's created (with public scope if you're working in the Command Window or with private scope in a program). If the array already exists, it's redimensioned and retains its current scope. This example puts the name and birth date of all employees located in the UK into an array. (The table is drawn from Visual FoxPro's example TasTrade database.)

```
SELECT first_name, last_name, birth_date ;
   FROM _SAMPLES+"TasTrade\Data\Employee" ;
   WHERE country = "UK" ;
   INTO ARRAY aUKEmps
```

The newly created array has 5 rows and 3 columns.

SCATTER and GATHER let you copy data from a single record to an array and back to the record. To use an array as the destination for the SCATTER command, add the TO keyword and the name of the array. Similarly, to collect data from an array, add the FROM keyword to GATHER. This example retrieves the first record from the Employee table, changes a few things, then restores the array data to the record

```
USE _SAMPLES+"TasTrade\Data\Employee"
SCATTER TO aOneRecord
aOneRecord[2] = "Jones-Buchanan"
aOneRecord[4] = "Sales Manager Extraordinaire"
GATHER FROM aOneRecord
USE
```

As the example shows, working with record data this way is complicated because you need to know which element of the array represents which field in the table. In FoxPro 2.x, SCATTER and GATHER were commonly used to provide data buffering.

The other pair of commands that move data between tables and arrays is COPY TO ARRAY and APPEND FROM ARRAY. Unlike SCATTER and GATHER, these commands can move data from more than one record at a time. Whether they do or not depends on the array definition. When the specified array is one-dimensional, a single record is transferred. When the array is two-dimensional, one record is moved for each row of the array.

Both commands (like SCATTER and GATHER) let you specify which fields are involved in the data transfer. You can provide either an explicit field list using the FIELDS clause or use the LIKE and EXCEPT clauses to specify a field skeleton. COPY TO ARRAY also lets you limit the records copied using the usual Xbase Scope, FOR and WHILE clauses.

This example performs exactly the same operation as the SQL SELECT above. It copies the name and birth date of all UK employees into an array:

```
USE _SAMPLES+"TasTrade\Data\Employee"
COPY TO ARRAY aUKEmps ;
   FIELDS First_Name, Last_Name, Birth_Date ;
   FOR Country = "UK"
```

Although SQL SELECT can do everything that COPY TO ARRAY can, COPY TO ARRAY is as much as an order of magnitude faster with small data sets. However, the time to execute

COPY TO ARRAY appears to increase with the size of the table, as well as with the size of the result set.

There's also a problem using COPY TO ARRAY with large data sets. If the data set is larger than the 65,000-element limit for arrays, you must define the array before the COPY TO ARRAY. Otherwise, the command fails. However, when you use COPY TO ARRAY with an array that already exists, the array is not redimensioned. The command fills only the number of elements provided. So, to successfully use COPY TO ARRAY with a large data set, you must COUNT the number of matching records first and dimension the array appropriately.

There's also an important difference between COPY TO ARRAY and SELECT INTO ARRAY when you're working with buffered data. SELECT draws its results from the original tables, while COPY TO uses the buffers. When you need to create an array based on buffered data, COPY TO is the way to go.

Finally, it's worth noting that SELECTing into a cursor rather than an array is faster than either of the methods of creating an array. With large result sets (my test extracted about 20,000 records from a table containing more than a million), SELECTing into a cursor is an order of magnitude faster. So, unless an array is absolutely necessary, when dealing with large record sets, use a cursor instead.

APPEND FROM ARRAY also supports the FOR clause. Each array element is evaluated as if it had already been added to the table. If the condition is true, the record is added; if the condition is false, the record is not added. This example takes the data in aUKEmps and copies it into a cursor:

```
CREATE CURSOR UKEmps (cFirst C(10), cLast C(20), dBirthDate D)
APPEND FROM ARRAY aUKEmps
```

Of course, this example could be accomplished in a single SQL SELECT. The only difference is that the cursor created here is read-write and, by default, a cursor created by SQL SELECT is read-only. However, in VFP 7, that's no longer an issue because of the addition of a READWRITE clause to SQL SELECT.

However, the approach of creating a cursor and using APPEND FROM ARRAY is very handy when dealing with the various functions that examine the environment and put their results into an array. You can take the results and move them to a cursor, where you can then use all of VFP's processing power.

Finally, there's the array version of INSERT INTO. This command can add one or many records stored in an array to a table. If the array is one-dimensional, the command adds a single record. If it's two-dimensional, the command adds one record for each row of the array. The syntax is:

```
INSERT INTO Table FROM ARRAY Array
```

Like APPEND FROM ARRAY, this command can be combined with any of the ones that copy data from an array to move records from one table to another. Here's an example using SQL SELECT:

```
SELECT first_name, last_name, birth_date ;
   FROM _SAMPLES+"TasTrade\Data\Employee" ;
```

```
   WHERE country = "UK" ;
   INTO ARRAY aUKEmps

CREATE CURSOR EmpNames (cFirst C(10), cLast C(20), dBirthdate D)
INSERT INTO EmpNames FROM ARRAY aUKEmps
```

As with APPEND FROM ARRAY, you're more likely to do things this way when you need to manipulate the records before moving them, or when you're copying from one table to an existing table, rather than to a newly-created cursor. INSERT INTO FROM ARRAY is also handy for moving the information in arrays created by the "A" functions discussed below into cursors or tables.

There is one big difference between APPEND FROM ARRAY and INSERT INTO FROM ARRAY. INSERT INTO can populate a memo field, while APPEND FROM cannot.

Moving data from strings to arrays

The ALINES() function, added in VFP 6, lets you move character strings into arrays, breaking them up into lines, so that each array element contains one line of the original string. In VFP 7, ALINES() has been enhanced to let you decide what characters indicate the end of a line.

The syntax for ALINES() is:

```
nLines = ALINES( Array, cString, [, lTrim ] [, cSeparator, … , cSeparator ] )
```

The function returns the number of lines in the resulting array. As is generally true for FoxPro functions that put something into an array, if the array already exists, it's resized. If the array doesn't exist, it's created.

If you pass only an array name and a string (which can be a memo field), CHR(13), CHR(10) or a combination of the two are considered to mark the end of a line. The line separator is not placed in the array.

The optional lTrimIt parameter lets you strip both leading and trailing blanks from each line.

The cSeparator parameter(s), added in VFP 7, let you expand the horizons of this function. You can list one or more characters that indicate the end of a line. Even when you specify your own line separators, the default CHR(13) and CHR(10) are still interpreted as ending a line. So the characters you specify are added to the list. As in the default case, the separator character that ends a line is not put into the array.

One handy way to use ALINES() is to break up delimited lists. For example, given a list like "red, orange, yellow, green, blue, purple", you can put each color in an array element. In VFP 7, use this code:

```
cColorList = "red, orange, yellow, green, blue, purple"
nColorCount = ALINES( aColors, cColorList, .T., ",")
```

In VFP 6, it's necessary to change the commas into one of the accepted separators before calling ALINES():

```
cColorList = "red, orange, yellow, green, blue, purple"
nColorCount = ALINES( aColors, STRTRAN(cColorList,",",CHR(13)), .T.)
```

ALINES() has one unusual feature. You can omit the lTrim parameter and begin specifying separators with the third parameter. For example, the RowSource of a combo with RowSourceType = 1 is a comma-separated list. Any blanks that occur are part of the item. To parse such a list, you can write:

```
nItems = ALINES( aItems, This.RowSource, "," )
```

Passing arrays as parameters

Arrays must be passed to functions and procedures by reference. When an array is passed by value, the routine receives only the first element.

Since the default for procedures is to receive parameters by reference, no special action is needed in that case. That is, when a routine is called using the DO command, all parameters are passed by reference, including arrays.

By default, parameters to functions are passed by value. (Although the default can be changed with SET UDFPARMS, that's not a good idea.) To pass a function parameter by reference, precede it with "@".

This program demonstrates the various options:

```
* Demonstrate the ways of passing an array
LOCAL aArray[3]
LOCAL cResult

WAIT WINDOW "Calling routine as procedure (using DO)"
DO PassArray WITH aArray, cResult
WAIT WINDOW cResult

WAIT WINDOW "Calling routine as function without @"
cResult = PassArray(aArray)
WAIT WINDOW cResult

WAIT WINDOW "Calling routine as function with @"
cResult = PassArray(@aArray)
WAIT WINDOW cResult

RETURN

PROCEDURE PassArray
* This procedure receives an array parameter.

LPARAMETERS aParm, cReturn

IF TYPE("aParm[1]") = "U"
   cReturn = "Not an array"
ELSE
   cReturn = "Array"
ENDIF

RETURN cReturn
```

Array properties cannot be passed as parameters, except to built-in functions. To pass an array, say from one method to another, you need to use ACOPY() (discussed in "Copying Arrays"

below) to copy the data to a local array, then pass that array to the other method. If you need changes made by the other method to be restored to the array property, use ACOPY() again after the method call. Here's the structure:

```
PROCEDURE MyMethod

* whatever happens before the call to the other method

ACOPY( This.aArrayProperty, aPlaceHolder )
This.MyOtherMethod( @aPlaceHolder )
ACOPY( aPlaceHolder, This.aArrayProperty)

* whatever happens after the call to the other method

ENDPROC
```

Passing arrays to COM components raises some special issues. See the section "Arrays and COM" for the details.

Returning array values

In VFP 6 and earlier versions, there's no way to directly return an array from a function. VFP 7 adds this ability, provided the array is in scope after the function returns. In practice, this means you can return an array from a method if the array is a property of that method's object. (You can also return a non-property array if you declare it sufficiently high in the scope chain. However, doing so violates good programming practices since the function that returns the array must reference a variable that it didn't create or receive as a parameter.)

To return an array, you must precede it with "@" in the return statement, as if it were a parameter.

This code demonstrates returning an array. The RGBComp method of the class receives an RGB value as a parameter and breaks it down into its components, storing them in an array, which it returns. This function does the same thing as the RGBComp() function in FoxTools, but that function handles the need to return three values by accepting them as parameters by reference.

```
#DEFINE CR CHR(13)

LOCAL nColor, aRGBColor[3], oRGB AS Colors

* Get a color
nColor = GETCOLOR()

oRGB=CREATEOBJECT("Colors")
aRGBColor = oRGB.RGBComp(nColor)

MESSAGEBOX("The color is " + TRANSFORM(nColor) + "." + CR + ;
           "Red: " + TRANSFORM(aRGBColor[1]) + CR + ;
           "Green: " + TRANSFORM(aRGBColor[2]) + CR + ;
           "Blue: " + TRANSFORM(aRGBColor[3]), ;
           "Show array return value", 64)

RETURN
```

```
DEFINE CLASS Colors AS Custom
* Color handling code
DIMENSION aRGB[3]

FUNCTION RGBComp(nColor) AS Array
* RGBComp
* Returns the Red, Green and Blue Components
* of a color in an array

This.aRGB[1] = -1
This.aRGB[2] = -1
This.aRGB[3] = -1

IF VARTYPE(nColor)="N"
   This.aRGB[3] = INT(nColor/(256^2))
   nColor = MOD(nColor,(256^2))
   This.aRGB[2] = INT(nColor/256)
   This.aRGB[1] = MOD(nColor,256)
ENDIF

RETURN @This.aRGB

ENDDEFINE
```

In versions prior to VFP 7, you have to simulate returning an array. One solution is to create an object with an array property and return the object. Here's the same class rewritten to work in VFP 6. The additional class, ArrayReturn, has a single custom property, to hold the return value.

```
#DEFINE CR CHR(13)

LOCAL nColor, oRGB, oReturn

* Get a color
nColor = GETCOLOR()

oRGB=CREATEOBJECT("Colors")
oReturn = oRGB.RGBComp(nColor)

MESSAGEBOX("The color is " + TRANSFORM(nColor) + "." + CR + ;
           "Red: " + TRANSFORM(oReturn.aReturnValue[1]) + CR + ;
           "Green: " + TRANSFORM(oReturn.aReturnValue[2]) + CR + ;
           "Blue: " + TRANSFORM(oReturn.aReturnValue[3]), ;
           64, "Show array return value")

RETURN

DEFINE CLASS Colors AS Custom
* Color handling code
DIMENSION aRGB[3]

FUNCTION RGBComp(nColor) AS Array
* RGBComp
* Returns the Red, Green and Blue Components
* of a color in an array

This.aRGB[1] = -1
This.aRGB[2] = -1
This.aRGB[3] = -1
```

```
IF VARTYPE(nColor)="N"
   This.aRGB[3] = INT(nColor/(256^2))
   nColor = MOD(nColor,(256^2))
   This.aRGB[2] = INT(nColor/256)
   This.aRGB[1] = MOD(nColor,256)
ENDIF

oReturn = CreateObject("ArrayReturn")
DIMENSION oReturn.aReturnValue[3]
ACOPY(This.aRGB, oReturn.aReturnValue)

RETURN oReturn

ENDDEFINE

DEFINE CLASS ArrayReturn AS Line
* Use Line because it's light-weight

DIMENSION aReturnValue[1]

ENDDEFINE
```

Array-filling functions

Visual FoxPro has quite a few functions whose purpose is to fill an array with certain data. One such function, ALINES(), is discussed in "Moving data from strings to arrays" above. The others are described below.

These functions all have some behaviors in common (except for exceptions noted below). First, they either resize or create the array, so that it's the right size for the data involved. That is, if the array exists, the function resizes it; if the array doesn't exist, the function creates it and makes it the right size. In addition, these functions return the number of rows or elements in the resulting array.

Array Manipulation

Visual FoxPro has a number of functions for working with arrays. A few change the array's contents, but most of them let you explore the array.

Determining array size

The ALEN() function tells you how big an array is. Depending how you call it, it can return the total number of elements, the number of rows or the number of columns. To determine the total number of elements in an array, call ALEN() and pass only the array, like this:

```
LOCAL aTest1D[ 12 ], aTest2D[ 7, 5 ]
? ALEN( aTest1D )  && returns 12
? ALEN( aTest2D )  && returns 35
```

To find the number of rows, add a second parameter of 1. For one-dimensional arrays, the result here is the same as if you'd omitted the second parameter. This is misleading since FoxPro views a 1-D array as a single row.

```
? ALEN( aTest1D, 1)  && returns 12
? ALEN( aTest2D, 1)  && returns 7
```

Passing 2 for the second parameter causes the function to return the number of columns. For a 1-D array, ALEN(Array, 2) returns 0. Again, this is misleading, but it also provides a way to determine whether an array is declared as one-dimensional or two-dimensional.

```
? ALEN( aTest1D, 2 )  && returns 0
? ALEN( aTest2D, 2 )  && returns 5
lIsOneD = ( ALEN( aTest1D, 2) = 0 )
```

Converting between element notation and row, column notation

As explained in "Array Basics", all FoxPro arrays are represented internally as one-dimensional. You can reference two-dimensional arrays using either a single element number or a row and a column. A pair of functions, AELEMENT() and ASUBSCRIPT(), convert between the two notations.

AELEMENT() takes an array, a row and a column (the latter is optional) and returns the element number. If the array is one-dimensional, the function returns the original row number. Interestingly, it does the same thing if only a row is passed – in this case, most likely, the array is being seen as 1-D and the second parameter is being interpreted as an element number.

Here are some examples:

```
DIMENSION aTest2D[ 13, 5 ]
? AELEMENT( aTest2D, 2, 4 )  && returns 9
? AELEMENT( aTest2D, 7, 1 )  && returns 31
? AELEMENT( aTest2D, 13 )    && returns 13
```

ASUBSCRIPT() performs a nearly inverse operation. It converts an element number to a row or a column. Because a function can return only a single item, it takes two calls to ASUBSCRIPT() to find both the row and the column.

This function has three parameters: the array, the element number and either 1 or 2 to indicate which subscript to return. Pass 1 to get the row, 2 to get the column. ASUBSCRIPT() works only on two-dimensional arrays – if you pass a one-dimensional array, you get an error.

Here are some examples:

```
DIMENSION aTest2D[ 13, 5 ]
? ASUBSCRIPT( aTest2D, 9, 1 )  && returns 2
? ASUBSCRIPT( aTest2D, 9, 2 )  && returns 4
? ASUBSCRIPT( aTest2D, 20, 1)  && returns 4
? ASUBSCRIPT( aTest2D, 20, 2)  && returns 5
```

ASUBSCRIPT() has a bug in versions prior to VFP 7 – it can't handle arrays with more than 32,767 rows. Of course, the only such arrays you can create have only one column. But, in that case, when you ask for the row number of an element beyond 32767, ASUBSCRIPT() returns 32767. This bug is fixed in VFP 7.

With VFP 7's new ability to return an array, we can write a function to return both the row and the column in a single array. The function is defined as a method of a class and the array is a property of the class. Here's the code for the class:

```
DEFINE CLASS cusArrayFns AS Custom

DIMENSION aReturn[2]

FUNCTION ABothSubscripts
* Return both subscripts of an array
* in an array

LPARAMETERS aInArray, nElement

* First, check parameters
DO CASE
CASE PCOUNT() <> 2
   * Did we get two params?
   ERROR 1229
   RETURN .T.

CASE VARTYPE(aInArray) = "U"
   * Does the array variable exist?
   ERROR 255
   RETURN .F.

CASE TYPE("aInArray[1]") = "U"
   * Is it an array?
   ERROR 232
   RETURN .F.

CASE ALEN( aInArray, 2 ) = 0
   * Is it a 2-D array?
   ERROR 1234
   RETURN .F.

CASE VARTYPE(nElement) <> "N"
   * Did we get an element number?
   ERROR 11
   RETURN .F.

CASE NOT BETWEEN( nElement, 1, ALEN(aInArray))
   * Is the subscript valid for this array?
   ERROR 1234
   RETURN .F.

OTHERWISE
   * So far, so good

ENDCASE

* Use the built-in functions to get the information
* and put it into the array.
This.aReturn[1] = ASUBSCRIPT( aInArray, nElement, 1 )
This.aReturn[2] = ASUBSCRIPT( aInArray, nElement, 2 )

RETURN @This.aReturn

ENDDEFINE
```

To use the function, we need to instantiate the class and then call the method:

```
LOCAL aTest[ 7, 5], aResult[ 2 ], oArrayFns
```

```
* Instantiate the class
oArrayFns = NewObject( "curArrayFns", "ArrayFns.PRG" )

* Call the method
aResult = oArrayFns.aBothSubscripts( @aTest, 15 )
```

Examining aResult in this case shows the row as 3 and the element as 5.

Adding and removing array data

Two functions let you manipulate the contents of an array. AINS() inserts a blank element, row or column at a specified position. ADEL() deletes the data from an element, row or column at a specified position.

Both AINS() and ADEL() adjust the data following the specified position appropriately. That is, AINS() moves the remaining elements closer to the end of the array, pushing the last one(s) out. ADEL() moves the remaining elements toward the front of the array, leaving the last one(s) empty. In most cases, you want to resize the array before calling AINS() and after calling ADEL().

While insertion and deletion combined with resizing work as you'd expect for rows, that's not so for columns. When you resize a 2-D array to add or remove columns, data is moved from one row to another. When you resize, then insert a column, data is lost. The same thing happens when you delete a column, then resize. (Of course, some data is discarded when you delete a column, but when you resize, additional data is lost.)

The syntax for AINS() is:

```
AINS( Array, nPos [, 2  ])
```

The interpretation of nPos is determined by the array and by the presence or absence of the third parameter. If the array is one-dimensional, nPos indicates an element. If the array is two-dimensional and the third parameter is omitted (or anything less than 2), nPos indicates a row. If the array is two-dimensional and the third parameter is 2, nPos indicates a column.

First, here's an example for one-dimensional arrays:

```
LOCAL aTest1D[ 7 ]
LOCAL nItem

FOR nItem = 1 TO ALEN( aTest1D )
   * Set each element to the corresponding letter of the alphabet
   aTest1D[ nItem ] = CHR(ASC("A") + nItem - 1)
ENDFOR

* Now add a new fourth item
AINS( aTest1D, 4 )

* Show result - fourth element is .F. and "G" is missing
FOR nItem = 1 TO ALEN( aTest1D )
   ? aTest1D[ nItem ]
ENDFOR
?
```

```
* Give the new element a value
aTest1D[ 4 ] = "New!"

* This time, enlarge the array first
DIMENSION aTest1D[ ALEN( aTest1D ) + 1]

* Now add a new second item
AINS( aTest1D, 2)

* Show result - second element is .F.
FOR nItem = 1 TO ALEN( aTest1D )
   ? aTest1D[ nItem ]
ENDFOR
?
```

This example shows insertion of rows and columns in a two-dimensional array.

```
LOCAL aTest2D[ 3, 7 ]
LOCAL nRow, nCol, nRowCount, nColCount

nRowCount = ALEN( aTest2D, 1)
nColCount = ALEN( aTest2D, 2)

* Fill array with element numbers
FOR nRow = 1 TO nRowCount
   FOR nCol = 1 TO nColCount
      aTest2D[ nRow, nCol ] = (nRow-1) * nColCount + nCol
   ENDFOR
ENDFOR

* Display initial data
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Add a new second row
AINS( aTest2D, 2 )

* Display result - second row is .F. and values above 15 are gone
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Reset to initial data
FOR nRow = 1 TO nRowCount
   FOR nCol = 1 TO nColCount
      aTest2D[ nRow, nCol ] = (nRow-1) * nColCount + nCol
   ENDFOR
ENDFOR
```

```
* This time, resize array first
DIMENSION aTest2D[ nRowCount + 1, nColCount ]
nRowCount = nRowCount + 1

* Now insert a new second row
AINS( aTest2D, 2 )

* Display result - second row is .F., but no data is lost
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Refill array with element numbers
FOR nRow = 1 TO nRowCount
   FOR nCol = 1 TO nColCount
      aTest2D[ nRow, nCol ] = (nRow-1) * nColCount + nCol
   ENDFOR
ENDFOR

* Now add a column
DIMENSION aTest2D[ nRowCount, nColCount + 1 ]
nColCount = nColCount + 1

* Display result – data has been rearranged
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Insert a new fifth column
AINS( aTest2D, 5, 2 )

* Display result - fifth column is .F., and data has been lost
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?
```

The last section of the code demonstrates the problem with inserting entire columns. Data is moved from one row to another to fill in empty spaces when the array is redimensioned. Then, when the column is inserted, some data falls out and is lost.

The syntax for ADEL() syntax is identical to AINS(), except for the name of the function:

```
ADEL( Array, nPos [, 2 ] )
```

Here's an example for a one-dimensional array:

```
LOCAL aTest1D[ 7 ]
```

```
LOCAL nItem

FOR nItem = 1 TO ALEN( aTest1D )
   * Set each element to the corresponding letter of the alphabet
   aTest1D[ nItem ] = CHR(ASC("A") + nItem - 1)
ENDFOR

* Now delete the fourth item
ADEL( aTest1D, 4 )

* Show result - "D" is missing and the last item is .F.
FOR nItem = 1 TO ALEN( aTest1D )
   ? aTest1D[ nItem ]
ENDFOR
?
```

Here's the two-dimensional example:

```
LOCAL aTest2D[ 5, 7 ]
LOCAL nRow, nCol, nRowCount, nColCount

nRowCount = ALEN( aTest2D, 1)
nColCount = ALEN( aTest2D, 2)

* Fill array with element numbers
FOR nRow = 1 TO nRowCount
   FOR nCol = 1 TO nColCount
      aTest2D[ nRow, nCol ] = (nRow-1) * nColCount + nCol
   ENDFOR
ENDFOR

* Display initial data
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Delete the second row
ADEL( aTest2D, 2 )

* Display result - values 8-14 are gone and last row is .F.
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Now, resize the array
DIMENSION aTest2D[ nRowCount - 1, nColCount ]
nRowCount = nRowCount - 1

* Display result - values 8-14 are gone and array has only 5 rows
FOR nRow = 1 TO nRowCount
   ?
```

```
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Refill array with element numbers
FOR nRow = 1 TO nRowCount
   FOR nCol = 1 TO nColCount
      aTest2D[ nRow, nCol ] = (nRow-1) * nColCount + nCol
   ENDFOR
ENDFOR

* Delete the fifth column
ADEL( aTest2D, 5, 2 )

* Display result - last column is .F., and 5th column data is gone
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Now resize the array
DIMENSION aTest2D[ nRowCount, nColCount-1 ]
nColCount = nColCount - 1

* Display result - only 6 columns, data has been rearranged
* and some items have been lost
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTest2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?
```

Again, the final example demonstrates the problems with manipulating columns. The deletion behaves as you expect, but when you resize the array to the smaller size, data is rearranged rather than just cutting out the empty final column.

Copying arrays

The ACOPY() function provides a quick and easy way to copy data from one array to another, but it also provides far more because the source and destination arrays don't have to match in size and shape. ACOPY() even lets you copy data from one place to another in the same array. The syntax for ACOPY() is:

```
nNumberCopied = ACOPY( SourceArray, DestArray
                       [, nSourceStart [, nNumberToCopy
                       [, nDestStart ] ] ] )
```

Perhaps the key to understanding what ACOPY() does is knowing that it treats all arrays as one-dimensional and, therefore, works at the element level. It just barely knows about rows and columns.

In the simplest case, you provide just the names of the source and destination arrays and data is copied. If the destination doesn't exist, it's created to exactly match the source. (Note that it's created as a private variable.) If the array exists and has a different shape (that is, a different number of rows or columns or both), data is copied element by element. If the destination is too small, an error is generated *after* the data has been copied.

If you specify nSourceStart, copying starts with that element. Similarly, if you specify nNumberToCopy, only that many elements are copied. Specifying these items lets you see one really strange behavior of ACOPY(). When the destination array is created, it always has exactly the same dimensions as the source array, even if you're only copying some of the source data.

Finally, nDestStart lets you indicate where to put the results in the destination array.

Here are some examples:

```
LOCAL aTest2D[ 5, 7 ]
LOCAL nRow, nCol, nRowCount, nColCount

nRowCount = ALEN( aTest2D, 1)
nColCount = ALEN( aTest2D, 2)

* Fill array with element numbers
FOR nRow = 1 TO nRowCount
   FOR nCol = 1 TO nColCount
      aTest2D[ nRow, nCol ] = (nRow-1) * nColCount + nCol
   ENDFOR
ENDFOR

* Copy everything
ACOPY( aTest2D, aCopy2D )

nRowCount = ALEN( aCopy2D, 1)
nColCount = ALEN( aCopy2D, 2)

* Display copied data
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aCopy2D[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Copy two rows
ACOPY( aTest2D, aTwoRows, 8, 14 )

nRowCount = ALEN( aTwoRows, 1)
nColCount = ALEN( aTwoRows, 2)

* Display copied data - note that array is 5 x 7, not 2 x 7
FOR nRow = 1 TO nRowCount
```

```
   ?
   FOR nCol = 1 TO nColCount
      ?? aTwoRows[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Specify start in destination
* Copy first row of source to 3rd row of destination
ACOPY( aTest2D, aTwoRows, 1, 7, 15 )

* Display copied data
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aTwoRows[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?
```

Due to its element-oriented nature, ACOPY() has one major limit. It's unable to copy columns. To do that, you have to write code. This function, called ACOLCOPY() does the job. Like ACOPY(), it has five parameters, but the last three refer to columns rather than elements.

```
FUNCTION aColCopy
* © 1998, Tamar E. Granor and Ted Roche
* From:  Hacker's Guide to Visual FoxPro 6.0
* Hentzenwerke Publishing. www.hentzenwerke.com

* Copy one or more columns from a source array to a destination array.

LPARAMETERS aSource, aDest, nStartCol, nColCount, nDestCol
  * aSource = array to be copied
  * aDest = destination array
  * nStartCol = first column to copy - required
  * nColCount = number of columns to copy - optional.
  *             Go to end of aSource, if omitted
  * nDestCol = first column of destination to receive copied data - optional
  *            1 if omitted

LOCAL nRetVal,nOldRows,nOldCols,nOldCount,nItem
  * nRetVal = return value, number of columns copied.
  *         = -1, if can't copy

* Check source array
IF TYPE("aSource[1]")="U" OR ALEN(aSource,2)=0
  * not a 2-d array, can't do it
  RETURN -1
ENDIF

* Check for starting column
IF TYPE("nStartCol")<>"N"
  RETURN -1
ENDIF

* Check number of columns. Compute if necessary
IF TYPE("nColCount")<>"N" OR nStartCol+nColCount>ALEN(aSource,2)
  nColCount=ALEN(aSource,2)-nStartCol+1
```

```
ENDIF

* Check destination column.
IF TYPE("nDestCol")<>"N"
  nDestCol=1
ENDIF

* Check destination array for size. It must exist to be passed in.
* First, make sure it's an array.
* Then, see if it's shaped right for all the data.
* Two cases - if enough cols, but not enough rows, can just add
* If not enough cols, have to move data around.
IF TYPE("aDest[1]")="U"
  DIMENSION aDest[ALEN(aSource,1),nColCount+nDestCol-1]
ELSE
  IF ALEN(aDest,2)>=nColCount+nDestCol-1  && enough columns
    IF ALEN(aDest,1)<ALEN(aSource,1)   && not enough rows
      DIMENSION aDest[ALEN(aSource,1),ALEN(aDest,2)]  && add some
    ENDIF
  ELSE
    * now the hard one
    * not enough columns, so need to add more (and maybe rows, too)
    nOldRows=ALEN(aDest,1)
    nOldCols=ALEN(aDest,2)
    nOldCount=ALEN(aDest)
    DIMENSION aDest[MAX(nOldRows,ALEN(aSource,1)),nColCount+nDestCol-1]

    * DIMENSION doesn't preserve data location, so we need to adjust the data
    * We go backward from the end of the array toward the front, moving data
    * down, so we don't overwrite any data by accident

    FOR nItem=nOldCount TO 2 STEP -1
      * Use new item number and old dimensions to determine
      * new item number for each element
      IF nOldCols<>0
        nRow=CEILING(nItem/nOldCols)
        nCol=MOD(nItem,nOldCols)
        IF nCol=0
          nCol=1
        ENDIF
      ELSE
        nRow=nItem
        nCol=1
      ENDIF

      aDest[nRow,nCol]=aDest[nItem]
    ENDFOR
  ENDIF
ENDIF

* finally ready to start copying
FOR nCol=1 TO nColCount
  FOR nRow=1 TO ALEN(aSource,1)
    aDest[nRow,nDestCol+nCol-1]=aSource[nRow,nStartCol+nCol-1]
  ENDFOR
ENDFOR

RETURN nColCount*ALEN(aSource,1)
```

Sorting array data

The ASORT() function lets you rearrange array contents based on that content. In VFP 7, ASORT() has been enhanced to allow case-insensitive sorting. The syntax for ASORT() is:

```
ASORT( Array [, nStartPosition [, nNumberToSort
       [, nSortOrder [, nFlags ] ] ] ]
```

The second parameter, nStartPosition, serves a double role. First, as its name suggests, together with nNumberToSort, it determines what portion of the array is sorted. Second, in a 2-D array, it indicates which column is used for sorting. (Since sorting a two-dimensional array moves entire rows, it would be meaningless to sort an array by row contents.) The position must be specified as an element number, not a row number. However, nNumberToSort indicates the number of elements for a 1-D array and the number of rows for a 2-D array. Pass –1 for either of these parameters if you need to specify one of the later parameters, but don't want to change these from the defaults (which sort the entire array, based on the first column, if it's 2-D).

You can sort into ascending or descending order. The default is ascending order. Pass 1 (or any other positive number) for nSortOrder to sort in descending order. A value of 0 or any negative number sorts in ascending order, the same as if you'd omitted the parameter.

The nFlags parameter is new in VFP 7. It provides a set of additive flags (that is, add all the ones you want together and pass the total for this parameter) to alter the sort operation. So far, there's only one flag value. Pass 1 to make the sort case-insensitive. A value of 0 for nFlags provides the default case-sensitive search.

Here are some examples:

```
LOCAL ARRAY aUKEmps[1]
LOCAL nRowCount, nColCount, nRow, nCol

SELECT first_name, last_name, birth_date ;
   FROM _SAMPLES+"TasTrade\Data\Employee" ;
   WHERE country = "UK" ;
   INTO ARRAY aUKEmps

nRowCount = ALEN(aUKEmps, 1)
nColCount = ALEN(aUKEmps, 2)

* Display results - note random order
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aUKEmps[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Sort into alphabetical order by last name
ASORT(aUKEmps, 2)

* Display results
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
```

```
      ?? aUKEmps[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* Sort into descending order by birthdate
ASORT(aUKEmps, 3, -1, 1)

* Display results
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aUKEmps[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?
```

The last example here shows the use of -1 for the nNumberToSort parameter, so that it's left at the default value.

This next example demonstrates the new case-sensitivity flag. The SELECT statement copies the first names of the employees three times, first as they appear in the table, then all lowercase, then all uppercase. This provides a data set where case-sensitivity is significant.

```
LOCAL ARRAY aNames[1]
LOCAL nElements, nItem

SELECT First_Name FROM _SAMPLES+"TasTrade\Data\Employee";
UNION ALL;
SELECT LOWER(First_Name) FROM _SAMPLES+"TasTrade\Data\Employee" ;
UNION ALL;
SELECT UPPER(First_name) FROM _SAMPLES+"TasTrade\Data\Employee" ;
INTO ARRAY aNames

nElements = ALEN(aNames)

* Display initial values - ordered as they came from the table
FOR nItem = 1 TO nElements
   ? aNames[ nItem ]
ENDFOR
?

* Sort case-sensitive (the default)
ASORT( aNames )

* Display result - all lower-case items are last
FOR nItem = 1 TO nElements
   ? aNames[ nItem ]
ENDFOR
?

* Now sort case-insensitive
ASORT( aNames, -1, -1, 0, 1 )

* Display result - same names are together
FOR nItem = 1 TO nElements
   ? aNames[ nItem ]
ENDFOR
```

```
?
```

The final example demonstrates that, for items with the same value, you can't predict the order in which they'll be put in the result.

Searching for data in an array

The final array manipulation function is ASCAN(). It lets you search for specific data in an array. Like ASORT(), ASCAN() has been enhanced in VFP 7. It now lets you specify the column to search, as well as indicating whether a search is case-sensitive, whether it should perform an exact search or a "starts with" search, and whether the function should return the element number or the row number of the matching item found.

The syntax for ASCAN() is:

```
nResult = ASCAN( Array, uSearchExpression
                 [, nStartPosition [, nNumberToSearch
                 [, nSearchColumn [, nFlags ] ] ] ] )
```

As in the other array functions, nStartPosition and nNumberToSearch let you limit the search to one portion of the array. But also, as in the other functions, these parameters are element-based. So, using them, it's not possible to search specific columns.

Enter the new nSearchColumn parameter. Pass a column number to search only that column. You can further limit the search by passing positive values for nStartPosition and nNumberToSearch. Then, only the specified elements in the specified column are searched.

The other new parameter, nFlags, lets you modify the search in several ways. It's additive, so you choose the items you want to include and add them together to get the value to pass. Table 1 shows the flag values.

Table 1 ASCAN() flags – Add the values shown together to create the nFlags parameter.

Bit	Value	Meaning
0	0	Search is case-sensitive. Default.
0	1	Search is case-insensitive.
1	0	Exact is off. Effective only when Bit 2 is set (4). Default.
1	2	Exact is on. Effective only when Bit 2 is set (4).
2	0	Current SET EXACT setting applies. Default.
2	4	Use the exactness setting from Bit 1.
3	0	Return the element number of the matching item. Default.
3	8	Return the row number of the matching item, if this is a two-dimensional array.

Two of the flag's bits deal with the exactness setting for the search. In FoxPro, you have a choice whether string searches look for an exact match or just for a string that has the search string at its beginning. The SET EXACT command controls this setting. Bit 2 of nFlags indicates whether the current SET EXACT setting should be used for this search. If bit 2 is off (0), then bit 1 is ignored. If bit 2 is on (4), then bit 1, rather than SET EXACT, determines whether this search is exact.

By default, ASCAN() returns the element number of the match found. Bit 3 of nFlags lets you specify that the row number should be returned instead.

These examples demonstrate a variety of ways to use ASCAN():

```
LOCAL ARRAY aUKEmps[1]

SELECT first_name, last_name, birth_date ;
   FROM _SAMPLES+"TasTrade\Data\Employee" ;
   WHERE country = "UK" ;
   INTO ARRAY aUKEmps

nRowCount = ALEN(aUKEmps, 1)
nColCount = ALEN(aUKEmps, 2)

* Display results - note random order
FOR nRow = 1 TO nRowCount
   ?
   FOR nCol = 1 TO nColCount
      ?? aUKEmps[ nRow, nCol ], " "
   ENDFOR
ENDFOR
?

* unlimited search
? ASCAN( aUKEmps, "Michael" ) && returns 4

* search beginning with row 3
? ASCAN( aUKEmps, "Michael", 7 )  && returns 0

* search only the first two rows
? ASCAN( aUKEmps, "Michael", 1, 6 ) && returns 4

* the actual element number is returned, even in a limited search
? ASCAN( aUKEmps, "Laura", 10, 3)  && returns 10

* search only column 1 - again element number is returned
? ASCAN( aUKEmps, "Laura", -1, -1, 1 )  && returns 10

* case-sensitive search
? ASCAN( aUKEmps, "laura" ) && returns 0

* case-insensitive search
? ASCAN( aUKEmps, "laura", -1, -1, -1, 1 ) && returns 10

* inexact search
? ASCAN( aUKEmps, "Call" ) && returns 11

* exact search
```

```
? ASCAN( aUKEmps, "Call", -1, -1, -1, 6 ) && returns 0

* exact, case-insensitive search and return row
? ASCAN( aUKEmps, "callahan", -1, -1, -1, 15 ) && returns 4
```

Exploring the environment

A number of functions retrieve information about the environment and put it into an array. This group can be divided into two subsets. The first, larger, subset tells you about what's out there on this computer and the network. The second tells you about your current status.

What's out there?

The functions in this group let you check out this computer and the network to which it's attached. They are ADIR(), AFONT(), AGetFileVersion(), ANetResources() and APrinters().

ADIR() fills an array with file information from a directory. You can limit the results by providing a file spec and by specifying file attributes that must be matched. In VFP 7, you can also indicate whether to preserve the capitalization of filenames and whether to provide the names in the DOS 8.3 format. The syntax for ADIR() is:

```
nFiles = ADIR( Array [, cFileSpec [, cAttributes [, nFlags ] ] ] )
```

cFileSpec uses the wildcards "*" and "?" and can include a drive and directory. If no directory is specified, ADIR() operates on the current directory.

By default, ADIR() returns only ordinary files, excluding directories, hidden files and system files. The cAttributes parameter lets you change that. You specify file attributes for cAttributes using the characters "D" for directory, "H" for hidden, "S" for system and "V" for volume. If "V" is included, the function returns only the volume name in a one-element array; all other attributes are ignored.

The nFlags parameter is additive, as with ASCAN(). In this case, only two bits are used. The lowest-order bit indicates whether to convert all filenames to uppercase (the default or 0) or show them with their existing capitalization (add 1 to nFlags). The next bit determines whether the actual filenames are returned (add 0) or their DOS-compliant 8.3 counterparts are used (add 2 to nFlags).

Except when "V" is included in cAttributes, ADIR() creates a five-column array, as shown in Table 2.

Table 2 File information – ADIR() creates a five-column, with each row representing one file.

Column	Meaning
1	File name—character.
2	File size in bytes—numeric.

3	Date file was last written—date.
4	Time file was last written—character.
5	File Attributes—a single five-character string. Each position represents one attribute. If the file has that attribute, the corresponding letter is there; if not, a "." is found in that location. The attributes are: R—read-only A—archive bit is set S—system file H—hidden file D—directory For example, "R...." means a read-only file, while ".A.H." means a hidden file needing to be archived.

There's one trick to working with ADIR(). To retrieve only files in a directory that have the specified attributes, specify the empty string for cFileSpec.

For example:

```
? ADIR( aRoot, "C:\*.*" ) && fills array with all files in root directory of C
? ADIR( aDirs, "", "D" )  && fills array with subdirectories of
                          && current directory
? ADIR( aHS, "", "HS" )   && fills array with only hidden and system files in
                          && current directory
? ADIR( aPrgs, "*.PRG" )  && fills array with program files in
                          && current directory
? ADIR( aLibs, "C:\WINNT\SYSTEM32\*.DLLS" ) && fills array with DLLS in
                                             && system directory
? ADIR(aPrgs, "*.PRG", "", 1) && fills array with program files in current
                              && directory, preserving capitalization
? ADIR(aPrgs, "*.PRG", "", 3) && fills array with program files in current
                              && directory, preserving capitalization and
                              && showing 8.3 versions of long filenames
```

A number of the array manipulation functions are useful for working with the results of ADIR(). In particular, the VFP 7 enhancements to ASCAN() make it much simpler to find information about a particular file. For example, you can search for a particular program in the array aPrgs, created above, like this:

```
nPrgRow = ASCAN( aPrgs, cPrgName, -1, -1, 1, 9 )
```

This search looks only in column one, is case-insensitive, and returns the row of the match.

The FileSystemObject that's part of the Windows Scripting environment provides access to the same information as ADIR() does. However, this approach takes about an order of magnitude longer than using ADIR().

What ADIR() does for files, AFONT() does for fonts. The function lets you find out what fonts are available. The syntax for AFONT() is:

```
lSuccess = AFONT( Array [, cFontName [, nFontSize ] ] )
```

Unlike the other array-filling functions, AFONT() returns a logical value indicating its success or failure.

When you hand AFONT() only an array, the function fills the array with a list of the installed fonts. But you can limit it to a particular font or even a particular font/size combination by passing the optional parameters. Doing so changes the structure of the resulting array.

If you pass just a font name, the array has one element for each size available for that font. For scalable fonts, the array has a single element containing -1.

If you pass a font name and a size, and the font is available in that size, the array has a single element containing .T. If the font is not available in the specified size, the function does one of two things. If the array already exists, the first element is set to .F., but the array is not resized. If the array doesn't already exist, it's not created. In either case, the function returns .F.

Although the function's behavior seems weird, it provides a lot of functionality. For example, you can collect the list of all installed fonts, so that you can let the user choose a font for reporting. Then you could pass that font name to the function to get a list of font sizes and let the user choose one of those, too.

The third option for the function gives you an easy way to find out whether a particular font/size combination is available. In this case, though, you probably won't ever look at the resulting array.

The third function that lets you explore the computer's configuration is APRINTERS(). It fills an array with a list of the installed printers and returns the number of printers found. The resulting array has two columns: the first contains the name of the printer and the second has the port on which it's installed. Here's a call to the function and its results on my primary machine:

```
? APRINTERS( aPtrs )  && returns 1
LIST MEMORY LIKE aPtrs  && results reformatted for readability

APTRS                   Pub             A
   (    1,    1)        C     "HP LaserJet 4L"
   (    1,    2)        C     "LPT1:"
```

There's one trap with APRINTERS(). What it tells you about a printer is that it was once installed in Windows. It doesn't tell you whether it's still attached, working and on-line.

If the machine your app is running on is attached to a network, you may want to know about what's available on the network. Enter ANetResources(). This function can ask about either files or printers and fills an array with a list of those shared resources. The syntax is:

```
nResources = ANetResources( aResources, cNetworkName, 1 | 2 )
```

Pass 1 for the third parameter to fill the array with network shares and 2 to fill it with printers. In VFP 6 (where this function was new), cNetworkName had to be the name of a computer on the network in UNC notation. VFP 7 has more flexibility in this area and accepts a domain name, as well. When you pass a domain name, the array is filled with members of the domain. You can use those as parameters to the function to find out what each offers.

AGetFileVersion(), added in VFP 6, gives you access to the file properties that are part of various kinds of executable files (including EXE, DLL, OCX, TLB, and OLB files). This is the same

information you can find by right-clicking on the file in Explorer and choosing Properties. Since you can build that information into your VFP executables, one of the things this function gives you is the ability to check the file version of your own applications. You can also use it to make sure you have the expected version of a particular program or tool, and to retrieve information for an About dialog.

AGetFileVersion() accepts an array and a file name. If the specified file contains version information, the array is sized to 15 elements, containing the version information. In this case, the function returns 15. If the file doesn't exist or doesn't contain version information, the function returns 0 and the array is unchanged. Table 3 shows the meaning of each of the array elements.

Table 3 File Version Information – The AGetFileVersion() function provides the information that's in the Properties dialog for .EXE and .DLL files and a whole lot more.

Element	Contents
1	File comments
2	Company Name
3	File Description
4	File Version
5	Internal Filename
6	Copyright
7	Trademarks
8	Original Filename
9	Private Build
10	Product Name
11	Product Version
12	Special Build
13	Indicates whether this file supports OLE self-registration.
14	Language
15	Translation Code

Here's a look at the information for VFP6.EXE (with SP3 installed):

```
? AGetFileVersion( aVersion, HOME()+"VFP6.EXE" )
```

```
LIST MEMORY LIKE aVersion  && output reformatted to fit

AVERSION                Pub            A
(   1)  C     ""
(   2)  C     "Microsoft Corporation"
(   3)  C     "Microsoft® Visual FoxPro®"
(   4)  C     "6.0.8492.0"
(   5)  C     "VFP6"
(   6)  C     "Copyright© Microsoft Corp. 1993-1998"
(   7)  C     "Microsoft® is a registered trademark of Microsoft Corporation."
(   8)  C     "VFP6.EXE"
(   9)  C     ""
(  10)  C     "Microsoft® Visual FoxPro®"
(  11)  C     "6.0.8492.0"
(  12)  C     ""
(  13)  C     "OLESelfRegister"
(  14)  C     "English (United States)"
(  15)  C     "040904e4"
```

In VFP 5, you can use the GetFileVersion() function in FoxTools to retrieve version information. There are a couple of differences between GetFileVersion() and AGetFileVersion(). First, GetFileVersion() expects its parameters in the opposite order; that is, the file name is first and the array second. In addition, the array must be created with 12 elements before calling the function and it must be passed by reference. Also, using GetFileVersion(), the resulting array has only 12 items. The last three items shown in Table 3 are not provided by the function.

Finally, it's worth noting that not even Microsoft uses all the elements of the version information uniformly. Comparing the results from different Microsoft products shows that different teams interpret these items differently. When you examine files from other companies, the variations are even wider.

What's going on here?

There are a couple of functions that let you check on what's happening inside VFP.

ADLLs(), as its name suggests, puts a list of the declared DLL functions into an array. This function is a welcome addition in VFP 7. In older versions, to get this information, you have to parse the output of LIST STATUS.

The only parameter to ADLLs() is the array. The function returns the number of declared DLL functions. They're listed in three columns in the array: the function's name, the function's alias (in case it was declared using the AS clause), and the library containing the function.

Here's an example:

```
DECLARE INTEGER GetSystemDirectory ;
        IN Win32api AS GetSysDir;
        STRING @cWinDir, ;
        INTEGER nWinDirLength
DECLARE DOUBLE GetSysColor ;
        IN Win32API;
        INTEGER nIndex
? ADLLS( aFns ) && returns 2
LIST MEMORY LIKE aFns  && reformatted for readability
```

```
AFNS                        Pub             A
    (    1,    1)           C       "GetSysColor"
    (    1,    2)           C       "GetSysColor"
    (    1,    3)           C       "C:\WINNT\system32\USER32.DLL"
    (    2,    1)           C       "GetSystemDirectory"
    (    2,    2)           C       "GetSysDir"
    (    2,    3)           C       "C:\WINNT\system32\KERNEL32.DLL"
```

Although it's not apparent from this example, the list is created in alphabetical order by function name.

The AMouseObj() function tells you where the mouse is, if it's over a form or certain elements of the VFP environment. It puts four items into the array you pass it: an object reference to the object over which the mouse is positioned, an object reference to that object's container, and the X and Y coordinates of the mouse position relative to the container. If you pass the optional second parameter, the second element is an object reference to the containing form (no matter how deep in the object hierarchy you are) and the last two elements get the coordinates of the point relative to the form rather than to the immediate container.

Some examples should make this clearer than any words can. The form in Figure 3 is designed to demonstrate AMouseObj(). The Mouse Position button's Click event runs AMouseObj(), then provides an English report of the results of the function call. The Relative to Form button's Click event runs AMouseObj() with the optional second parameter and reports on the results. In this example, the mouse is positioned over the checkbox on page 1 of the pageframe. The edit box shows the results.

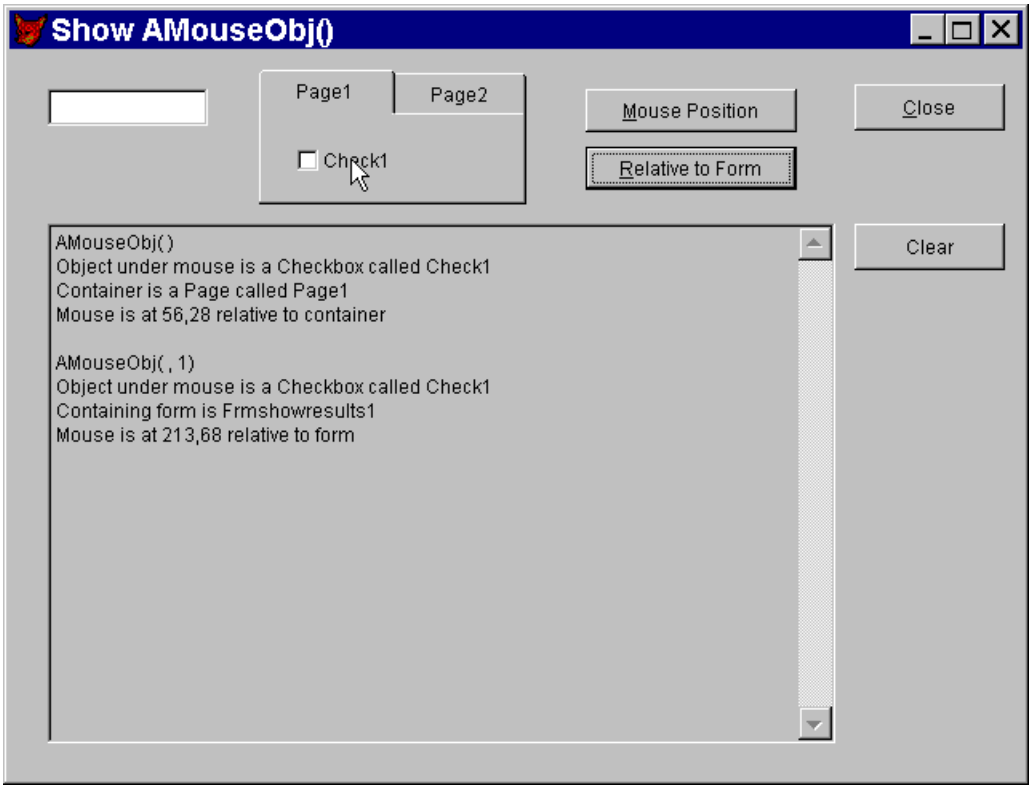

Figure 3 Demonstrating AMouseObj() – Here, the mouse is over the checkbox on page 1 of the pageframe. The edit box contains an English language explanation of the results of calling AMouseObj() without the second parameter, then calling it with the second parameter.

Note that, on this form, the buttons must be used by their hotkeys. Clicking on a button would move the mouse and change the results.

AMouseObj() doesn't always behave as you'd expect and its behavior at runtime is sometimes different than at design-time. In either case, if the mouse is over a page of a pageframe (except for the tab portion), AMouseObj() shows the page as both the object and the container.

In grids, there's an assortment of behaviors. At design-time, the function can identify headers, but at runtime, with the mouse over a header, the function shows the column as both control and container. For the controls in a column, AMouseObj() sees the column as both control and container. If you need to figure out where you are in a grid, use the grid's GridHitTest method instead.

Working with data

Another group of functions provides information about databases and their contents and the data environment. Some of these functions tell you about what's open and available now, while the others document databases and tables.

What data is available?

ADatabases(), ASessions() and AUsed() work together to let you see what databases and tables are open at any given time. Using them, you can get a fairly complete picture of the current data status.

ADatabases() fills an array with a list of the open databases. The array has two columns, with the first containing the name of the database and the second holding its full path. For example:

```
OPEN DATABASE _SAMPLES + "\TasTrade\Data\TasTrade"
OPEN DATABASE _SAMPLES + "\Data\TestData.dbc"
? ADATABASES( aOpen )  && returns 2
LIST MEMORY LIKE aOpen   && reformatted for readability

AOPEN                    Pub            A
   (    1,    1)        C     "TESTDATA"
   (    1,    2)        C     "D:\SEDONA\SAMPLES\DATA\TESTDATA.DBC"
   (    2,    1)        C     "TASTRADE"
   (    2,    2)        C     "D:\SEDONA\SAMPLES\TASTRADE\DATA\TASTRADE.DBC"
```

ASessions(), which is new in VFP 7, fills an array with a list of the active data sessions. The resulting array has one column, which contains the data session id. Although it would seem that each element of this array would be the same as its position, that's not so.

Data session ids are given out in the order in which they're created. However, when a data session is closed, its id is recycled. So, for example, if you open three forms with private data

sessions, they'll be assigned session ids 2, 3 and 4. (Data session 1 is always the default, public, data session.) If you then close the form with data session 2, then open a new form, the new form is assigned data session 2. But ASessions() lists the sessions in the order they were created. (You can see the same behavior in the current session dropdown of the Data Session window.)

What about finding out what tables, views and cursors are open in a particular data session? Before VFP, it was simple enough to loop through all work areas to see what they contained. But VFP has 32,767 work areas per data session, far too many to loop through. Enter AUsed(). This function fills a two-column array with the list. The first column holds the alias and the second has the work area number. AUsed() accepts two parameters. The first is the array. The second is optional and specifies a data session. If it's omitted, the current data session is used.

You can combine ASessions() and AUsed() to get a complete picture of open tables. Here's a function that accomplishes that task. Pass it an array by reference. Like the built-in array functions, it returns the number of rows in the result. If the result is non-0, it resizes the array appropriately.

```
* aOpenTables.PRG
* Fill an array with a complete list of open tables (cursors and views),
* regardless of data session. The resulting array has three columns:
* Alias, Work Area, and Data Session.
LPARAMETERS aOpenAreas

LOCAL ARRAY aSessionList[1], aOpen[1], aAllOpen[1,3]
LOCAL nSessionCount, nSession, nOpenCount, nTotalOpen

nTotalOpen = 0
* First, get the list of data sessions.
nSessionCount = ASESSIONS( aSessionList )

FOR nSession = 1 TO nSessionCount
   * Get the list for one session
   nOpenCount = AUSED( aOpen, aSessionList[ nSession ])

   IF nOpenCount > 0
      * Add to overall list. First, enlarge array
      DIMENSION aAllOpen[ nTotalOpen + nOpenCount, 3]

      * Copy data. ACOPY() is no use because it can't hold columns.
      FOR nAlias = 1 TO nOpenCount
         aAllOpen[ nTotalOpen + nAlias, 1 ] = aOpen[ nAlias, 1]
         aAllOpen[ nTotalOpen + nAlias, 2 ] = aOpen[ nAlias, 2]
         aAllOpen[ nTotalOpen + nAlias, 3 ] = aSessionList[ nSession ]
      ENDFOR

      nTotalOpen = nTotalOpen + nOpenCount
   ENDIF
ENDFOR

* If anything found, copy results to parameter array
IF nTotalOpen > 0
   DIMENSION aOpenAreas[ nTotalOpen, 3 ]
   ACOPY( aAllOpen, aOpenAreas )
ENDIF
```

```
RETURN nTotalOpen
```

What does my data look like?

Three array functions combine to give you a pretty good look at the structure of a database. They are ADBObjects(), AFIELDS(), and ATagInfo().

ADBObjects() works at the database level. Depending on a parameter, it fills an array with information about the tables, the views, the connections or the relations in the current database. The syntax is:

```
nObjectCount = ADBObjects( Array, cInfoType )
```

The cInfoType parameter can be "TABLE", "VIEW", "CONNECTION" or "RELATION". For the first three, the resulting array is one-dimensional with one element for each item of the specified type. When you pass "RELATION", the resulting array has five columns, as shown in Table 4.

Table 4 Relation information – When the second parameter to ADBObjects() is "RELATION", the resulting array has these five columns.

Column	Meaning
1	Child table.
2	Parent table.
3	The tag the relation is based on in the child table.
4	The tag the relation is based on in the parent table.
5	The referential integrity constraints on the relation in the order: Update, Delete, Insert. For each type of integrity, there are three possible values: C = cascade R = restrict I = ignore If no referential integrity of a type was defined, that position is empty.

Once you know what objects are in the database, you can use DBGetProp() to find out more about them. In fact, that's the strategy used by GenDBC.PRG (which comes with VFP – look in the Tools\GenDBC directory). GenDBC creates a program you can run to recreate the structure of a database.

Information about a specific table can be gathered using AFields() and ATagInfo(). AFields() fills an array with field information. By default, it works on whatever's open in the current work

area, but an optional second parameter lets you specify an alias or work area. In VFP 5 and later, the resulting array has 16 columns, as shown in Table 5.

Table 5 What's in a table? – AFields() creates an array of field information, containing these 16 columns. In VFP 3, the table has only 11 columns, omitting the last 5 shown here.

Column	Type	Meaning
1	Character	Field name.
2	Single Character	Field type.
3	Numeric	Field size.
4	Numeric	Decimal places.
5	Logical	Nulls allowed?
6	Logical	Code page translation NOT allowed?
7	Character	Field Validation rule (from DBC).
8	Character	Field Validation text (from DBC).
9	Character	Field Default value (from DBC).
10	Character	Table Validation rule (from DBC).
11	Character	Table Validation text (from DBC).
12	Character	Long table name (from DBC).
13	Character	Insert Trigger expression (from DBC).
14	Character	Update Trigger expression (from DBC).
15	Character	Delete Trigger expression (from DBC).
16	Character	Table Comment (from DBC).

In addition to containing information about the first field in the table, the first row of the array contains some database-level information, such as the table rule and triggers.

The changes to ASCAN() make it easier to find the information for a particular field in a table:

```
* Check whether current table has the field named by cField
AFields( aFldList )
nRow = ASCAN( aFldList, UPPER(cField), -1, -1, 8)
IF nRow > 0
   * the desired field is in row nRow
ENDIF
```

The newcomer to this group is ATagInfo(), added in VFP 7. It puts information about a table's index tags into an array. In addition to the array, the function can accept two additional parameters, both optional. The latter of these is the alias or work area. The optional second parameter is the name of a particular index field to look at. Here's the syntax:

```
nTags = ATagInfo( Array [, cIndexFile [, cAlias | nWorkArea ] ] )
```

The resulting array has six columns, as shown in Table 6. It contains almost everything you need to know to rebuild your indexes. The only thing missing here is which index file the tag comes from. In most situations, that's irrelevant, since most tables have only a structural .CDX file.

Table 6 Collecting index info – The array created by ATagInfo() has five columns, containing the details of each index tag. All data is character.

Column	Contents
1	Name
2	Type ("REGULAR", "CANDIDATE", "PRIMARY", "UNIQUE")
3	Key expression
4	Filter expression
5	Direction ("ASCENDING", "DESCENDING")
6	Collation sequence

Due to the way that VFP handles Ascending vs. Descending order in indexes, the last column of the array tells you only the current order for each tag, not the order with which it was created.

One function that VFP is missing is KeyExists() to accept a key expression and tell you whether a table has a key with that expression. ATagInfo() provides one way to write such a function.

```
* KeyExists.PRG
* Determine whether a table has a tag
* with a specified key expression.

LPARAMETERS cKeyExpr, cIndexFile, uWhatTable
   * 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)
```

```
LOCAL ARRAY aTagList[1]
LOCAL nTagCount, nResult

* Check key expression parameter
IF VarType( cKeyExpr ) <> "C"
   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 )
ELSE
```

```
   * There was no such index file as specified
   nResult = 0
ENDIF

RETURN nResult > 0
```

Exploring objects

Yet another group of functions provides information that aids in object-oriented programming. Many of the functions in this group are more oriented toward building tools than for use in applications.

The most tool-oriented of these functions are ASelObj() and AGetClass(). ASelObj() provides access to the currently selected objects. It fills an array with an object reference to each selected object. Passing 1 for the optional second parameter gives you a reference to the containing object for each selected object, instead. Passing 2 for that parameter gives you a reference to the data environment of the containing form for each object.

ASelObj() is most useful for creating builders. You can select a bunch of objects, then run a program (either from the Command Window or through the builder mechanism). Then use ASelObj() to find out which objects the builder is supposed to work on. For example, this example code lines up the left-hand edge of all the selected objects and makes them all the same height.

```
LOCAL ARRAY aObjs[1]
LOCAL nObjCount, nObj

nObjCount = ASelObj( aObjs )

IF nObjCount > 0
   FOR nObj = 1 TO nObjCount
      aObjs[ nObj ].Left = 37
      aObjs[ nObj ].Height = 40
   ENDFOR
ENDIF
```

AGetClass() brings up the Open dialog, configured for choosing a class. When the user makes a choice, the class library and name of the class chosen are stored in a two-element array. Like other functions that display system dialogs, AGetClass() has parameters to let you customize the dialog's appearance:

```
lChoseOne = AGetClass( Array [, cClassLib [, cClass [, cDialogCaption
                       [, cFileNameCaption [, cButtonCaption ] ] ] ] ] )
```

The second and third parameters, cClassLib and cClass, let you prime the pump. When these are passed, the dialog appears with the specified class library and class highlighted. cDialogCaption lets you specify the text that appears on the dialog's title bar. cFileNameCaption specifies the text that appears next to the file name textbox. Finally, cButtonCaption indicates the text for the OK button. For example, the call shown here produces the dialog in Figure 4.

```
AGETCLASS(aClInfo, HOME() + "FFC\_Base", "_textbox", ;
          "Choose a textbox class", "Classlib:", "Choose")
```

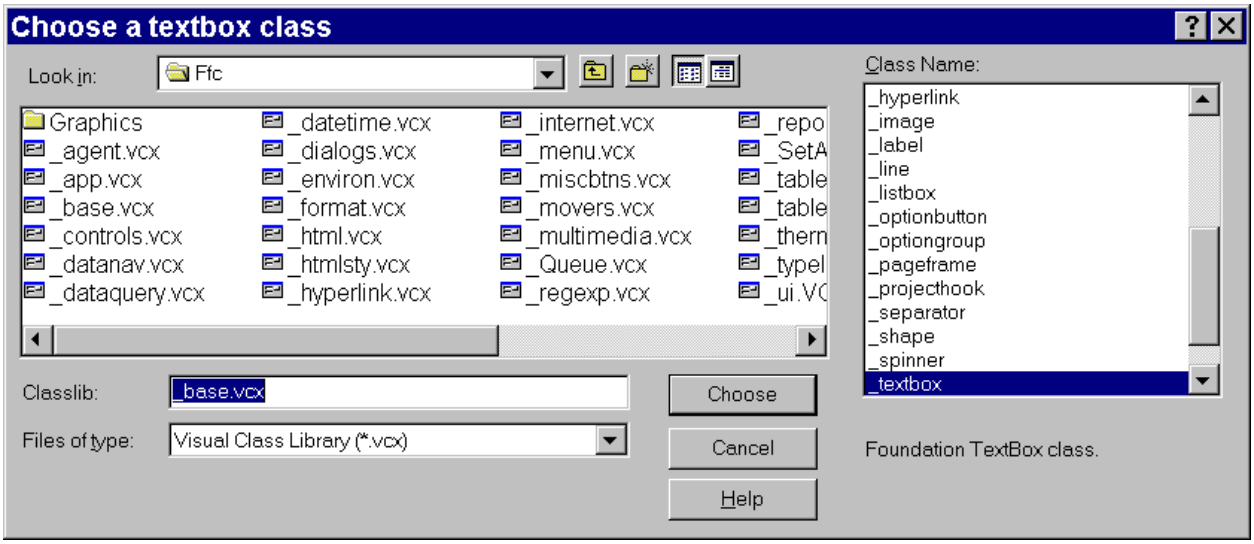

Figure 4 Choosing a class – The parameters to AGetClass() let you change the appearance of the specialized Open dialog for classes.

AGetClass() is handy when you want to create a new form or class and don't remember exactly where to find it. Use this function to locate and choose the class. Then, use the results in the CREATE FORM or CREATE CLASS command, like this:

```
AGetClass( aClInfo )
CREATE FORM MyNewForm AS (aClInfo[2]) FROM (aClInfo[1])
```

Another function that's probably used more for tools than for end-user applications is AVCXClasses(), which fills an array with a list of the classes in a specified class library. The resulting array has 11 columns, providing the basic information about the class such as the name of the class, the base class, the parent class name and class library and the description. However, the function doesn't provide information like the properties or methods. It's aimed more at identifying classes than being able to put them back together. In addition, properties and methods are available through other language features such as AMembers(), ReadExpression and ReadMethod.

This example probes the _datetime class library in the FoxPro Foundation Classes:

```
nClassCount = AVCXClasses( aClassList, HOME() + "FFC\_DateTime" )
```

Here are the results, slightly reformatted:

```
ACLASSLIST                Pub             A
   (     1,     1)         C      "_stopwatch"
   (     1,     2)         C      "container"
   (     1,     3)         C      "_container"
   (     1,     4)         C      "_base.vcx"
   (     1,     5)         C      "graphics\watch.bmp"
   (     1,     6)         C      "graphics\watch.bmp"
   (     1,     7)         C      "Pixels"
   (     1,     8)         C      "stop watch with start, stop, and reset methods"
   (     1,     9)         C      ""
   (     1,    10)         C      ""
```

```
(    1,   11)        L     .F.
(    2,    1)        C     "_olecalendar"
(    2,    2)        C     "olecontrol"
(    2,    3)        C     "olecontrol"
(    2,    4)        C     ""
(    2,    5)        C     ""
(    2,    6)        C     ""
(    2,    7)        C     "Pixels"
(    2,    8)        C     ""
(    2,    9)        C     ""
(    2,   10)        C     ""
(    2,   11)        L     .F.
(    3,    1)        C     "_clock"
(    3,    2)        C     "container"
(    3,    3)        C     "_container"
(    3,    4)        C     "_base.vcx"
(    3,    5)        C     "graphics\clock.bmp"
(    3,    6)        C     "graphics\clock.bmp"
(    3,    7)        C     "Pixels"
(    3,    8)        C     "day, date, and time control"
(    3,    9)        C     ""
(    3,   10)        C     ""
(    3,   11)        L     .F.
```

Once you know what classes are in a library, you might want to know where they came from. AClass() takes an object and fills an array with its inheritance hierarchy. Unlike AVCXClasses(), which works with just the name of a class library, for AClass(), you need an instance of the class. For example, to find out the heritage of the _stopwatch class in the _datetime class library, use:

```
LOCAL oWatch
LOCAL ARRAY aHierarchy[ 1 ]
oWatch = NewObject( "_stopwatch", HOME() + "FFC\_datetime" )
nLevels = AClass( aHierarchy, oWatch )
```

The array has as many elements as there are levels in the hierarchy, with the object itself at the top and the base class at the bottom. In the _stopwatch example, the array has three elements:

```
_STOPWATCH
_CONTAINER
CONTAINER
```

While AClass() tells you where a class came from, AInstance() tells you how it's being used right now. You pass it an array and a class name and the array is filled with the names of variables that reference instances of that class. For example:

```
LOCAL oWatch, oClock, oDontWatch, oWatchCopy
oWatch = NewObject( "_stopwatch", HOME() + "FFC\_datetime" )
oClock = NewObject( "_stopwatch", HOME() + "FFC\_datetime" )
oDontWatch = NewObject( "_stopwatch", HOME() + "FFC\_datetime" )
? AINSTANCE( aWatches, "_stopwatch")  && returns 3
oWatchCopy = oWatch
? AINSTANCE( aWatches, "_stopwatch")  && returns 4
```

The last example demonstrates an interesting feature of AINSTANCE(). It finds not unique instances of the specified class, but all variables that reference it. This is useful when you're trying to avoid leaving dangling references (one of the best ways to crash VFP). However, that use is limited because the function only looks at variables and doesn't consider objects contained in other objects.

The final object-oriented array function, AMembers(), is probably the most used. It's handy at design-time and runtime. The function provides a variety of information about classes and objects. Depending on the parameters you pass, it can give you a list of properties, of all members (properties, methods and contained objects), or of contained objects. In VFP 7, the function can return detailed information about the PEMs. The syntax is:

```
nCount = AMembers( Array, oObject | cClass [, 1 | 2 | 3 [, cFlags ] ] )
```

While you can provide a class name as the second parameter, it only works if the class has been instantiated at least once so that the class definition is in memory. Issuing SET CLASSLIB is not sufficient. In VFP 7, oObject can be a COM object as well as a native VFP object, but the third parameter must be 3, in that case.

The structure of the array varies, depending on the third parameter. Table 7 shows the various choices.

Table 7 AMembers() options – The third parameter to AMembers() determines the structure of the result.

Second parameter	Result
Omitted	The array is one-dimensional and contains one element for each property.
1	The array has two or three columns. The first contains the name of a property, method or contained object. The second contains the string "Property", "Method" or "Object". If cFlags contains "#", there is a third column that contains the flags that apply to the member.
2	If cFlags is omitted or does not include "#", the array is one-dimensional and each element contains the name of one contained object. If cFlags includes "#", the array has two columns. The first contains the name of the contained objects and the second contains the flags that apply to the objects.
3 (VFP 7 only)	The array has four or five columns and lists all properties and methods. The first column contains the name of the property or method. The second contains a string that indicates the type of member: "METHOD", "PROPERTYPUT", "PROPERTYGET", or "PROPERTYPUTREF". The third column is empty for properties of native objects. For methods of native objects, it contains the parameter list. For COM objects, it contains the property's or method's signature (which includes parameters and return value). The fourth column contains the help string for the property or method. If cFlags contains "#", there is a fifth column that contains the flags that apply to the property or method.

The cFlags parameter, new in VFP 7, lets you restrict the members placed in the array. It's valid only when the third parameter is 1 or 2, or when the third parameter is 3 and the object or class is native VFP (as opposed to a COM object). The flags you can specify can be divided into groups and the flags in each group are mutually exclusive. However, when you specify multiple flags, by

default, they're combined with "OR". Add the "+" flag somewhere in the string to combine flags with "AND" rather than "OR". Table 8 shows the flag characters.

Table 8 Limiting AMembers() results – Combine these flag characters to restrict the array filled by AMembers() to only some members.

Flag character	Filter group	Meaning
P	Visibility	Protected
H	Visibility	Hidden
G	Visibility	Public
N	Origin	Native PEMs
U	Origin	User-defined PEMs
I	Inheritance	Inherited PEMs
B	Inheritance	Base PEMs
C	Changed	Changed
R	Read-only	Read-only
+	Control	Combine flags with "AND" rather than "OR".
#	Control	Add the flags column to the result.

When "#" is included in cFlags, the array has an extra column. For each item in the array, the final column contains a string, showing which flag characters apply.

The form in Figures 5 and 6 demonstrates AMEMBERS() using 3 for a third parameter.

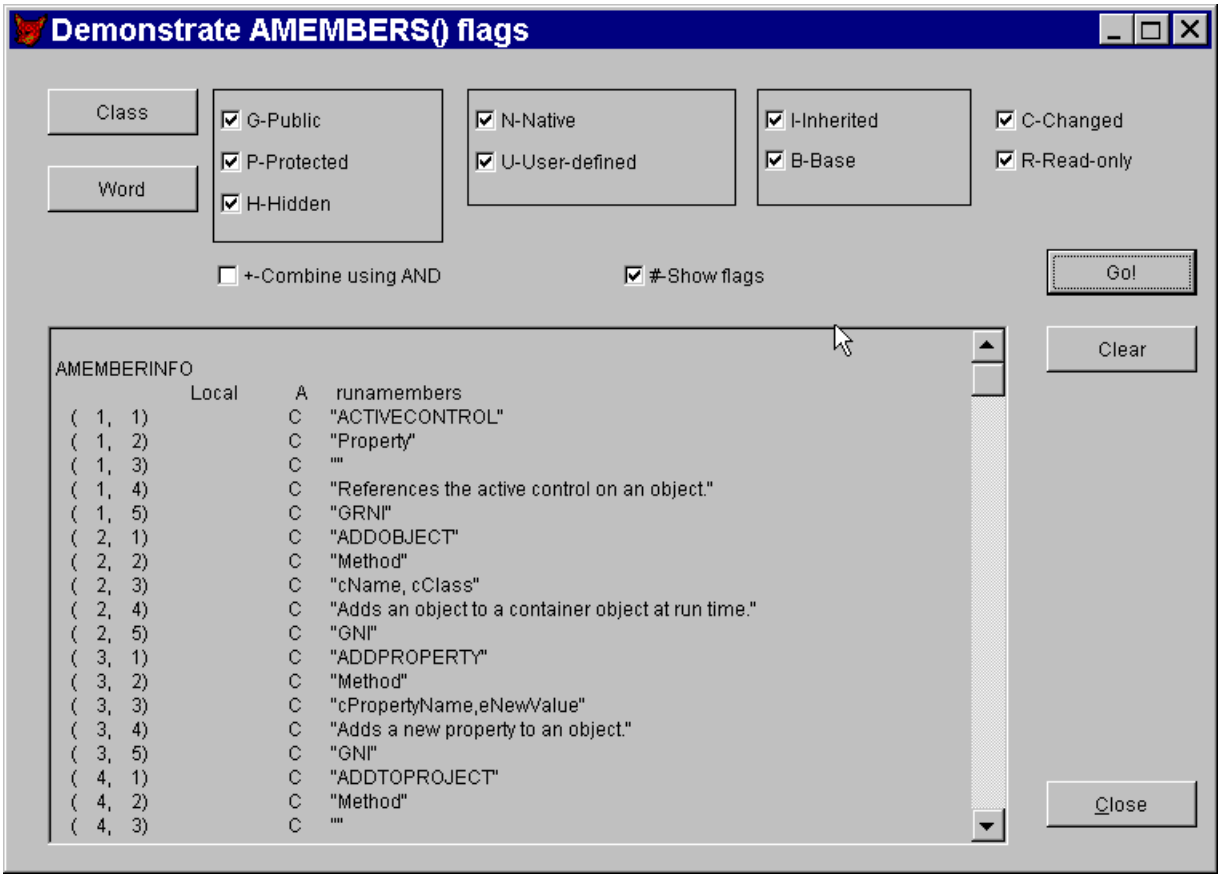

Figure 5 Returning detailed information from AMembers() – The object whose members are shown here was instantiated from the _StopWatch class in the FFC. AMembers() was called with a third parameter of 3, and with "#" included in the cFlags parameter.

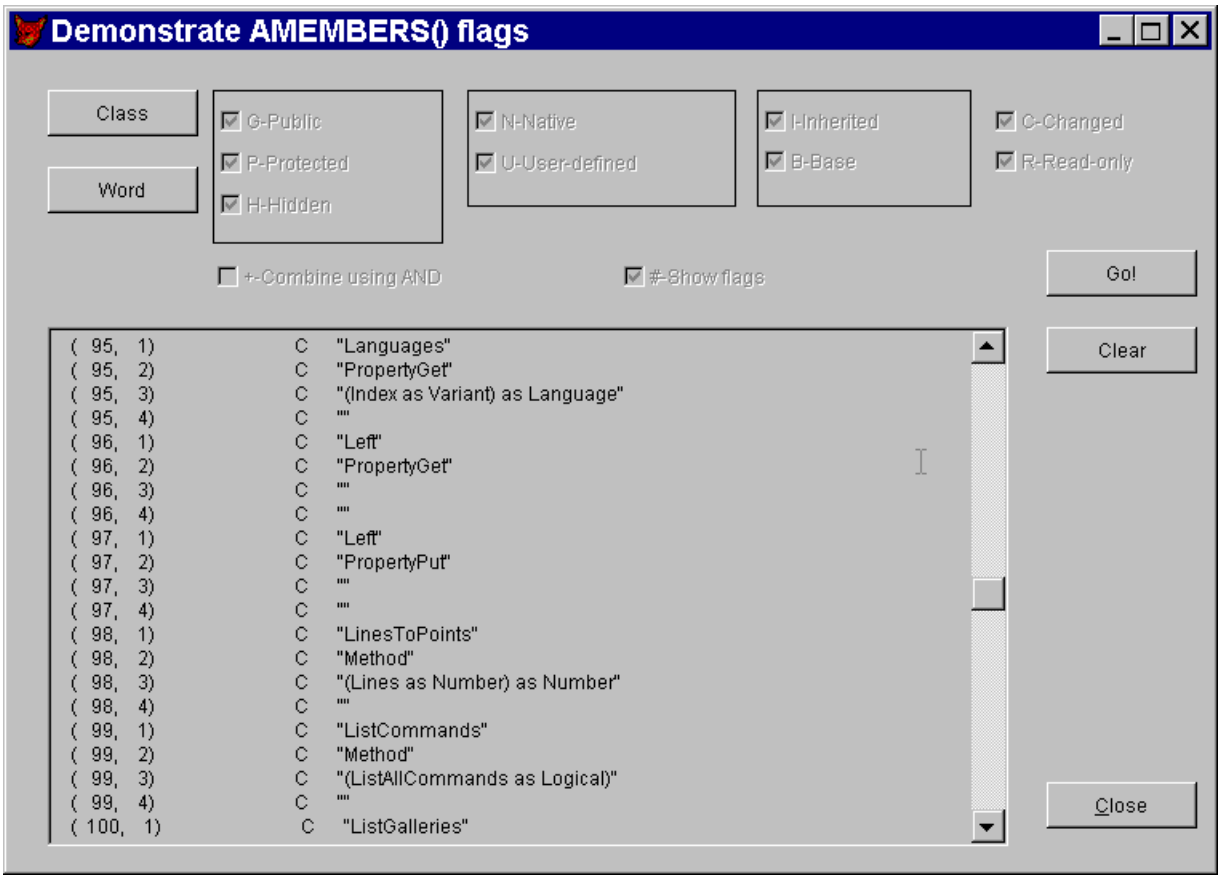

Figure 6 Applying AMembers() to COM objects – Here, the object passed to AMembers() is the Word Application object. The third column of the array contains the signature of the item. The fourth column is empty throughout – apparently, Word's creators didn't include help text.

Editing and debugging information

The final group of "A" functions helps with building editing tools and debugging applications. They are AProcInfo(), ALanguage(), AStackInfo(), and AError().

Supporting editor functions

New in VFP 7, AProcInfo() provides the same information about a program file that's available in the Document View. Document View, which is also new in VFP 7, gives you an extra window showing a list of sections (beginning of class definition, beginning of method or subprogram, each compiler directive) in the code in an editing window. You can click on an item in Document

View to change the cursor position in an editing window. An optional parameter to AProcInfo() determines exactly what information about the file is placed in the array. The syntax is:

```
nCount = AProcInfo( Array, cFileName [, nWhichInfo ] )
```

Table 9 shows the choices for nWhichInfo.

Table 9 Tell me about a program file – The optional, third parameter to AProcInfo() determines what information is put in the array.

Third parameter	Meaning
0 or omitted	Include all document view information, including preprocessor directives, class definitions, methods, functions and procedures. The array has four columns, with the name in the first, the line where it's defined in the second, the type of item in the third ("Define", "Directive", "Class", "Procedure"), and the amount of indentation in the fourth.
1	Include only class definitions. The array has four columns: the name, the line number, the parent class, and either the string "OLEPUBLIC" or the empty string.
2	Include only method information. The array has two columns: the method name, including the class name, and the line number.
3	Include only preprocessor directives. The array has three columns: the name, the line number, and the type ("Directive", "Define").

The information in the array can be used with the new EditSource() function to let you open a program to exactly the desired spot. For example, this code opens SCCText.PRG (which comes with VFP) to the definition for the SccTextEngine class:

```
LOCAL cFile, cClass, nClassCount, nRow

cFile = HOME() + "SCCText.PRG"
cClass = "SccTextEngine"
nClassCount = AProcInfo( aClassList, cFile, 1)
IF nClassCount > 0
   * Find the one we want in column 1;
   * return row, search is case-insensitive
   nRow = ASCAN( aClassList, cClass, -1, -1, 1, 9 )
   IF nRow > 0
      EditSource( cFile, aClassList[ nRow, 2 ] )
   ENDIF
ENDIF
```

It wouldn't take much work to turn that into a generic routine. Just change cFile and Class into parameters and add some error-checking and you have a routine that can open any PRG to any identifiable point. The form OpenPrg.SCX in the conference materials puts an interface to this process.

ALanguage() is another function that's designed to support a feature new to VFP 7. It fills an array with VFP language components. The second parameter determines which kind of components is grabbed. The syntax is:

```
nCount = ALanguage( Array, nType )
```

Table 10 shows the choices for nType.

Table 10 What's in the language? – The second parameter of ALanguage() determines which aspect of the language fills the array.

Second parameter	Meaning
1	Create a one-dimensional array of commands
2	Create a two-column array of functions. The first column contains the name and the second column contains a character string with the number of parameters accepted. The second column may also include the letter "M", which indicates that the function name cannot be shortened; the full function name must be used.
3	Create a one-dimensional array of base classes.
4	Create a one-dimensional array of DBC events.

This function was aided to help customize IntelliSense. If you want to design your own FoxCode table (that's the table that drives IntelliSense), you can use the data returned by ALanguage() as a starting point.

Getting debugging information

AStackInfo() is another function that's new in VFP 7. It fills an array with the program stack at the time it's called. The array has six columns, as shown in Table 11.

Table 11 Who's executing now? – AStackInfo() fills a six-column array with one row for each item on the program stack.

Column	Contents
1	Stack level, with 1 for the main program.
2	Name of the file containing the routine that's executing.
3	Name of the routine that's executing. Can be a procedure, function, or method. For a method, the entry includes the object name.
4	Name of the source file containing the routine.
5	Line number within the file (not within the routine).

6	Source code

One obvious choice for this information is saving it into a log file when an error occurs. But the information might also be used to open one or more of the programs at the point at which the function was called. For example, in an error handler, you might use code like this to open the failed program at the right point. (Presumably, you'd only do this in development mode.)

```
LOCAL ARRAY aProgStack[1]
LOCAL nStackSize

nStackSize = AStackInfo( aProgStack )
EditSource( aProgStack[ nStackSize - 1, 4], aProgStack[ nStackSize -1, 5] )
```

Arrays and COM

Two array-related features of VFP make working with COM easier. One helps you pass arrays among COM objects, while the other makes your COM objects better players.

COMArray() determines how arrays are passed to COM objects. There are two issues. The first is whether they're passed by value or passed by reference. The second is whether the array is zero-based or one-based. In FoxPro, arrays are one-based – that is, the first element is numbered 1. Many other languages number the first element 0. COMArray() gives you options for each combination of those attributes. In addition, in VFP 7, you can also specify that the array is of a fixed size and cannot be redimensioned by the called routine.

The syntax for COMArray() is:

```
nCurrentType = COMArray( oObject [, nNewType ] )
```

The first parameter is the object to which we want to pass arrays. If the second parameter is omitted, the function returns the current setting. If the second parameter is specified, the setting is changed and the function returns the new value.

To determine the value to pass for nNewType, you add three parts together (much as you do for the nFlags setting of ASCAN()). First, you determine whether the array is zero-based (0) or one-based (1). The next part is the passing style. Add 0 to pass by value or 10 to pass by reference. Finally (in VFP 7 only), if the array's size can be changed, add 0; if it has fixed size, add 100. So, there are 8 possible settings, as shown in Table 12.

Table 12 Pass the array, please – The second parameter to COMArray() determines how the specified object treats arrays passed as parameters. There are eight possible settings.

Second parameter	Meaning
0	Zero-based, passed by value, can be redimensioned.
1	One-based, passed by value, can be redimensioned.
10	Zero-based, passed by reference, can be redimensioned.

11	One-based, passed by reference, can be redimensioned.
100	Zero-based, passed by value, cannot be redimensioned.
101	One-based, passed by value, cannot be redimensioned.
110	Zero-based, passed by reference, cannot be redimensioned.
111	One-based, passed by reference, cannot be redimensioned.

To use this function properly, you need to determine what the particular COM object expects. The component's supplier should be able to provide that information.

There's various information that you may want to specify when you write a COM object yourself. This includes making some properties read-only or write-only and providing a Help string for the public members of the class. In VFP 7, you can create a special array of properties for a property. To do so, you dimension and populate an array called PropName_COMATTRIB, where PropName is the name of the property to which this array applies. (This mechanism is like the one used for Access and Assign methods.)

There are two ways to define COMATTRIB. It can be an array with four elements or it can be a scalar value. In the second case, you can specify only the flags that indicate things like read-only. You need the array version to specify the help string, capitalization and type. In either case, this technique is only available for classes written in code.

This example shows a very simple class that has a single custom property for which the whole set of attributes has been defined. (You can specify all the same information for methods, but only the attributes use COMATTRIB. The others have been incorporated into the syntax for defining a method. See the example.)

```
DEFINE CLASS cuscomtest AS custom OLEPUBLIC

   *-- The "Hello World" message
   chellomsg = "Hello World"
   Name = "cuscomtest"

   DIMENSION chellomsg_comattrib[4]
   chellomsg_comattrib[1] = 0x100000  && Read-only
   cHelloMsg_comattrib[2] = "Specifies the Hello World message"
   cHelloMsg_comattrib[3] = "cHelloMsg"
   cHelloMsg_comattrib[4] = "String"

   PROCEDURE Hello AS VOID HelpString "Displays the Hello World message"
   *-- Say "hello, world"
   RETURN This.cHelloMsg
   ENDPROC

ENDDEFINE
```

Once this class has been compiled into an EXE and registered, we can examine it in Office's Object Browser. Figure 7 shows the cHelloMsg property as it appears in the Object Browser.

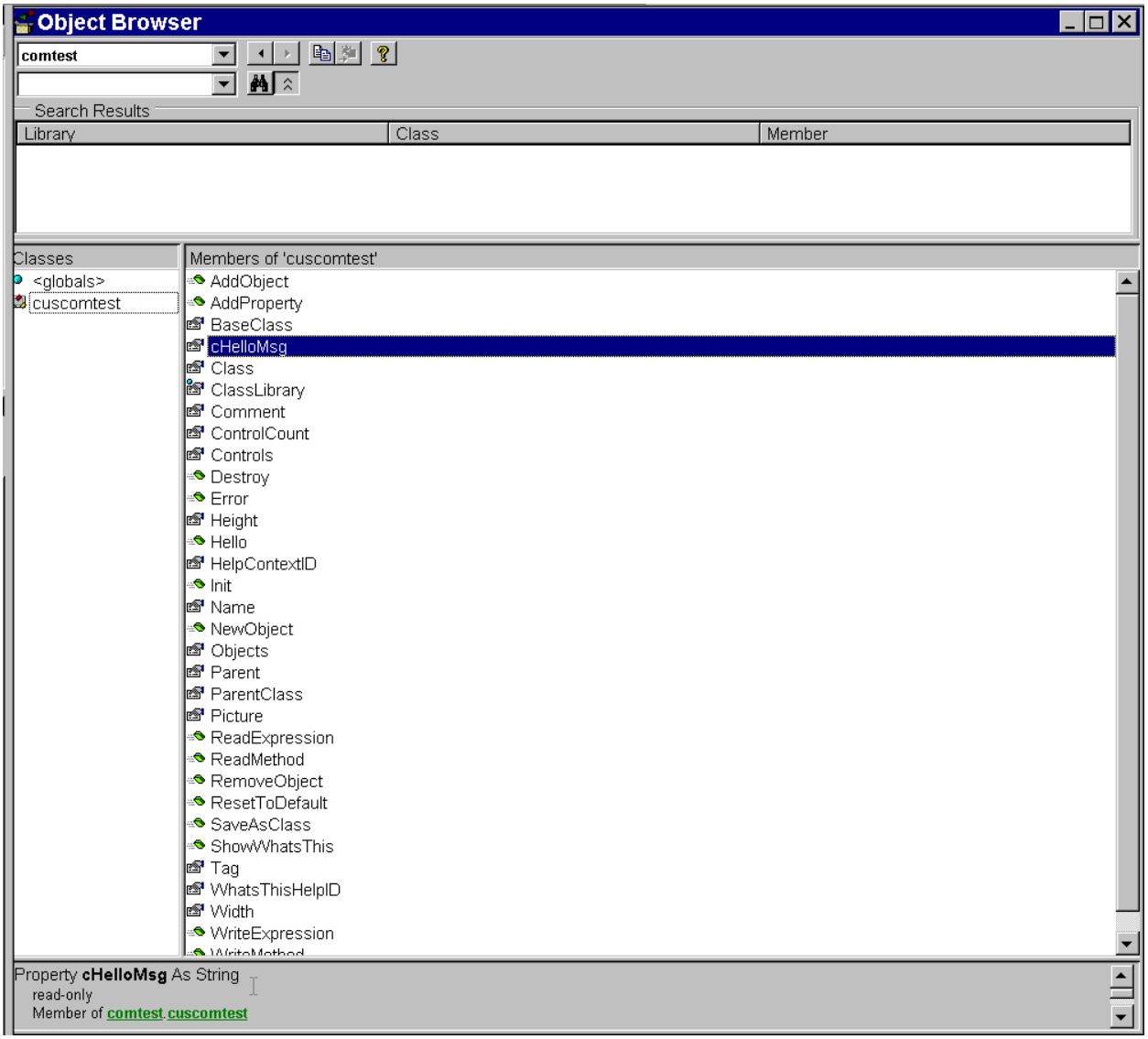

Figure 7 Specifying attributes –The attributes assigned to the cHelloMsg property using the special COMATTRIB array are visible in Office's Object Brower. Note also that both the property and the Hello method use the specified capitalization.

Wrapping up arrays

Visual FoxPro offers tremendous support for arrays and the large number of functions that put data into arrays makes it worth learning how to manipulate them. Properly used, arrays can be a valuable addition to the developer's toolkit.

Acknowlegements

Some of the examples and tables in these notes are drawn from *Hacker's Guide to Visual FoxPro 6.0* by Tamar E. Granor and Ted Roche (Hentzenwerke Publishing, 1998).