Clipper 5's super arrays

Data Based Advisor

November 01, 1992 | Gutierrez, Dan D.

For many years, xBase developers have lived without a way to collect and manipulate groups of data under a single, subscripted name. Arrays, as these groups are usually called, weren't part of the dBase language until dBASE IV. Early on however, Nantucket, Fox, and Wordtech were quick to include this sorely-missed feature in their products--a powerful selling point. The **Clipper** 5 language has taken even greater strides in giving developers tremendous flexibility with its new array capabilities. This month, we'll examine **Clipper** 5's so-called "ragged arrays," namely how to define an array, how to assign it values, how to manipulate it using standard operators and special built-in functions, and how code blocks can be applied to processing arrays.

Why are Clipper's arrays so special?

The overwhelming difference between **Clipper** 5 arrays and those of other xBase languages, is the flexibility that **Clipper**'s arrays give developers. **Clipper** arrays may assume any "topology" that makes sense for the problem being solved. This means that in **Clipper**, more than just the standard n-dimensional array can be represented. Arrays with all sorts of structures become possible. For example, consider the case where customer order-summary information is extracted from a table and needs to be represented in memory for easy manipulation. **Clipper** 5 arrays can handle this easily.

Pictorially it may appear as:

Customer base information

Name: Character
Contact: character
Address: character
Phone: character
Credit history: array of 3 elements
Invoice totals for 30 - 60 - 90 days
Current detail: array, unbounded dimension
Product Code: character
Quantity Ordered: numeric
Backorder: logical
Ship Date: date

With **Clipper**, representing this complex structure isn't a problem since each element of an array can be an array of multiple dimensions. **Clipper** doesn't "officially" have the ability to declare a data structure as in C, but with such a flexible array capability, these structures can be represented.

Unlike other xBase array structures, **Clipper**'s arrays can be virtually any size, although each array has a maximum of 4,096 elements. There's no limit to the number of sub-arrays that can be attached. The only real limit is the amount of memory available, but with the virtual memory manager (VMM) in **Clipper**, arrays can become quite large.

Declaring and using arrays

Clipper 5 arrays are easy to define. There's still the Summer'87 `DECLARE` statement which you can use with **Clipper** 5. The trend, however, has been to declare an array's name and dimension in a `LOCAL, STATIC, PUBLIC,` or `PRIVATE` declarative statement. Using `DECLARE` is the same as using `PRIVATE`. Here are some examples of array declarations:

```
LOCAL aStates := {}       // Null array, no elements
LOCAL aQtr{4}             // Empty array with
                          // 4 elements
LOCAL aMatrix{2,2}        // Two dimensional array
STATIC aPrices {5,5,5}    // Three dimensional array
```

The first example declares a null array. A null array is an array type variable, but it has no elements--the elements can be added dynamically. Always make sure you use the Hungarian notation prefix "a" for an array. The next three examples show that you can define an array of fixed dimensions if you know how many elements you need but aren't sure what to put in them. An array's size can be altered at runtime.

Assigning values to array elements is easy, too. For example, you can assign values to an array when you declare it:

```
LOCAL aMonths := { 'Jan', 'Feb', 'Mar', 'Apr', ; 'May', 'Jun', 'Jul', 'Aug', ; 'Sep', 'Oct',
'Nov', 'Dec', }
```

Once initialized, changing the values of array elements is easy:

```
aStates{1} := 'Alaska'   // Now, null array
                         // has one element.
aMonths{6} := 'Foo'      // 'Jun' has been
                         // replaced with
                         // 'Foo'
aPrices {2,3,4} := 25.67 // Comma subscripting:
                            // [2,3,4] is equivalent to
                            // bracket subscripting: [2] [3] [4]
```

Note: Multi-dimensional arrays in **Clipper** 5 aren't treated traditionally. For example, aMatrix has two elements and each of these is a sub-array containing two elements. This is a symmetrical topology, also possible in other xBase dialects. If you want to define the dimensions of a "ragged array," a definition like the one below is possible:

```
LOCAL aRagged := {1027.55, .T., {'a', 'b', {1, 2}}}
```

This array has three elements, aRagged[1] is a numeric, aRagged[2] is a logical, and aRagged[3] is a sub-array. Applying the LEN() function, which can be used with arrays as well as strings, we get an interesting result:

? LEN(aRagged) // Length is 3!

The length of aRagged is three because the elements in the sub-array aren't counted. You can extrapolate from this discussion that **Clipper** 5 arrays are completely flexible in shape. The type of information that can be stored in an array is also flexible:

```
? VALTYPE(aRagged)      // 'A'
? VALTYPE(aRagged[3])   // 'A'
? VALTYPE(aRagged[3,1]) // 'C' ? VALTYPE(aRagged[3] [3]) // 'A' ? VALTYPE(aRagged[2]) // 'L'
```

Array references can be generalized by using a subscript variable as in:

```
* Save integers 1..0 in aCount

LOCAL i, aCount [10]
FOR i :=1 TO 10 ; aCount [1] :=i ; NEXT
```

Here's one last example of assigning values to arrays:

```
LOCAL aFirst := {1,2,3}
LOCAL aSecond := {}
aSecond := aFirst // Will not result in 2 arrays
```

At first glance, it may seem that another copy of aFirst is generated and named aSecond. This isn't the case because in **Clipper** 5 an array's reference can be viewed separately from its value. In this case, aSecond "points" to the same memory locations as aFirst so:

```
? aFirst [2] := 'Hello world'
? aSecond [2] // "Hello word" is displayed.
```

To create a unique copy of aFirst, use the special built-in function, ACLONE().

Passing arrays as parameters

It's fine to pass an array to a UDF and return an array from a UDF. The only difference between arrays as parameters and regular memory variables is that arrays are passed by reference, unlike memvars which are normally passed by value.

This means that arrays can be altered inside the UDF. Single array elements, however, are passed by value. There's no way to pass a whole array by value and there's no way to pass a single element by reference. The following example shows these concepts:

```
FUNCTION main

LOCAL aArray := {1,2,3}
LOCAL nNum := 7

  myfunc (aArray)        // Pass array By Reference
  myfunc (nNum)          // Pass memvar By Value
  myfunc (@nNum)         // Pass memvar By Reference
  myfunc (aArray [1])    // Pass array element By
                         // value
  * myfunc (@aArray [1]) // Illegal,
                                  // compile-time error
RETURN NIL

FUNCTION mufunc (a)
a++ // Modify formal parameter
RETURN NIL
```

Array functions and AEVAL()

Clipper 5 has quite a compliment of built-in functions to support arrays. A list of these functions is presented in Table 1. Array functions can be grouped in terms of functionality: array manipulation, user interface, table, code block, and miscellaneous. As in the calling sequence descriptions, some functions accept arrays, some return them.

Here's a brief tour of several important array functions. Trace the flow of the program to see how the array values evolve. This is a good time to experiment with the **Clipper** 5 debugger. Hint: Control the execution of the code and check out the intermediate results.

```
LOCAL aSimple:= {},
      aNIL := { {NIL,NIL,NIL},{NIL,NIL,NIL} } // Think of aNIL as a traditional 2-by-3 matrix

? LEN(aSimple)           // 0 length
? LEN(aNIL)              // 2: since two sub-array elements

ASIZE(aSimple, 5)        // Dynamically dimension array
? LEN(aSimple)           // 5 empty elements
? aSimple[1]             // NIL:  value of empty elements

AADD(aSimple, '6th')     // Now {NIL,NIL,NIL, NIL,NIL,'6th'}
? LEN(aSimple)           // of length 6
aSimple[2]:=.T.          // Now {NIL,.T.,NIL, NIL,NIL, '6th'}

ADEL(aSimple, 1)         // Now {.T.,NIL,NIL, NIL,'6th',NIL}
                         // still with 6 elements!
                         // ADEL() gets rid of 1st element and brings in a
                         // NIL element from the right.  Length preserved.
ASIZE(aSimple, 5)        // This get rid of dangling NIL element
                         //  and re-dimension array.
AADD(aSimple,ARRAY(3))   // ARRAY() returns:{NIL,NIL,NIL}
                         // and it is added at the 6th element of aSimple,
                         // producing an
                         // "L" shaped topology.
? LEN(aSimple)           // Still 6

aSame := ACLONE(aSimple) // Duplicates aSimple's topology, i.e.
                         // aSame has the same "L" shape as aSimple.
AADD(aNIL, ARRAY(3))     // Now aNIL is 3-by-3, but empty
AFILL(aNIL, 'Foo')       // An attempt to fill
                         // all 9 elements
                         // produces unexpected
                         // results:
                         // {'Foo','Foo','Foo'}
                         // Remember, aNIL is an
                         // array with 3 elements!

aSimple[6] := {1, 2, 3}

? ATAIL(aSimple[2] )     // Displays 2, note that
                         // since ATAIL() returns
                         // an array in this case,
                         // it may be subscripted.
AINS( aNIL, 2 )          // {'Foo', NIL, 'Foo'}.
                         // Here length preserved,
                         // new 2nd element NIL is
                         // inserted, rightmost
                         // element is lost.
```

Take care when using array functions on multi-dimensional arrays because you may not get the results you expect. Remember that multi-dimensional arrays are just arrays of sub-arrays.

Continue experimenting with the other functions by writing test code and tracing the program flow in the debugger.

Arrays and code blocks

One of the most important and far reaching interdependency of features in the **Clipper** 5 language is the one between arrays and code blocks. Much **Clipper** code dealing with arrays can be optimized by using code blocks to process elements. Without code blocks, you'd need the usual WHILE/END or FOR/NEXT constructs. This is especially true of ragged arrays with their potentially strange configurations. Code blocks are specifically suited to traversing unconventional structures. In a general sense, code blocks can be called recursively to enable the navigation through any array structure. We'll explore this at the end of the section.

Let's consider a simple example of using a code block with an array:

```
LOCAL aNum := {1,2,3}, nSum := 0
AEVAL (aNum, {|elem| nSum += elem})
? nSum // 6: thanks to the code block
```

Reviewing the definition of AEVAL() in Table 1, you can see that aArray and bBlock are defined. Instead of assigning the code block to a memvar, it's simply passed as a parameter to AEVAL(). In this example, the block is used to "process" each element of aNum. Usually, when using code blocks with arrays, the code block requires a parameter. Here, the formal parameter is ELEM. Each element of aNum is passed to the block via ELEM. Inside the block, the parameter is operated on. In this case, nSum collects the sum of the elements.

The trick in applying code blocks to arrays is to recognize where and how to include them. Consider :

```
#include "directry.ch"
LOCAL aFiles := DIRECTORY ("*. PRG")
AEVAL (aFiles, {|elem| QOUT( elem[F[_NAME] ) } )
```

The DIRECTORY() function yields an array of sub-arrays, each containing information about disk files referenced in the filespec parameter, "*.PRG". The DIRECTRY.CH include file, found in the standard **Clipper** 5 include directory, contains manifest constants which can be used as subscripts for the array returned by DIRECTORY(). In this case, F_NAME is a 1, and indicating the first element of each sub-array, which is the file name. Take a peek at DIRECTRY.CH to see the other subscripts.

Each element of aFiles, namely each of the sub-arrays, is passed to the code block via ELEM. QOUT() then prints the first element, or file name of each sub-array.

When first faced with writing the code to solve this problem, a loop seems appropriate. A looping construct could be used, but AEVAL() and a code block is a much more optimized solution.

As a final, somewhat esoteric, example of array access: What if you need to print each element of an array no matter what the topology? No problem! The solution involves a recursive function call (where a function calls itself) inside a code block evaluation.

```
FUNCTION main
   LOCAL aRagged := {1,1,1, {2,2, {3, {4,4} } } }
   PrintRagged( aRagged )
RETURN NIL

FUNCTION PrintRagged( aArray )
   * If a sub-array is encountered, dive deeper
   * into the recursion, otherwise just displaythe element's value.
   AEVAL ( aArray, {|elem| IFF ( VALTYPE ( elem )== “A”, ;
                                 PrintRagged( elem ) ,;
                                 QOUT( elem ) ) } )
RETURN NIL
```

The output of this program is simply a list of numerics: 1, 1, 1, 2, 2, 3, 4, 4. PrintRagged() visits each element of each sub-array in aRagged and displays each value. Notice the call to PrintRagged() from within PrintRagged(). This makes the function recursive. At first, recursion is difficult to conceptualize. It's like believing in infinity. Recursion is the only way to traverse a completely flexible array structure.

Code blocks can also be stored in arrays. This results in some curious applications:

```
LOCAL i, aJumpTable : { { || gl() }, ;
                        { || ar() }, ;
                        { || ap() }, ;
                        { || so() } }
FOR i := 1 TO 4
   EVAL ( aJumpTable[ i ] ) // Visit each UDF
NEXT
```

In this example, a series of code blocks are stored in an array. The EVAL() function is used to evaluate each array element, thus calling the named UDF.

Conclusion

This month we've seen that **Clipper** 5's arrays are quite generous in the features they provide to developers. They're truly a pleasure to use.

But still, we've only scratched the surface. In the future, I'll show you how the dynamic pair--arrays and code blocks--team up to form a powerful combination.

Dan D. Gutierrez is the President of AMULET Consulting, a Los Angles-based xBase development firm. He teaches Xbase and **Clipper** programming at UCLA and is the co-author of dBase IV: Beyond the **Basics**. He is currently writing **Clipper** 5: Step By Step, and provind **Clipper** 5 training in the L.A. area. Dan can be reached at (301)479-4877 or on CompuServer (73317,646).