by George Defenbaugh

C DLL Extends VB Array Sorting

ROLL YOUR OWN VISUAL BASIC 2-D ARRAY SORTING DLL USING THESE C TECHNIQUES.

Recently I was surprised to find that Visual Basic programmers often write their own sort routines. While some languages provide standard routines for sorting, such as C's qsort( ), Basic programmers use vendor sample code or examples from books (like Gary Cornell's book: The Visual Basic 3 for Windows Handbook). Since I'm cursed with an attraction to the C language, I often incorporate C programs and routines into my personal programming projects. In fact, at our company, we actively look for opportunities to develop C DLLs whenever possible because they can be used with various 4GL tools such as Visual Basic, ObjectView and PowerBuilder. This approach allows us to develop reusable software in and for the Windows environment. Our strategy is very similar to that explained in the recent article on mixing C and Visual Basic ("Mix C and VB for Maximum Performance and Productivity," Visual Basic Programmer's Journal, Aug/Sept '93, Vol. 3, #5, pg. 20). And because VB has no built-in arraying sorting routines, it was a natural opportunity to do the job with a C DLL.

DLLs provide a language-independent way to build components for Window's applications. They can provide enterprise-standard functions that will work with any program, regardless of the language used by the caller. DLLs can also include specialized functions that work only with programs developed in a particular language. The DLL will make use of specific functions provided by VB Professional's Control Development Kit, so it won't be usable with applications written in other products like PowerBuilder or ObjectView. The VBPro CDK functions are explained in the Professional Features Book 1 and are not provided as part of the Personal edition of VB.

If you're the type who likes to jump immediately to the code, you can find the entire DLL in Listings 1, 2, and 3. Listing 1 is the DLL header file and includes preprocessor directives and function prototypes. Listing 2 is the DLL itself and contains code for the LibMain( ), WEP( ) and vbSortArray( ) functions. Listing 3 is the module definition file and is shown to emphasize the importance of selecting the correct name on the LIBRARY parameter. In the end, it turned out to be a fairly short DLL, but that wasn't intuitively obvious at first.

The first hurdle to overcome was learning how VB stores array elements in memory. Wow! Fortran programmers will feel right at home, but for this C enthusiast, it was quite a shock. How dare they store them backwards and use column-major rather than row-major order. But, to get everyone on the same level playing field, let's review some properties of arrays.

First, array elements are always stored contiguously in memory—no extra space is allowed between any two elements. This is necessary because an array subscript is multiplied by the element size to generate the location of a particular element. Second, array elements are always the same size: a requirement for the multiplication scheme to work. Otherwise you'd need another array of locations to locate elements in the first array!

Finally, having all the elements the same size also forces the rows and columns of a multidimensional array to be the same size as other rows or columns, respectively. This allows elements in any row or column to be easily located by adding the number of intervening rows or columns multiplied by the row/column size, to the location value (pointer) in the particular row or column. What, then, is this row-major versus column-major order thing? This table helps clarify the difference:

	Column:	1 2
2-D Array:	Row 2:	c d
	Row 1:	a b

Row-major order: a b c d
Column-major order: a c b d (used by VB)

Whether VB uses row- or column-major order is usually immaterial to the VB programmer. It's really only important to know that VB, like Fortran, uses column-major order when you want to manipulate the memory locations with another tool, such as C. If row-major order was used, then rows could be exchanged by moving blocks of memory the size of an entire row. For example, rows 1 and 2 could be exchanged by picking up elements "a b" and swapping them with "c d." Because column-major order is used, individual elements must be exchanged one by one. This means that you must swap "a" with "c," then "b" with "d."

HOW THE DLL KNOWS

The final question is how the C-language DLL will become aware of the format and size of the array. The calling program could explicitly tell the DLL all the necessary information (starting location in memory, number of dimensions, bounds of each dimension, and size of an element), or the DLL could discover the information for itself. The former approach would be necessary if the DLL was destined to be used by different languages. The latter can be used if VB-specific functions are available to reveal the information—and they are.

Using VB-specific functions allows a DLL to hook into VB itself and directly manipulate or investigate VB data elements. These functions are provided as part of the CDK, and can be used to build either DLLs or

George Defenbaugh is a project manager for Amerada Hess Corporation in Tulsa, Oklahoma. He is currently working to implement client/server technology for applications used in the natural gas industry. He has written a college-level textbook about C, and teaches Advanced C, Windows Programming in C and Visual Basic classes part-time at Tulsa Junior College. Reach him at 918-599-4791 or via CompuServe at 71004,413.

custom controls. Use them to reduce the number of arguments required by the DLL to one: the array descriptor (the other argument, nDebug, can always be zero without affecting the DLL's function). Four separate functions allow you to obtain information about the array. In addition to the manuals, these functions are also documented in the VBAPI.HLP file provided with VBPro.

The first function we use is VBArrayBounds, which is called once for each dimension and returns the number of rows or columns. The column boundary is dimension one; the row boundary is dimension two. The high and low boundaries are both stored in a long integer and HIBOUND and LOBOUND are used to extract the separate values. The low boundary is subtracted from the high, and one is added to yield the number of rows or columns.

The second is VBArrayElemSize, which returns the size of an element. Our DLL only supports integer arrays, so this function is primarily for illustration. You'll use it to calculate the height of a column, but could easily have used sizeof(int) instead.

The third function is VBArrayIndexCount, which reveals the number of dimensions. This DLL supports one or two, and a one-dimension array becomes a specialized version of a 2-D array.

Last comes VBArrayFirstElem, which returns a pointer to the first element of the array. That pointer tells the C DLL where the array begins in memory. All other elements fall in line immediately after the first, with column one appearing first, column two next, and so on.

The DLL supports only integer arrays. Handling arrays of other types requires duplicating the core portion of the code that does the sorting. The problem with writing a general-purpose C routine that handles any data type is that C, unlike VB, requires variables to be defined before use with a particular data type. A variable's data type can be temporarily changed on the fly, but it can't be permanently changed while the program is executing. Consequently, a general-purpose sort routine for arrays of different types would really be a series of special-purpose routines, each handling a particular type. Separate pointers would be needed for each data type, and the choice of which pointer to use would be determined by an argument passed to the DLL.

One useful thing about pointers in C is that pointer arithmetic is scaled to the size of the item pointed to: if a pointer references one integer or element of the array, adding one to the pointer causes it to point to the next integer or element of the array. Likewise, if there are five rows in the 2-D array, and you start with a pointer to the very first element, you can add five to the pointer to make it point to the second element of the first row. Remember that the array is laid out in column-major order, so the entire first column occurs before the second element of the first row, which starts the second column.

USE A SIMPLE RIPPLE SORT

```
// sortary.h
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "vbapi.h"
// bas prototypes for VB specific functions

// Defined constants
#define MAX_MSG_SIZE                          255
// max size of a message
#define MAX_NBR_DIMENSIONS                    2
// nothing over 2D
#define RTN_OK                                0
#define RTN_ERR_TOO_MANY_DIMENSIONS           -1
#define RTN_SORT_CANCELED                     -2

// Function prototypes
int CALLBACK _export      vbSortArray
                  (HAND badArray, int nDebug);
```

Include a header file, SORTARY.H, contains include files, defined constants, and function prototypes. While a header file isn't absolutely required, it's a good programming practice. Defined constants avoid the problems associated with hard coding numeric constants in the program.

```
//---------------------------------------------
// vbSortArray ()  Sorts a VB integer array of
//  one or two dimensions
//---------------------------------------------
#include "sortary.h"
int FAR PASCAL LibMain (HINSTANCE hInstance,
    WORD wDataSeg, WORD wHeapSize, LPSTR lpszCmdLine)
{
 if (wHeapSize > 0)
  UnlockData (0);
 return TRUE;
}
// LibMain
int CALLBACK WEP (int nParameter)
{
 return TRUE;
}
// WEP

int CALLBACK _export vbSortArray
                  (HAND badArray, int nDebug)
{
 LONG      lRowRange;
 // Holds lower and upper row boundaries
 LONG      lColRange;
 // Holds lower and upper column boundaries
 int       nNbrRows, nNbrCols;
 // Calculated number of each
 int       nCntCols, nCntRows;
 // Temporary counters for loops
 USHORT    usAryElemSize;
 // Size returned from VB
 SHORT     sNbrAryIdxs;
 // Number returned from VB
 LPVOID    lpvFirstArrayElem;
 // Location of array in memory
 int       * lpnLoRow, * lpnHiRow;
 // Row pointers for sort
 int       * lpnLoRowStop, * lpnHiRowStop;
 // Last rows to compare
 int       * lpnLoRowElem, * lpnHiRowElem;
 // Used to scan rows
 // Get the bounds of the array and all
 //  relevant sizes
 lColRange = VBArrayBounds (badArray, 1);
 // column boundary
 lRowRange = VBArrayBounds (badArray, 2);
 // row boundary
 nNbrCols = HIBOUND (lColRange)
                - LOBOUND (lColRange) + 1;
 nNbrRows = HIBOUND (lRowRange)
                - LOBOUND (lRowRange) + 1;
 usAryElemSize = VBArrayElemSize (badArray);
 sNbrAryIdxs = VBArrayIndexCount (badArray);
 lpvFirstArrayElem = VBArrayFirstElem (badArray);
 if (nDebug)
 // This approach carries debug code into production
 {
    char  szTemp[MAX_MSG_SIZE];
    wsprintf (szTemp, "Number cols: %d\nNumber rows: %d\n"
                "Element size: %d\nNumber indexes: %d",
          nNbrCols, nNbrRows,
                usAryElemSize, sNbrAryIdxs);
    MessageBox (NULL, szTemp,
                "Characteristics of the Array",
                MB_OK | MB_TASKMODAL);
 }
 if (sNbrAryIdxs > MAX_NBR_DIMENSIONS)
 // Can't handle > 2D
 {
    char  szTemp[MAX_MSG_SIZE];
    wsprintf (szTemp, "Array has %d
                dimensions; only %d allowed",
                sNbrAryIdxs,
 MAX_NBR_DIMENSIONS);
    MessageBox (NULL, szTemp, "vbSortArray Error",
                MB_OK | MB_TASKMODAL);
    return (RTN_ERR_TOO_MANY_DIMENSIONS);
 }
 // Uses a Ripple sort - nothing fancy!
```

LISTING 2. CONTINUED IN NEXT COLUMN.

Build your C DLL. This is essentially the entire program. Hungarian notation is used to help form each variable's prefix, and meaningful names help the code explain itself. After the variables are defined, VB specific functions are used to obtain information about the 2-D array. That information is used by various pointers, which search through the rows and columns.

LISTING 2: CONTINUED FROM PREVIOUS COLUMN.

```
    lpnLoRow = lpvFirstArrayElem;
    // Start out on first row
    lpnLoRowStop = lpnLoRow + nNbrRows - 2;
    // Set stopping points
    lpnHiRowStop = lpnLoRow + nNbrRows - 1;
    // lpnLoRow 'sits' on a row while
    // lpnHiRow searches the column
    for (lpnLoRow <= lpnLoRowStop; ++lpnLoRow)
    // No initial value
    {
        // loop through remaining rows to
        // compare against lpnLoRow
        for (lpnHiRow = lpnLoRow + 1;
        // Initial value
                lpnHiRow <= lpnHiRowStop;
        // Test condition
                ++lpnHiRow)
    // Increment expression
        {
            lpnLoRowElem = lpnLoRow;
            // Start out with first column
            lpnHiRowElem = lpnHiRow;
            // loop through elements in a
            // row (columns) to do comparison
            for (nCntCols = 0;
            // Initial value
                    nCntCols < nNbrCols;
                    // Test condition
                    ++nCntCols
                    // Increment expressions
                        lpnLoRowElem += nNbrRows
                            lpnHiRowElem += nNbrRows)
            {
                if (*lpnLoRowElem < *lpnHiRowElem)
                    break;
                    // Leave loop if already in order
                if (*lpnLoRowElem == *lpnHiRowElem)
                    continue;
                    // Go to next column if values equal
                // first 'hit' causes row to be exchanged
                if (*lpnLoRowElem > *lpnHiRowElem)
                // Ascending sort
                {
                    // loop back through elements in
                        // row to do exchange
                    lpnLoRowElem = lpnLoRow;
                    // Start back with first col
                    lpnHiRowElem = lpnHiRow;
                    for (nCntCols = 0;
                    // Initial value
                            nCntCols < nNbrCols;
                            // Test condition
                            ++nCntCols
                            // Increment expressions
                            lpnLoRowElem += nNbrRows
                                lpnHiRowElem += nNbrRows)
                    {
                        int nTmp;
                        // Need place to use for swapping
                        nTmp = *lpnLoRowElem;
                        // Save lo row element
                        *lpnLoRowElem = *lpnHiRowElem;
                        // Move hi row
                        *lpnHiRowElem = nTmp;
                        // Restore lo row element
                        if (nDebug)
                        {
                            int     nMsgBoxRtn;
                            char    szTemp[MAX_MSG_SIZE];
                            wsprintf (szTemp, "New Row -
                                    #%d - Col #%d values:"
"\nLow row: %d, Hi row: %d",
nCntRows,
nCntCols,
*lpnLoRowElem,
*lpnHiRowElem);
                            nMsgBoxRtn = MessageBox
(NULL, szTemp, "Elements of a Swap",
MB_OKCANCEL | MB_TASKMODAL);
                            if (nMsgBoxRtn == IDCANCEL)
                                return (RTN_SORT_CANCELED);
                        }   // end of if debug
                    }   // end of for loop to do exchange
                    break;
                    // terminates loop through columns
                }   // end of if that detected
                        // out of order condition
            }// end of for loop through columns
        }   // end of inner for loop - controls lpnHiRow
    }   // end of outer for loop - controls lpnLoRow
    return (RTN_OK);
}// vbSortArray
```

A ripple sort is used in this DLL; while most other sort algorithms are faster, they are also more complicated. Using a ripple allows you to focus on the interface between C and VB.

The basic idea of the ripple sort is to anchor yourself to one row, then ripple through the rest of the rows looking for any that are smaller than the anchor row. Each time you find one, an exchange is done. After all the rows have been searched, the anchor row will contain the smallest of all rows examined. The process is repeated by moving to the next row and designating it as the anchor row. Each cycle in the process has to examine one less row, and the next to the last row is the last one to serve as an anchor.

Since you're sorting a 2-D array, the comparison process is a little more involved than when sorting only one dimension; you must loop through each element of the two rows being examined. In memory, each element in a row is separated from adjacent elements in the same row by the height of a column of elements (the number of rows). Looping through the elements in a row means you add the number of rows to a pointer each time.

The DLL has two arguments. The first is hadArray, or "handle to array descriptor." Through it the DLL can discover all about the array by using the functions mentioned above. The other argument is a flag that controls whether debugging information is displayed by the DLL. This information can be crucial during initial development. The approach also allows a runaway sort process to be terminated. The second MessageBox includes the MB_OKCANCEL style option with a subsequent test of the MessageBox return value. If IDCANCEL is returned, the DLL terminates.

Including a debug flag in the DLL's argument list requires that the debugging code be carried into production, rather than removed before distribution. This might not be a good approach to use for a commercial product, but can be helpful in a corporate or personal environment where the programmer has close contact with the developers who use the DLL. This allows developers to include features in their applications that can turn the debugging option on when needed or requested, such as while the user is on the phone with a support group or assistance center.

BUILDING THE DLL

At this point, I've covered everything except to note special files required to build the DLL. With VBPro, both of these files will be found in the \CDK directory of a standard installation. First is VBAPI.H. In Listing 1, it's referenced in a #include directive and provides VB-specific definitions to the DLL. This includes function prototypes for the functions VBArrayBounds, VBArrayElemSize, VBArrayIndexCount and VBArrayFirstElem.

Second is VBAPI.LIB, the import library containing the linkage that enables the DLL to find the VB-specific functions I just mentioned. The file has to be identified to the linker, either in the list of libraries if using a make file, or else in the list of project files if an integrated development environment is used.

Third, the DLL itself is shown in Listing 2. The LibMain() and WEP() functions are the same as those found in almost any simple DLL. The function of interest and the one called by the application is vbSortArray(

```
LIBRARY         SortArv
DESCRIPTION     'Array Sort Subroutine for VB
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'
CODE            LOADONCALL MOVEABLE DISCARDABLE
DATA            LOADONCALL MOVEABLE DISCARDABLE SINGLE
HEAPSIZE            1024
```

Match your names. This .DEF file establishes how the program will be organized. The LIBRARY name must match the file name or VB won't be able to use the DLL. LOADONCALL defers the DLL load until it's needed, and WINSTUB.EXE provides a message to the user if the program is accidently run from DOS rather than from within Windows.

). It can be viewed in four parts. The first ten lines are where the local variables are defined. The next eight lines make use of the VB-specific functions to obtain information about the format and size of the array. Then follow two if() statement blocks that perform debug processing and error checking. The ripple sort code brings up the rear in a moderately long for() loop.

Fourth, the module definition or .DEF file is shown in Listing 3. As I've already mentioned, it's included to emphasize how the internal name of the DLL must match the external name. Otherwise, VB won't be able to locate the subroutine. This is a feature peculiar to VB and some other 4GLs, and it doesn't occur when the DLL is accessed by a C-language calling program.

Finally, a test driver must be constructed and should remain available during the life of the DLL, to be used whenever the DLL is changed. One idea for a driver to test vbSortArray is to use the Grid control to provide a visual representation of a 2-D array. The Grid control can be loaded using the Rnd function, and a separate form can allow specific cells to be "plugged" with user-supplied values. Plugging is necessary so that rows with identical values in a particular column can be tested, to ensure that the sort isn't just ordering the rows based on the first column. The VB code needed to identify the function in the DLL is fairly simple:

```
Declare Function vbSortArray Lib "sorary.dll"
    (gnAry() As Integer, ByVal nDebug As Integer)
    As Integer
```

Note how the array argument is defined, and that it doesn't include the ByVal keyword. However, the nDebug argument uses ByVal so that it's passed by value and not by reference.

One final note that will be very helpful to some readers. Simultaneous development of the VB test driver code and the DLL is easy to arrange. VB can be running at the same time as an integrated development environment or a programmable editor like Codewright. The developer can bounce back and forth between changing the DLL and changing the VB test code. However, it's very easy to put VB into debug mode and switch back to the C development environment without actually stopping the VB program.

If SHARE is used, and the DLL is rebuilt while it's still in use by the VB test program, disaster can strike. It's likely you'll have to restart DOS—not just Windows—and any unsaved files will be lost. So save all files before testing any code, and stop all programs before starting any compiles. n