

Working with Array Functions and DLLs in Excel VBA

AVt, Oct 2005

1. Arrays	2
2. Calling DLL Functions with an Array as Argument	4
3. Returning an Array to Excel.....	6
4. Using VBA Functions in a DLL: callback.....	8
5. An Advanced Example.....	9
6. Global Arrays in VBA	11
7. Callback for Functions with Vector Arguments.....	14
7.1 A Simple Way	15
7.2 A General Solution.....	17
8. Application Examples.....	20
8.1 Parametric Integration.....	20
8.2 Least Square Fitting.....	22
Appendix: The True and Lazy Way.....	25

I always hated what I have seen about working with arrays, Excel and DLLs.

So I wrote up what seems to be necessary for me: these are several commented examples which should make it clearer (for me) and are thought to be used as recipes for actual coding problems.

The examples are covered by an Excel worksheet and C source code for the DLL.

Remember: For compiling a DLL the option "`__stdcall`" has to be used and functions to be exported need an additional export file named `*.def` (no, I do not want to write about generating a DLL ...).

A general remark to prevent undesired automatics and prevent crashes while using DLLs (and it is almost a must for working with callbacks):

- always declare types of variables in function arguments,
- use an explicit calling convention `ByVal` or `ByRef` for them (arrays are called by reference within VBA),
- use explicit return types for functions

Use `long` instead of `integer` when working with DLLs (to avoid different byte length in C and VBA).

Hm ... and even if I use global variables here: try to avoid them. But if you can not resist, then do not have too much of them ...

Conventions:

- all array indices start at 1,
- all arrays are of type double and are column vectors in n-space \mathbb{R}^n (i.e. `array(i) = array(i,1)` and all the arrays here are numerical ones)

So usually any VBA module starts with

```
Option Explicit
Option Base 1
```

All functions exported from the DLL have names ending with "`_DLL`" in Excel (a naming convention to make things easier to read).

1. Arrays

Let us first look at arrays in Excel and how to use them in VBA.

The following creates an array of desired length (and at least of length 1)

```
Function createArray( _
    ByVal nLength As Long) As Variant
Dim arr() As Double ' do not use a fixed dimension

If nLength < 1 Then
    nLength = 1
End If
ReDim arr(nLength)
createArray = arr
End Function
```

Having an array it may be printed to the debug window (limited to the first 100 entries and returning the number of items printed). Note the calling convention when calling that function with an array.

```
Function printArray( _
    ByRef arr() As Double) As Long
Dim i As Long, iMax As Long

iMax = UBound(arr) ' this works for arrays
If (100 < iMax) Then iMax = 100

For i = 1 To iMax
    Debug.Print arr(i)
Next
printArray = i - 1
End Function
```

As short hand I want some function to fill an array by positive natural numbers starting from 1 (since arrays contain zeros only after initialisation):

```
Function fillArray( _
    ByRef arr() As Double) As Long
' fill array with the natural numbers starting from 1
' return number of items
' a variation would be: Function fillArray(ByRef arr) As Long
Dim i As Long

For i = 1 To UBound(arr)
    arr(i) = i
Next
fillArray = i - 1
End Function
```

A simple numerical function giving the sum of the entries:

```
Function fct_Example1(ByRef arr() As Double) As Double
' example: sum up entries
Dim i As Long
Dim s As Double

s = 0
For i = 1 To UBound(arr)
    s = s + arr(i)
Next

fct_Example1 = s
End Function
```

Now play with that.

If the array is not initialised one can not use UBound or catch errors in a reasonable way. But a simple solution is to use `ReDim array(1)` and then everything works.

```
Sub tst_notInitialized()
Dim kDummy As Long
Dim p() As Double
Dim q() As Double
Debug.Print "---"

Debug.Print IsError(p)
'Debug.Print IsNumeric(UBound(p))
'Debug.Print IsError(p(1))

Debug.Print "-"
ReDim p(1)
Debug.Print IsError(p(1))
Debug.Print IsNumeric(UBound(p))
kDummy = printArray(p)

Debug.Print "-"
q = createArray(1)
Debug.Print IsError(q(1))
Debug.Print IsNumeric(UBound(q))
kDummy = printArray(q)
End Sub
```

So create an array (here of length 1): then its VBA type is 8197 - an array of type double, Excel recognizes it as array and it always has at least 1 element.

```
Sub tstType_createArray()
' check the return type for a created array of length n=2

Debug.Print "---"
Debug.Print VarType(createArray(1)) ' 8197 = array of type double
Debug.Print IsArray(createArray(1)) ' true
Debug.Print createArray(1)(1)      ' returns the 1st element (always
exists)
End Sub
```

More explicitly: create an array of length $n = 4$, fill it with natural numbers, print it to the debug window and sum up its elements (which should give $n * (n + 1) / 2$ as value) using `fct_Example1`:

```
Sub tst_createArray()
' show the handling: create an array of length n=4,
' fill it with natural numbers, print it and sum them up
' using fct_Example1

Dim arr() As Double ' array to be created
Dim n As Long      ' length of that array
Dim k As Long
Debug.Print "---"

n = 4
arr = createArray(n) ' create array

k = fillArray(arr)   ' populate it with positive natural number
k = printArray(arr)  ' print each item
                        ' and now apply a function taking it as argument:
Debug.Print "summing up = " & fct_Example1(arr)
Debug.Print "n*(n+1)/2 = " & n * (n + 1) / 2
End Sub
```

2. Calling DLL Functions with an Array as Argument

I want to use the following function (summing up an array and multiply that by an extra argument x)

```
double __stdcall sumUp(
    double x,
    double arr[],
    long nLength )
{
    int i=0;
    double s=0;

    for(i=0;i<nLength;i++)
    {
        s = s + arr[i];
    }

    return x * s;
}
```

Remember that array start with index 0 in C (how ugly ... even it is called the offset)!

For that the function has to be "declared" to Excel. Note that the calling conventions have to be declared explicitly (always!) and that the array has to be handed over by reference

```
Declare Function sumUp_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "sumUp" ( _
    ByVal x As Double, _
    ByRef arr As Double, _
    ByVal nLength As Long) As Double
```

Calling is done by using the first array element (do not use a variable for that, do it this way):

```
Sub tst_arrayArguments()
' having an array as argument call a DLL
' sum up the array and multiply by x
Dim x As Double
Dim n As Long, k As Long
Dim arr() As Double ' do not use a fixed dimension
Debug.Print "----"

n = 5
arr = createArray(n)
k = fillArray(arr)
x = 2

' call the DLL by reference using the first array element
Debug.Print "DLL result = " & sumUp_DLL(x, arr(1), n)
Debug.Print "VBA x*Sum = " & x * fct_Example1(arr)
End Sub
```

Of course computing in VBA directly with `x * fct_Example1(arr)` gives the same result.

Working with Array Functions and DLLs in Excel VBA

Calling by reference means: the DLL does not work with values or a copy of the array - it gets the original array and has access to that memory space. This can be used to alter it and to pass values!

The following function receives an array, sets its 2nd (!) entry to 1.1 (to be held through some global variable) and returns $x * 3$ rd entry (so use an array of length 3 at least):

```
double g_arr[10] = {0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};

double __stdcall fromVB(
    double* arr,
    double x,
    long nLength)
{
    if (nLength < 3){
        return -1;}

    arr[1] = g_arr[1];
    return x*g_arr[2];
}
```

In Excel we call it as follows:

```
Declare Function fromVB_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "fromVB" ( _
    ByRef arr As Double, _
    ByVal x As Double, _
    ByVal nLength As Long) As Double
```

And a test confirms the stated behaviour:

```
Function handArrayToDLL( _
    ByRef arr() As Double, _
    ByVal x As Double, _
    ByVal nLength As Long) As Double
' to show an alternative input (since arr(1) is not common in VBA)
' sets arr(2) to 1.1 and returns x * 2.2
handArrayToDLL = fromVB_DLL(arr(1), x, nLength)
End Function
```

```
Sub tst_handArrayToDLL()
Dim x As Double
Dim n As Long, k As Long
Dim arr() As Double ' do not use a fixed dimension
Debug.Print "----"

x = 2

n = 3
arr = createArray(n)
k = fillArray(arr) ' natural numbers
Debug.Print "arr(2) = " & arr(2)

' call the DLL by reference using the first array element
Debug.Print "return = " & handArrayToDLL(arr, x, n)
Debug.Print "arr(2) = " & arr(2)
End Sub
```

3. Returning an Array to Excel

Typical task: having an array in a DLL send it to Excel where the array is thought to be hold in a C function. Usually this is done through safearrays (which I hate and it is not need if using Excel as the main system). But there is another way since the above behaviour can be used systematically to retrieve arrays from Excel:

provide space from Excel, hand it to the DLL and update it there to have it in Excel.

The following function holds an array of 10 descending numbers and writes them to the space which it gets as argument:

```
long __stdcall fctHoldingArray(
    double arr[],
    long nLength )
{
    double arrLocal[10] = {10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0};
    long nLocal = 10;
    long i=0;

    for(i=0;i< min(nLength,nLocal);i++)
    {
        arr[i] = arrLocal[i];
    }

    return i;
}
```

In Excel we declare it

```
Declare Function fctHoldingArray_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "fctHoldingArray" ( _
    ByRef arr As Double, _
    ByVal nLength As Long) As Long
```

and check that:

```
Sub tst_fetchArrayFromDLL()
' uses fctHoldingArray, holding an array of length 10
' of descending numbers 10, 9, 8, ... , 1.
' Now fetch the first n elements to have them in Excel:
' For that create an array and send it for update in place.

Dim nLength As Long, outLength As Long
Dim arr() As Double ' do not use a fixed dimension
Dim k As Long
Debug.Print "----"

nLength = 3
arr = createArray(nLength)

outLength = fctHoldingArray_DLL(arr(1), nLength)
k = printArray(arr)

End Sub
```

We expect the first n=3 elements from the DLL and will see 10, 9, 8 ... as desired.

Working with Array Functions and DLLs in Excel VBA

This can be used to get values for numerical function in the DLL which have vectors as results:

Any function $F: \mathbb{R} \rightarrow \mathbb{R}^n$ can be recovered through its graph $(x, F(x))$ and calling for it just means to provide memory from VBA to be updated in the DLL. The former example (just an array) means that F is a constant.

As a simple example take $F(x)$ to consist of the powers of x

```
long __stdcall graph_vectorF(  
    double x,  
    double arr[],  
    long nLength )  
{  
    long i=0;  
  
    for(i=0;i<nLength;i++)  
    {  
        arr[i] = pow(x,i);  
    }  
  
    return i;  
}
```

and indeed in Excel we see the result:

```
Declare Function graph_vectorF_DLL _  
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _  
    Alias "graph_vectorF" ( _  
    ByVal x As Double, _  
    ByRef arr As Double, _  
    ByVal nLength As Long) As Long
```

```
Sub tst_graph_vectorF_DLL()  
' call a fct in the DLL which 'returns' an array  
' graph_vectorF(x,arr,n) computes an array of length n  
' where the entries are x^i, i = 0,...,n-1  
  
Dim nLength As Long, outLength As Long  
Dim arr() As Double ' do not use a fixed dimension  
Dim x As Double  
Dim k As Long  
Debug.Print "----"  
  
x = 2 ' to get power of 2  
  
nLength = 5 ' do not exceed 1024 ... as 2^1024 ~ 1e308  
arr = createArray(nLength)  
  
outLength = graph_vectorF_DLL(x, arr(1), nLength)  
k = printArray(arr)  
  
'Debug.Print "last element: " & arr(nLength)  
'Debug.Print "length in and out: " & nLength, outLength  
  
End Sub
```

which should give 1, 2, 4, 8, 16 in the debug window.

4. Using VBA Functions in a DLL: callback

Given some function $f: \mathbb{R} \rightarrow \mathbb{R}$ in VBA one wants to use it in a DLL. This is named a **callback** (since we start from Excel and call back to our departure).

A typical use: one has a routine for numerical integration in C and wants to use it within Excel for various function (so one does not want to hard code the integrand in the DLL). We will look at this later and take a simple example first.

For that the function as to be brought to the DLLs knowledge and using the operator **AddressOf** solves it, it is like &f in C. For proper work it has to be wrapped in a function type long for which the function dummyLong is used (it should be in a own module and I place it where all declarations are):

```
Function dummyLong(ByVal x As Long) As Long
dummyLong = x
End Function
```

Now take some function in VBA (it even can be a worksheet function like the cumulative normal distribution (which would be idiotic as it is a quite bad implementation in Excel)):

```
Function someFct( _
    ByVal x As Double) As Double
someFct = Exp(Sin(x)) ' Application.WorksheetFunction.NormSDist(x)
End Function
```

Define a function fct_callback in C which computes values using VBA and return that value:

```
typedef double (__stdcall * pF_arg1)(double);

double __stdcall fct_callback(
    pF_arg1 pF1,
    double x )
{
    return pF1(x);
}
```

and call it from VBA to check whether it works: one has to send the address of the VBA function and an argument x in which the DLL should evaluate it calling back to where it was send: it works!

```
Declare Function fct_callback_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "fct_callback" ( _
    ByVal adr As Long, _
    ByVal x As Double) As Double
```

```
Sub tst_callback()
' The example in the sub takes someFct and sends its adress pointer
' to the DLL, where it is evaluated in x. The result is sent to VBA
' so one can compare the results in VBA and from C.
Dim x As Double
Dim result As Double
Dim adr As Long
Debug.Print "----"

x = 0.5

adr = dummyLong(AddressOf someFct)
Debug.Print adr

result = fct_callback_DLL(adr, x)
Debug.Print "result by callback: " & result
Debug.Print "result in VBA:      " & someFct(x)

End Sub
```


5. An Advanced Example

Now combine what we have so far and how to proceed for several variables:

Take an array in n -space \mathbb{R}^n and a function $f: \mathbb{R} \rightarrow \mathbb{R}$ in VBA, send that to the DLL where it should be processed - just evaluate the array entries in the DLL by the function through callback and send the updated array to VBA.

As a slight variant additionally a parameter w is handed over, so I want the new elements to be $w * f(\text{arr}[i])$ where f even will depend on $\text{arr}[i]$ and i .

So we need a function of 2 variables in VBA (to be called back) and one external function to compute and to fetch the data where this is named `fctAdvEx`

```
Function someFct_2args( _
    ByVal x As Double, _
    ByVal j As Double) As Double
' to be called back from the DLL
' note that one has to change the definition in C if j would
' be declared of type long
someFct_2args = x ^ j
End Function
```

One has to care for the prototype of the function which has to be called back and for that use an appropriate type definition: be careful, except you like crashes.

```
typedef double (__stdcall * pF_arg2)(double, double);

long __stdcall fctAdvEx(
    pF_arg2 pF2,
    double w,
    double arr[],
    long nLength )
{
    long i=0;

    for(i=0;i<nLength;i++)
    {
        arr[i] = w*pF2(arr[i],i+1);    // i=0 should give 1 ...
    }

    return i;
}
```

The function returns the number of elements processed (so declare for long as return value)

```
Declare Function fctAdvEx_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "fctAdvEx" ( _
    ByVal adr As Long, _
    ByVal x As Double, _
    ByRef arr As Double, _
    ByVal nLength As Long) As Long
```

The address of the function of course does not know that the function has 2 arguments as input, so it is declared as long (and as in the last example it is accessed by its value).

Working with Array Functions and DLLs in Excel VBA

For a test take an array filled with ascending natural numbers. The result printed to the debug window thus is expected to consist of $w * i^i$, $1 \leq i \leq n$:

```
Sub tst_advancedExample()  
Dim adr As Long ' for callback  
Dim w As Double ' weighting  
Dim arr() As Double ' to be sent and updated  
Dim nLength As Long  
  
Dim outLength As Long, k As Long, i As Long  
Dim result As Double  
Debug.Print "---"  
  
w = 1  
  
nLength = 5  
arr = createArray(nLength)  
  
k = fillArray(arr) ' with ascending natural numbers  
'k = printArray(arr)  
  
adr = dummyLong(AddressOf someFct_2args)  
  
outLength = fctAdvEx_DLL(adr, w, arr(1), nLength)  
'Debug.Print outLength  
  
'k = printArray(arr)  
  
For i = 1 To outLength  
Debug.Print arr(i), w * someFct_2args(i, i) ' = w * i ^ i  
Next i  
  
End Sub
```

6. Global Arrays in VBA

Well, using globals is not a very structured coding style. But useful and if one wants to use and interface existing code without much modification it even might be a good choice.

I prefer namings from which I can see that it is a global variable (using a prefix or similar).

```
Global g_param() As Double
```

Note: no fixed dimension is used, that will allow easy assignments, but the variable is not initialised!

Without comment some examples how to work with it: just as for usual arrays, but with care.

Use a global vector of parameters on which the function operates; that vector has to be fill first (well, it is more a procedure than a function):

```
Function globalArrayFct( _
  ByVal nLength As Long) As Double
  ' implicitly uses:
  ' g_param (a global array) to be thought as parameters
  ' for the function, but given as an array
  ' function = fct_Example1 or any other may be coded here
  '
  ' and: do NOT modify any implicite things (the gobal) here,
  ' since this kind of coding is already bit dirty ...
  Dim arr() As Double
  Dim i As Long
  Dim s As Double

  If IsArray(g_param) Then ' that is need at least ...
  Else
    ' error processing ...
  End If

  ' Either use an existing one:
  'globalArrayFct = fct_Example1(g_param) ' summing the entries

  ' Or code it explicitly:
  arr = g_param ' I prefer a copy ...
  s = 0
  For i = 1 To UBound(arr)
    s = s + arr(i)
  Next

  globalArrayFct = s

End Function
```

The 'only' thing one has to care for is:

- initialise it and
- after populating it with data do not forget, that the data still stand in that array ...

Note that an easy assignment `g_param = array` works.

Working with Array Functions and DLLs in Excel VBA

```
Sub tst_globalArrayFct()  
Dim p() As Double ' parameters for the function  
Dim n As Long, kDummy As Long  
Dim result As Double  
Debug.Print "----"  
  
n = 5  
p = createArray(n)  
g_param = p  
  
kDummy = fillArray(g_param) ' some function to populate the vector  
  
result = globalArrayFct(n)  
Debug.Print "globalArrayFct(n)= " & result  
  
' do some house keeping if you feel the need  
g_param = createArray(1)  
g_param(1) = 0  
'printArray (g_param)  
  
End Sub
```

Data types are ok, just as for the usual case

```
Sub tst_createGDA_1()  
' create a global data area  
' show type and length  
Dim n As Long, kDummy As Long  
Dim arr() As Double  
Debug.Print "----"  
  
n = 3  
arr = createArray(n)  
g_param = arr  
Debug.Print "type of GDA: " & VarType(g_param)  
Debug.Print "is GDA an array? " & IsArray(g_param)  
Debug.Print "GDA length: " & UBound(g_param)  
End Sub
```

```
Sub tst_createGDA_2()  
' creates a global data area to be used by array functions  
' for that fill and print are used  
' for a longer test see below at the end of this module  
Dim n As Long, kDummy As Long  
Dim arr() As Double  
Debug.Print "----"  
  
n = 5  
arr = createArray(n)  
g_param = arr  
  
kDummy = fillArray(g_param)  
kDummy = printArray(g_param)  
End Sub
```

Some more tests for global arrays: use different sizes

```
Sub tst_createGDA_more()  
Dim n As Long, kDummy As Long  
Dim arr() As Double  
Debug.Print "----"  
  
n = 1  
arr = createArray(n)  
g_param = arr  
'Debug.Print "type of GDA: " & VarType(GDA)  
'Debug.Print "is GDA an array? " & IsArray(GDA)  
Debug.Print "GDA length: " & UBound(g_param)  
kDummy = fillArray(g_param)  
kDummy = printArray(g_param)  
  
n = 8  
arr = createArray(n)  
g_param = arr  
'Debug.Print "type of GDA: " & VarType(GDA)  
'Debug.Print "is GDA an array? " & IsArray(GDA)  
Debug.Print "GDA length: " & UBound(g_param)  
kDummy = fillArray(g_param)  
kDummy = printArray(g_param)  
  
n = 2  
arr = createArray(n)  
g_param = arr  
'Debug.Print "type of GDA: " & VarType(GDA)  
'Debug.Print "is GDA an array? " & IsArray(GDA)  
Debug.Print "GDA length: " & UBound(g_param)  
kDummy = fillArray(g_param)  
kDummy = printArray(g_param)  
  
n = 4  
arr = createArray(n)  
g_param = arr  
'Debug.Print "type of GDA: " & VarType(GDA)  
'Debug.Print "is GDA an array? " & IsArray(GDA)  
Debug.Print "GDA length: " & UBound(g_param)  
kDummy = fillArray(g_param)  
kDummy = printArray(g_param)  
  
n = -10  
arr = createArray(n)  
g_param = arr  
'Debug.Print "type of GDA: " & VarType(GDA)  
'Debug.Print "is GDA an array? " & IsArray(GDA)  
Debug.Print "GDA length: " & UBound(g_param)  
kDummy = fillArray(g_param)  
kDummy = printArray(g_param)  
End Sub
```

7. Callback for Functions with Vector Arguments

Given a function $f: \mathbb{R}^n \rightarrow \mathbb{R}$ in VBA one wants to use it in a DLL, a typical situation in estimating n parameters for curve fitting or multidimensional integration where the routine is written in C.

If there are only some few parameters then one can do that by using individual arguments $p1=p(1)$, $p2=p(2)$... and calling back as shown. A variant to avoid much typing and introducing lots of type definitions will be treated first in 7.1.

For more arguments it is convenient to have the parameters as array and is shown in 7.2

Thus we have

1. some DLL function g to be called as g_DLL from VBA
2. some given VBA function $f: X \times \mathbb{R}^n \rightarrow \mathbb{R}$ with parameters in \mathbb{R}^n
3. and some procedure $operateF$ in C which operates on f and needs to call f in VBA (as f should not be hard coded in C)

I will ignore the operation on f .

It is convenient to code f with a global array (which needs some care in handling, the global should be individual for f).

The idea:

The C function receives the pointer of a VBA function, which is able to sent an array of desired length to the DLL by reference. Then an update done in the DLL is available in VBA.

So if the array stands for the parameters of a vector function and are hold in a global array then this fixes the parameters and the function becomes an ordinary 1 dim function (of usual and individual arguments) to be called back in the way already seen.

So we need:

- a VBA function and its pointer for receiving an array (a kind of booting)
- a VBA function and it versions with fixed parameters to be called back by its pointer

And to see it from Excel we additionally need a function to trigger the function in the DLL.

Let us do the sending first and the rest in 8.1 and 8.2.

7.1 A Simple Way

A simple way would be working element wise (or processing even more elements within one call) and looping over all, this always works. This approach may be chosen, if only a small array has to be sent (and certainly is enough for integrals or fitting).

Suppose we have a function `holdingArray` in DLL which should send its array (the array in the DLL consist of $1/i$, $i = 1 \dots 10$) to VBA we need to trigger it from VBA to see that working in VBA and this is named `trigger_holdingArray`.

Additionally we have to send the address of `updateGDA` for a function to update data in VBA. This might have already been done (and stored in a global variable in the DLL) before by some initial set up - if so: no need to send it again. But to save typing code we do send it.

Of course that approach works for non-global arrays as well.

```
typedef long      (__stdcall * l_pF_dl)(double, long);

long __stdcall holdingArray(
    l_pF_dl ptr_updateGDAVBA,
    long n )
{
    double localArray[10] = {0.0};
    long localLength = 10;
    long i;
    long iCheck;

    for(i=1;i<=min(n,localLength);i++){
        localArray[i-1] = (double)1.0/i;
    }

    for(i=1;i<=min(n,localLength);i++)
    {
        iCheck = ptr_updateGDAVBA(localArray[i-1],i);
    }

    return i-1;
}
```

The function to be called for parameter updates in Excel by its address is

```
Function updateGDA_Small( _
    ByVal x As Double, _
    ByVal j As Long) As Long
' be sure that calling that index is correct,
' especially GDA has to be initialized before ...
GDA_Small(j) = x ' : Debug.Print x, j
End Function
```

which uses a global data array:

```
Global GDA_Small() As Double
```

To trigger the things we need a DLL

```
long __stdcall trigger_holdingArray(
    l_pF_dl ptr_updateGDAVBA,
    long n )
{
    return holdingArray( ptr_updateGDAVBA, n );
}
```

Working with Array Functions and DLLs in Excel VBA

with according declaration in VBA

```
Declare Function trigger_holdingArray_DLL _  
  Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _  
  Alias "trigger_holdingArray" ( _  
  ByVal adr As Long, _  
  ByVal nLength As Long) As Long
```

Then it works:

```
Sub tst_updateGDA_Small()  
Dim nLength As Long  
Dim arr() As Double  
Dim adr As Long  
Dim kDummy As Long  
Dim result  
Debug.Print "----"  
  
nLength = 3  
arr = createArray(nLength)  
  
GDA_Small = arr          ' initialize, holding zeros  
kDummy = printArray(GDA_Small) ' before update  
  
' trigger the DLL function (which holds an array) to send its data to  
' Excel (limited to the first 10 elements):  
adr = dummyLong(AddressOf updateGDA_Small)  
result = trigger_holdingArray_DLL(adr, nLength)  
  
' check the result within the GDA, expecting GDA(j) = 1/j:  
Debug.Print "updates: " & result  
kDummy = printArray(GDA_Small) ' after update  
  
End Sub
```

Note that this data transfer is not initiated by Excel, it comes from the DLL (and works, since we have Excel open and the function pointers are valid).

7.2 A General Solution

That follows the same principle, but avoids operating element wise. For that a global data array (set large enough for the problem under question) in the DLL is used over which the DLL functions can exchange data.

```
#define lengthGDA 8192 // 2^13 = 8192
double GDA_DLL[lengthGDA] = {0.0}; // each of 4 bytes = 32768 = 32 kB
```

The DLL function has the following form (where we need the pointer to the VBA which will give the parameter array):

```
typedef long (__stdcall * long_pF_long)(long);

long __stdcall holdingArrayGeneral(
    long_pF_long ptr_GDAFromDLLtoVBA,
    long n )
{
    double localArray[lengthGDA] = {0.0};
    long localLength = lengthGDA;
    long iCheck;
    long i;

    // populate a local array with some data
    for(i=1;i<=min(n,localLength);i++){
        localArray[i-1] = 1.0 +(double)1.0/i;}

    //memset( GDA_DLL, 0, lengthGDA );

    iCheck = copyLocalArrayToGDA( localArray, n);

    iCheck = ptr_GDAFromDLLtoVBA(n);

    return iCheck;
}
```

The trigger is need to kick it to work

```
long __stdcall trigger_holdingArrayGeneral(
    long_pF_long ptr_GDAFromDLLtoVBA,
    long n )
{
    return holdingArrayGeneral( ptr_GDAFromDLLtoVBA, n );
}
```

with declaration in VBA as

```
Declare Function trigger_holdingArrayGeneral_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "trigger_holdingArrayGeneral" ( _
    ByVal adr As Long, _
    ByVal nLength As Long) As Long
```

What we still need are C functions to copy an array within a C function to the global data area in the DLL and backwards:

Working with Array Functions and DLLs in Excel VBA

```
long __stdcall copyLocalArrayToGDA(  
    double* arr,  
    long n )  
{  
    long iCount;  
    long i, iLast, iStart;  
  
    iStart = 0;  
    iLast = min(n,lengthGDA) - 1;  
  
    iCount = min(n,lengthGDA) * sizeof(double);  
  
    //memcpy(GDA_DLL,arr, iCount);  
  
    for(i=iStart;i<=iLast;i++){  
        GDA_DLL[i] = arr[i];  
    }  
  
    return i;  
    //return iCount;  
}
```

```
long __stdcall copyGDAToLocalArray(  
    double* arr,  
    long n )  
{  
    long iCount;  
    long i, iLast, iStart;  
  
    iStart = 0;  
    iLast = min(n,lengthGDA) - 1;  
  
    iCount = min(n,lengthGDA) * sizeof(double);  
  
    //memcpy(arr, GDA_DLL, iCount);  
  
    for(i=iStart;i<=iLast;i++){  
        arr[i] = GDA_DLL[i];  
    }  
  
    return i;  
    //return iCount;  
}
```

One direction is done within the function which has the array, the other direction is done in Excel:

```
Declare Function copyGDAToLocalArray_DLL _  
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _  
    Alias "copyGDAToLocalArray" ( _  
    ByRef arr As Double, _  
    ByVal nLength As Long) As Long
```

which is used by

```
Function copyGDAFromDLLtoVBA( _  
    ByVal n As Long) As Long  
' does not modify GDA, be sure that GDAVBA has been initialized before  
' otherwise use the version below  
Dim kDummy As Long  
  
' that DLL fct copies n data from a global array to the provided array  
kDummy = copyGDAToLocalArray_DLL(GDA_VBA(1), n) 'Debug.Print GDA_VBA(1)  
  
copyGDAFromDLLtoVBA = kDummy  
End Function
```

or use

Working with Array Functions and DLLs in Excel VBA

```
Function fetchGDAFromDLLtoVBA( _  
    ByVal n As Long) As Long  
    ' this version initialize GDA_VBA before copying the data  
    ' (thus it modifies the GDA)  
    Dim p() As Double ' parameters  
    Dim kDummy As Long  
  
    p = createArray(n)  
  
    ' that DLL fct copies n data from a global array to the provided array  
    kDummy = copyGDAToLocalArray_DLL(p(1), n)  
  
    GDA_VBA = p  
    fetchGDAFromDLLtoVBA = kDummy  
End Function
```

This is more complicated than updating by running through the elements (and some graphics could make it clearer), but it works. Here is the test routine:

```
Sub tst_fetchingArray()  
    Dim nData As Long  
    Dim arr() As Double  
    Dim adr As Long  
    Dim kDummy As Long  
    Debug.Print "----"  
  
    nData = 3  
  
    adr = dummyLong(AddressOf copyGDAFromDLLtoVBA)  
    'adr = dummyLong(AddressOf fetchGDAFromDLLtoVBA)  
    'Debug.Print adr  
  
    arr = createArray(nData)  
    nData = CLng(UBound(arr))  
    GDA_VBA = arr ' initialize, holding zeros  
  
    'kDummy = fillArray(GDA_VBA) ' if one wants it different from zeros  
  
    kDummy = trigger_holdingArrayGeneral_DLL(adr, nData) ' tst_hasArray in DLL  
  
    Debug.Print "items processed: " & kDummy  
    kDummy = printArray(GDA_VBA)  
  
    ' last item:  
    Debug.Print "GDA_VBA(nData) = " & GDA_VBA(nData) ' maximal index = 8192  
    Debug.Print " should be = " & 1 + 1 / nData  
  
End Sub
```

8. Application Examples

Ok, that are not the real examples, they are intended to show how one can do it.

8.1 Parametric Integration

The operation on the C function may be thought as a routine to integrate some 1-dim function $f(x,p)$ with $x \in \mathbb{R}$ and parameters $p \in \mathbb{R}^n$ from a to b . This typically needs evaluations (and summations) of f in points between.

We take $\text{integral}(f(x), x = a \dots b) = (b-a) * (f(a)+f(b))/2$ just one trapezoid as integration routine, so we do not need an extra operateF (this is just the formula doing only 1 step of an actual routine) and do not need to handover the parameters to the DLL in this case.

So this is just using global parameters in VBA.

```
double __stdcall someIntegrationRoutine(
    pF_arg1 pF1,
    double a,
    double b )
{
    double result = 0;
    result = (b - a) * ( pF1(a) + pF1(b) ) * 0.5;
    return result;
}
```

This C function needs a declaration

```
Declare Function someIntegrationRoutine_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "someIntegrationRoutine" ( _
    ByVal adr As Long, _
    ByVal a As Double, _
    ByVal b As Double) As Double
```

and then we are in VBA only using global arrays for the parameters (the function is a polynomial with arbitrary coefficients)

```
Global g_paramF() As Double
```

```
Function given_F( _
    ByVal x As Double) As Double
    ' This function depends only of 1 explicit argument, but
    ' uses a global array g_paramF to be thought of length m:
    ' a polynomial of degree m, the coefficients are the parameters
    Dim s As Double
    Dim i As Long

    'Debug.Print "global number of parameters: " & UBound(g_paramF)
    'Debug.Print g_paramF(UBound(g_paramF))

    s = 0
    For i = 1 To UBound(g_paramF)
        s = s + g_paramF(i) * x ^ (i - 1)
    Next i

    given_F = s
End Function
```

Working with Array Functions and DLLs in Excel VBA

For testing of course the parameters have to be initialised (as this is not a 'symbolic' solution, it needs numerical values):

```
Sub tst_integrating_f()  
Dim m As Long  
Dim p() As Double  
Dim a As Double, b As Double  
Dim s As Double  
Dim i As Long  
Debug.Print "---"  
  
m = 4  
p = createArray(m)  
  
' set parameters for f  
p(1) = 1  
p(2) = 1 / 4  
p(3) = 1 / 9  
p(4) = 1 / 9  
  
' write to global for the function  
g_paramF = p  
  
' Only after that steps the function now actually is defined!  
  
' set integration bounds  
a = 0  
b = 1  
  
Debug.Print "through DLL: " & integrate_F(p, a, b)  
  
' check through the trapez formula  
Debug.Print "through VBA: " & (b - a) * (given_F(a) + given_F(b)) / 2  
  
' check it explicitly for the polynomial f using p  
s = 0  
For i = 1 To m  
    s = s + (b - a) * (p(i) * a ^ (i - 1) + p(i) * b ^ (i - 1)) / 2  
Next i  
Debug.Print "explicitly: " & s  
  
End Sub
```

The debug window will show results through the DLL and those within VBA only by using the function or using an explicit calculation. They are identical.

8.2 Least Square Fitting

A dummy version again, but it shows how to interface a running C program with Excel through a DLL without the need to adapt the code as it works only on the residues and through an initial call the necessary addresses and data can be written to global variables.

Task:

fit a model function $f(x,p)$ with parameters and $x \in \mathbb{R}^1$ against measured data to determine the parameters. This typically is done through a least square routine in a compiled library and for that all the residuals $f(x,p) - y$ have to be known in the DLL while it varies p and x,y run through the observations.

To keep things simple always take $y = 0$ (the parameter array in the DLL consist of $1/i$, $i = 1 \dots 10$)

So a callback is needed for a parametric function $f(x,p)$.

For x varying in data only one p is needed, so one p will have to be sent to Excel - thus we have the case of function which implicitly depends on a global parameter array in VBA, a case already shown:

Now just do it with a variable x .

```
typedef double (__stdcall * d_pF_dl)(double, long); // type of model
double __stdcall residuum(
    l_pF_dl ptr_updateGDAVBA,
    d_pF_dl ptr_model,
    double x,
    long mParams )
{
    double arrayParameters[10] = {0.0};
    long localLength = 10;
    double result;
    long i;
    long iCheck;

    // populate with some parameters
    for(i=1;i<=min(mParams,localLength);i++){
        arrayParameters[i-1] = (double)1.0/i;}

    // send it to VBA, updating the GDA there
    for(i=1;i<=min(mParams,localLength);i++)
    {
        iCheck = ptr_updateGDAVBA(arrayParameters[i-1],i);
    }

    // now the model function in VBA knows the above parameters,
    // hence evaluate the model function through callback to VBA
    result = ptr_model(x, mParams);

    return result;
}
```

Again a trigger is needed

```
double __stdcall trigger_residuum(
    l_pF_dl ptr_updateGDAVBA,
    d_pF_dl ptr_model,
    double x,
    long mParams )
{
    return residuum(ptr_updateGDAVBA,ptr_model,x,mParams);
}
```

to be declared in Excel

```
Declare Function trigger_residuum_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "trigger_residuum" ( _
    ByVal adr_updateGDA As Long, _
    ByVal adr_model As Long, _
    ByVal x As Double, _
    ByVal nLength As Long) As Double
```

In Excel the model function is set up using a parameter array as input

```
Function modelFct( _
    ByVal x As Double, _
    ByRef p() As Double, _
    ByVal m As Long) As Double
    ' uses parameter array p to be thought of length m
    Dim s As Double
    Dim i As Long

    s = 0
    For i = 1 To m
        s = s + p(i) * x ^ (i - 1)
    Next i
    modelFct = s
End Function
```

This again is a polynomial function (i.e. fitting a polynomial against data is intended).

It is made 1 dimensional ($x \in \mathbb{R}^1$) by fixing the parameters through a global array:

```
Global g_paramModelFct() As Double
```

```
Function g_modelFct( _
    ByVal x As Double, _
    ByVal m As Long) As Double
    ' uses g_paramModelFct instead of p to be thought of length m
    ' so it becomes 1 dimensional (by fixing at some p)
    Dim s As Double
    Dim i As Long

    's = modelFct(x, g_paramModelFct, m) ' or explicitly the following:
    s = 0
    For i = 1 To m
        s = s + g_paramModelFct(i) * x ^ (i - 1)
    Next i

    g_modelFct = s
End Function
```

Write down the VBA function to transport the parameters, one of the two following:

```
Function updateParamModelFct( _
    ByVal x As Double, _
    ByVal j As Long) As Long
    ' be sure that calling that index is correct
    ' and g_paramModelFct has been initialized before
    g_paramModelFct(j) = x
End Function
```

```
Function copyGDAFromDLLtoParamModelFct( _
    ByVal n As Long) As Long
' be sure that g_paramModelFct has been initialized before
Dim kDummy As Long

' that DLL fct copies n data from a global array to the provided array
kDummy = copyGDAToLocalArray_DLL(g_paramModelFct(1), n)

copyGDAFromDLLtoParamModelFct = kDummy
End Function
```

I prefer the first and simple version: it avoids a global array in the DLL and usually there are not that much parameters (remember that callbacks need time), the example in the DLL is coded for that case.

Now let it run:

```
Sub tst_fitting()
Dim adr_updateGDA As Long
Dim adr_model As Long
Dim x As Double
Dim m As Long, mDummy As Long
Dim result As Double
Debug.Print "----"

adr_updateGDA = dummyLong(AddressOf updateParamModelFct)
adr_model = dummyLong(AddressOf g_modelFct)

m = 10
g_paramModelFct = createArray(m) ' erst init !

x = -1.23456
result = trigger_residuum_DLL(adr_updateGDA, adr_model, x, m)

Debug.Print "parameters are: "
    mDummy = printArray(g_paramModelFct)
Debug.Print "residuum seen in DLL: " & result
Debug.Print "residuum seen in VBA: " & modelFct(x, g_paramModelFct, m)
End Sub
```


Appendix: The True and Lazy Way

After all that: there is an easy way using an API call (sorry if you have read all linear from beginning).

We have seen, that arrays provided from Excel by reference can be read and updated in the DLL. The converse can be done as well. And here is my limited understanding for that:

A given function `f` in C it can be used through `r = f_DLL(p(1))` after it has been declared by some `Declare function f_DLL(ByRef p As Double) as Double`.

This is the same as if `f` of type `double f(double *)` is called in C as `f(arr)`, `arr` the array in C:

A function in C with an array as argument is to be understood by its first element `arr[0]` (its address is used) and the type `double*` of its argument by working with offsets for the base `arr[0]`.

Now if `f = ptr_fct` is given as the address of a VBA function `H(ByRef p as Double) as Double`, then `f(arr)` in C provides `H` with the base and knowing that one can use the offset of the memory of `arr` - the memory is kept and one can access it to read and write.

This can not be seen from the VBA definition for `H`, see below.

Note that the definition for `H` has to use the calling convention `ByRef` and this is done without the brackets (!) to have the argument available as array in its body.

Working with the transported array data in VBA usually needs to copy them to VBA arrays.

But VBA complains in about a code like `p = arr` or `p(i) = arr(i)` as it understand the base as double and not as an array. For that one has use an API function, see the code below.

Ok, that does make calls to the Windows system, but that parts are 'only' extremely fast implementations of functions of ANSI C and thus they can always be made available otherwise.

To find out limitations there are two versions, one with an array in C of fixed length, the other allocates memory to use a pointer.

The declaration for handling the memory:

```
Declare Sub RtlMoveMemory Lib "kernel32" ( _  
    hpvDest As Any, _  
    hpvSource As Any, _  
    ByVal cbCopy As Long)
```

It allows to copy data from `Source` to `Dest` while `cbCopy` determines the length, it is in Bytes (so for arrays of doubles one needs `8 * length`). The function should be described in descent API guides.

As already said it is used in VBA, the function which needs it can be altered to provide the necessary local or global arrays and other desired VBA functionalities (as seen in the stuff before) and is given as follows:

Working with Array Functions and DLLs in Excel VBA

```
Function updateArray(ByRef arr As Double, ByVal m As Long) As Long
Dim i As Long, iDummy As Long
Dim p() As Double
Dim dummy

p = createArray(m) ' consists of 0.0 entries

'Debug.Print "array base: " & arr
'Debug.Print "byte length array base: " & Len(arr)
'Debug.Print "byte length local array base: " & Len(p(1))

' p = arr or p(i) = arr(i) does _not_ work
Call RtlMoveMemory( _
    p(1), _
    arr, _
    m * Len(p(1)))

' after that p contains a copy of arr (care for correct space!)
' to check whether p was filled one can print it or similar

' iDummy = printArray(p)

' now overwrite p by natural numbers
For i = 1 To m
    p(i) = i
Next i

Call RtlMoveMemory( _
    arr, _
    p(1), _
    m * Len(arr))

' then arr holds a copy of p (care for correct space!)
' and if it was sent from the DLL the data are available there

updateArray = m
End Function
```

Next we need the C function which will call that VBA function and a trigger to kick and watch the things from Excel (the 'static' version is in the source as well).

```
#include <malloc.h>

typedef long          (__stdcall * l_pF_al)(double*, long); // appendix
```

```
double __stdcall
trigger_directDynamic(
    l_pF_al ptr_updateVBA,
    long mParams )
{
    double result;
    result = directDynamic(ptr_updateVBA, mParams);
    return result;
}
```

where the trigger needs a declaration in VBA

```
Declare Function trigger_directDynamic_DLL _
    Lib "C:\_Work\MyProjects\array_excel\Release\arr_XL.dll" _
    Alias "trigger_directDynamic" ( _
    ByVal adr As Long, _
    ByVal nLength As Long) As Double
```

Working with Array Functions and DLLs in Excel VBA

```
double __stdcall directDynamic(
    l_pF_al ptr_updateVBA,
    long mParams )
{
    double result;
    long i;
    long iCheck;

    double* arr;
    arr = (double *)calloc(mParams, sizeof(double));

    // populate with some parameters
    for(i=1; i<=mParams; i++){
        arr[i-1] = 1.0 + (double)(1.0/i);}

    // send it to VBA, updating an array there and getting updated itself
    iCheck = ptr_updateVBA(arr, mParams);

    // evaluate to see whether array was modified in VBA
    result = sumUp(1.0, arr, mParams);

    free(arr);

    return result;
}
```

Now its time for a test:

```
Sub tst_updateArray_DynamicVersion()
' This trigger kicks a C function "direct" which calls updateArray
' (by the address) providing a static array filled with 1 + 1/i.
' So after the first use of the API call the local array there will
' contain that values (activate printing the VBA function to check).
' After updating p and the second API call the provided array will
' hold natural numbers. The DLL function sums up and returns the
' value which thus should give m * (m + 1) / 2
Dim m As Long
Dim adr As Long
Debug.Print "----"

m = 100000
adr = dummyLong(AddressOf updateArray)
Debug.Print "result from DLL: " & trigger_directDynamic_DLL(adr, m)
Debug.Print "and as VBA sum : " & CDbl(m) * (CDbl(m) + 1) / 2
End Sub
```

The debug window immediately should show 5000050000 for both outputs.

For larger m like 10^6 do not forget that at least a loop in VBA needs its time, but it works.