Access VBA Made Easy

Arrays and Collections

08

www.accessallinone.com

This guide was prepared for AccessAllInOne.com by:
Robert Austin

This is one of a series of guides pertaining to the use of Microsoft Access.

Contents

Arrays and Collections

Computing is all about sets of similar looking data; appointments, files, pictures, addresses, UDP packets, tracks, database records, patient records, library records, lots of records. These different data structures inside our programs, computers, hard-drives and memory will be stored as repeating rows making up arrays and collections.

This unit will first introduce Arrays as the traditional data structure and also in VBA's somewhat extended variant. This will lay the foundation for understanding Collections and appreciating the differences between the two structures and be able to choose which best suits your particular task.

Traditionally, an Array has always been a block of memory put aside to hold values of a particular type. Its size is set at the time it is initiated and any element within it may be accessed randomly or sequentially. The best way to envisage an Array is like a table of data that is held in memory.

A Collection is an object that holds references to other objects of a similar type. It is somewhat similar to an array, in that it holds a list of things, but a collection is normally dynamic in size and, over all, easier to use than an Array. Objects in a collection can also be randomly or sequentially accessed.

Declaring Arrays

You can think of an array as a row of boxes with a number on each, 0 to *n*. When we first declare an array we must at least state its type and may also state its size (we can set the size later if we wish).

Firstly, we will create an array that will hold Integer types (whole numbers).

```
1   Dim myIntegerArray() as Integer
```

myIntegerArray : Array of Integers

Figure 8.1

The opening and closing parenthesis after the variable name are the indicator that myIntegerArray is an array. At this point, VBA is aware that myIntegerArray will be an array containing Integers but it doesn't know how large we want it.

In this example we will set the size of the array when we declare it. We will make a 10 integer array. *Each Integer takes up 4 bytes.*

```
1   Dim myIntegerArray(10) as Integer
```

myIntegerArray: Array of Integers (0..9)

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Figure 8.2

If we don't want to set the size of the array when we declare it we can omit the number indicating the total items it can hold and use a redim statement to set the size later on in the code.

```
1   Dim myStringArray() as String
2   ReDim myStringArray(10)
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
""	""	""	""	""	""	""	""	""	""

Figure 8.3

VBA initialises Strings to "", an empty String. Each character of a string takes up at least 2 bytes.

Let's take a look at how Access initialises other data types.

```
1   Dim myFloatArray() as Float
2   ReDim myFloatArray(10)
```

myFloatArray: Array of Floats (0..9)									
0	1	2	3	4	5	6	7	8	9
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 8.4

VBA initialises Floats to 0.0. A float takes up 8 bytes.

```
1    Dim myDateArray() as Date
2    ReDim myDateArray (10)
```

myDateArray: Array of Dates (0..9)									
0	1	2	3	4	5	6	7	8	9
00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

Figure 8.5

VBA initialises Dates to 00:00:00. A date takes up 8 bytes.

```
1   Dim myBooleanArray() as Boolean
2   ReDim myBooleanArray (10)
```

myBooleanArray: Array of Boolean (0..9)									
0	1	2	3	4	5	6	7	8	9
False	False	False	False	False	False	False	False	False	False

Figure 8.6

VBA initialises Boolean values to False. A Boolean takes up 1 byte.

Referencing Arrays

Continuing with our row of boxes analogy, an array is referenced by its name and the box we wish to work with. For example, to get the value of box 0 we use myIntegerArray(0); to reference box 9 we use myIntegerArray(9).

```
3   myIntegerArray(0) = 10
4   myIntegerArray(1) = 36
5   myIntegerArray(2) = 77
6   myIntegerArray(4) = 87
7   myIntegerArray(5) = -10
```

myIntegerArray: Array of Integers (0..9)									
0	1	2	3	4	5	6	7	8	9
10	36	77	87	-10	0	0	0	0	0

Figure 8.7

Here are some useful Strings, 10 Top-Level Domains:

```
3   myStringArray (0) = "UK" : myStringArray (5) = "ME"
4   myStringArray (1) = "RU" : myStringArray (6) = "COM"
5   myStringArray (2) = "HR" : myStringArray (7) = "INFO"
6   myStringArray (3) = "DE" : myStringArray (8) = "NET"
7   myStringArray (4) = "FR" : myStringArray (9) = "EU"
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
UK	RU	HR	DE	FR	ME	COM	INFO	NET	EU

Figure 8.8

Fixed Length and Dynamic Arrays

One of the headaches with arrays is that they are static blocks of memory and are not designed to change in size. If we want to add another 5 domain names to myStringArray we have to re-declare the array. Oh, and by the way, doing so usually gives you back a new clean array!

ReDim and Preserve

VBA offers the ReDim function which performs much of the leg-work involved in changing an array's size. ReDim also has a useful keyword Preserve which preserves the data in your array as you change its size.

```
1   Dim myIntegerArray() as Integer ' define array variable
2   ReDim myIntegerArray(10)        ' set array size and memory allocation
3   myIntegerArray(0) = 22          ' set (0) to 22
4   ReDim Preserve myIntegerArray(20)' extend array preserving (0)=22
```

Figure 8.9

The standard ReDim function would destroy the old array and make a new one; with the Preserve keyword included, VBA creates the new array of the new size and copies over the previous arrays values, making them available to us.

A fixed-length array is what the above arrays are called – they cannot be changed. A dynamic array is more flexible allowing the array to grow and shrink in size over time without having to recreate the array data and structure.

Variant Arrays

Another type of array that VBA implements, is the Variant Array. Variant arrays handle all primitive types and each element of the array can be loaded with any data type. This contrasts with "standard" arrays which can hold only one primitive data type.

Variant arrays handle just like regular arrays, requiring us to ReDim to change the number of variables it can store.

```
1   ' A variant array can hold any primitive data type, but it is
2   ' actually stored as an object
3
4   Dim myVariableArray As Variant
5   myVariableArray    = Array(10)
6   myVariableArray(0) = "First element"
7   myVariableArray(1) = 2
8   myVariableArray(2) = new Date(#12-09-1989#)
```

Figure 8.11

Erasing an Array

Erasing an array is so important that VBA – a language that usually makes things easy for programmers – provides a dedicated function to release memory held by an array. If you don't remove an array VBA, will *garbage collect* memory space left when variables go out of scope, but you are advised to explicitly erase array structures when finished with them. Once erased the variable must be ReDim'd.

```
1   Dim myVariableArray() As Variant
2   ReDim myVariableArray(10)
3   myVariableArray(0) = 1
4   myVariableArray(1) = 2
5   myVariableArray(2) = 3
6
7   Erase myVariableArray ' myVariableArray has no more data and must be
8   ReDim'd to be used
```

Figure 8.12

Split Function

The split function splits a string into an array of strings based on some delimiter. The following example demonstrates splitting a string based on spaces.

```
1   Sub SplitFunction()
2   Dim i As Integer
3   Dim myArray() As String
4          myArray = Split("here;we;go;again!", ";")
5          For i = LBound(myArray) To UBound(myArray)
6              Debug.Print myArray(i)
7        Next i
8   End Sub
```

Figure 8.13

Join Function

Join does the exact opposite of split; it requires an array and a delimiter and returns a single string.

```
1   Sub JoinFunction()
2       Dim myArray() As Variant
3       myArray() = Array("here", "we", "go", "again", "!")
4       Debug.Print Join(myArray(), " ")
5   End Sub
```

Figure 8.14

Multi-Dimensional Arrays

All the arrays shown above are one-dimensional arrays. It is also possible to create an array with more than one dimension. For example, you may have an array of week numbers with days to hold an Integer number.

```
1   Dim myIntegerArray() as Integer
2   ReDim myIntegerArray(52,7)
```

Figure 8.15

In the above code, we tell Access to create an array of 364 elements (7×52).

In the code below, we create a 3×5 array, fill certain elements with values and print it to the immediate window.

```
1   Sub MultiDimendionalArrays()
2   Dim myIntegerArray() As Integer
3   Dim element As Variant
4   ReDim myIntegerArray(3, 5)
5   'we create a 3 x 5 array
6
7   myIntegerArray(0, 0) = 1
8   myIntegerArray(1, 2) = 2
9   myIntegerArray(3, 3) = 3
10  myIntegerArray(2, 3) = 4
11  myIntegerArray(3, 3) = 5
12  myIntegerArray(3, 1) = 6
13  'We fill some of the array elements with values
14  'Any we don't fill will get a default value of
15  '0 because the array is of type integer
16
17  For Each element In myIntegerArray
18      Debug.Print element
19      'here we print the array to the immediate window
20  Next
21
22
23  End Sub
```

Figure 8.16

The output in the immediate window will be:

```
1
0
0
0
0
0
0
6
0
2
0
```

o
o
o
4
5
o
o
o
o
o
o
o
o

The example above illustrates clearly the idea of two-dimensional arrays; think of them like tables that you can't see…

Collections

A Collection is an object that stores other objects. Usually a collection will store objects of a particular type so servicing those objects with functionality specifically required by them. When using collections we don't need to worry about ReDim'ing them; the collection will increase in size all by itself so we need only declare a variable to point to a Collection Object and instantiate a new Collection Object.

```
1   Function makeCar() As Collection
2     Dim parts As New Collection
3     Dim part As Variant
4
5     parts.Add "Volvo"
6     parts.Add 5
7     parts.Add "V70R"
8     parts.Add "Sunroof"
9     parts.Add "Drive"
10    parts.Add #12/24/2012#
11
12    Dim t As Integer
13    For Each part In parts
14      t = t + 1
15      Debug.Print t, part, TypeName(part)
16    Next
17
18  End Function
```

Figure 8.17

The above code creates a new Collection object, adds some primitive types to it then cycles through the Collection outputting it as position, value and data type. Below is the output.

```
 1             Volvo          String
 2              5             Integer
 3             V70R           String
 4             Sunroof        String
 5             Drive          String
 6             24/12/2012     Date
```

Figure 8.18

Relationship with Objects

Collections are used everywhere in VBA and Access. For example, AllForms, AllQueries, AllReports, AllMacros, AllModules, AllViews, Form.Controls, Page.Properties, Form.Properties … Report.Controls … basically a collection is used to hold everything about your application; even collections inside collections. In VBA there are dozens of different collections and although they all inherit from a generic Collections Class one must work the particular Collection for a particular object.

Properties Associated with Objects

Every class of object in VBA and Access have a Properties Collection that for the most part is built-in to the class. It is possible to create user-defined properties and add them to a class instance.

Practical Uses of Collections : Form and Report Controls

```
1   Function listCtrlsAndProperties()
2     Dim ctrls As Access.Controls
3     Dim ctrl As Access.Control
4     Dim prop As Property, tmp As String
5
6     ' the Forms collection is only populated with instantiated forms
7     DoCmd.OpenForm "frmStudentsDataEntry", acDesign, , , , acHidden
8
9     Set ctrls = Application.Forms("frmStudentsDataEntry").Controls
10
11    For Each ctrl In ctrls
12      Debug.Print ctrl.Name
13      For Each prop In ctrl.Properties
14        tmp = tmp & "," & prop.Name
15      Next
16      Debug.Print tmp
17      tmp = ""
18    Next
19    DoCmd.Close acForm, "frmCoursesNav"
20  End Function
```

Figure 8.19

Note: *The Set operator is required because we are dealing with Objects and not primitive types – this is a VBA specific requirement.*

The above function opens a form in design view, cycles through the controls collection of the form, then for each control cycles through its properties collection. This demonstrates that a Collection can hold a Collection – the Controls collection has individual Controls which in turn have a Properties Collection and individual properties, like Height, BackColor and ForeColor.

You can also change the appearance of objects on the screen by changing related properties. Use the following instruction to change the height of a button on a form

- Open a new form and save it with the name “frmButtonChangeTest”.
- Add to the form a button and call it “btnChangeAppearance”.
- Navigate to the Events tab and double-click the On Click event and open the VBA Editor.
- Insert the following code

```
1   Private Sub btnChangeAppearance_Click()
2
3     Dim lHeight As Double
4     Dim btn As CommandButton
5
6     Set btn = Forms!frmButtonChangeTest.controls!btnChangeAppearance
7     lHeight = btn.Height    : btn.Height = lHeight + 50
8
9     lHeight = Me.controls("btnChangeAppearance").Properties("Height")
10    Me.controls("btnChangeAppearance").Properties("Height")=lHeight+50
11
12  End Sub
```

Figure 8.20

Collections: Control. ControlType

Using the collection Controls of a form all controls can be cycled and only those of a particular type can be targeted. In the following example all controls of the form are checked for 1) their type and 2) the section in which they appear. If a control is in the Detail of the form their values are output to the Immediate Window.

```
1   Private Sub Form_BeforeUpdate(Cancel As Integer)
2
3     Dim c As Variant
4     For Each c In Me.Form.controls
5
6       If c.ControlType = acTextBox And c.Section = acDetail Then
7         Debug.Print c.name & " = '" & c & "'"
8       End If
9
10    Next
11
12  End Sub
```

Figure 8.21

Checking if a Form is loaded

To check whether a form is currently open or not use the CurrentProject.AllForms collection which has an IsLoaded function which returns true if the form is loaded. CurrentProject also contains all the other All* collections.

```
1   Function isMyFormOpen(frmName As String) As Boolean
2     isMyFormOpen = CurrentProject.AllForms(frmName).IsLoaded
3   End Function
```

```
? isMyFormOpen("frmStudentsDataEntry")
False
```

Figure 8.22

Referencing Controls

Since we have discussed form and report controls, we thought you might like to know how to reference forms and their controls.

Me keyword

The Me keyword is associated with classes and object modules – using it in the standard module will result in a compilation error. In a Form module, Me refers to the form itself. Writing "me" tells VBA to reference the current form or report.

```
1   Option Compare Database
2
3   Private Sub Command11_Click()
4     MsgBox Me.Form.name ' msgbox opens with the form's name
5   End Sub
6
7   Private Sub Command12_Click()
8     MsgBox Me.Form!field1 ' msgbox opens display content of field1
9   End Sub
```

```
? isMyFormOpen("frmClassesNav")
False
```

Figure 8.23

Full Form Reference

Referencing the form itself can be performed by writing:

```
1   Option Compare Database
2
3   Private Sub Command0_Click()
4     MsgBox Forms(Form.name).name ' msgbox opens with the form's name
5   End Sub
```

Figure 8.24

You can also reference another form if it is open. All open forms are held in the Forms collection. Accessing other forms is very helpful when passing data between forms or setting up a form that edits a child record of the first form.

```
1   Private Sub Command2_Click()
2     If AllForms("otherform").IsLoaded Then
3       Forms("otherform").controls("customerID") = Me![CustomerID]
4       Forms("otherform").FilterOn = True
5     End If
6   End Sub
```

Figure 8.25

Sub Form Reference

A form may be embedded into a parent form so showing records of some child table. The subform can be accessed by accessing the embedded form's name. The subform is added to the parent form's Controls collection so is referenced like *any other control on the form*.

```
1   Private Sub Command2_Click()
2       Me.frmCarDataSub.Form.Detail.BackColor = vbRed
3   End Sub
```

Figure 8.26

This last item demonstrates what this whole unit is about. Arrays and Collections are the containers of all our data and highly versatile. They are only lists of primitive data or lists of objects but they take up the most space and the most resources. Creating an array can sink a system or make it run lightning fast, as long as it is well maintained.

Common Errors

Not Releasing Memory

Whenever you instantiate an object you should always release the memory. Explicitly releasing memory by erasing arrays or removing an object from a collection forces VBA, .Net or Java to process that memory hole. Leaving objects floating and relying on garbage collectors can slow down you application, and worse, cause memory leaks.

Out of Memory

Not releasing arrays and collections, or requesting too much space can result in an Out of Memory error. This was quite frequent 10 years ago, and even now with virtual memory on TB hard drives, running out of memory is possible

Slooooooow Response Times

Again, creating arrays and collections you will not use. When you request a block of memory your computer will allocate it. When that memory isn't in use or doesn't fit into physical memory, it will be swapped out to a hard drive or SD Card, and getting that data back into memory can result in serious slowdown.

Exception: Out of Bounds

Make sure not to attempt to access elements of an array that don't exist by knowing the upper and lower bounds of your arrays. Collections in VBA start at 1. Arrays usually start at 0 but may start at 1. The upper bounds of arrays shouldn't be passed either; this can cause Out of Bounds exceptions, or in a really bad situation may try to execute data as if it were instructions – that is how viruses get their code executed.

Questions

1) Describe the structure of an array?

2) What is the difference between a dynamic array and a fixed length array?

3) Which of the following defines an array or pointer to an array correctly in VBA?
   a. Integer[] myIntegerArray;
   b. Dim myStringArray = new String(10)
   c. Dim myStringArray;
   d. ReDim myIntegerArray(10)
   e. Dim myDateArray() as Date

4) Why can arrays and collection cause many problems?
   a. They fire too many rounds
   b. They can take up a lot of memory
   c. CPU time can be huge
   d. Virtual memory can be used up
   e. Collections are never a problem

5) The following code wipes out the old data. Correct it to maintain old data.

```
1   Dim integerArray(3) as Integer
2   integerArray(0) = 20
3   integerArray(1) = 99
4   integerArray(2) = 887
5
6   ReDim integerArray(10)
7   integerArray(3) = 44
```

6) aString = “My son went to market and brought dried bananas”
   What letter appears in the following?
   a. aString(9)
   b. aString(31)
   c. a = 20 : aString(a)
   d. c = 4 * 8 : aString(c)
   e. instr(1,aString,"y")
   f. aString(instr(1,aString,"i"))

7) Which of the following are not VBA or Access collections?
   a. AllForms
   b. AllModules
   c. AllStrings
   d. Report.Controls
   e. Properties
   f. Fields
   g. Recordset.Fields

h. Me.Controls

8) Fill out the following table.

```
1   myIntegerArray = array(8,9,10,5,3,23,65,99,121,00)
```

myIntegerArray: Array of Integers (0..9)									
0	1	2	3	4	5	6	7	8	9

9) Fill out the following table.

```
1   myString = "There,follows,a,party,political,broad,cast,!,?"
    myStringArray = Split(myString, ",")
```

myStringArray: Array of Strings									
0	1	2	3	4	5	6	7	8	9

10) From (9) complete the following Immediate window statement to print all array elements.

```
For Each ___ In _____________ : ? a : next
```

11) Fill in the missing numbers.

```
3   myStringArray (___) = "INFO" : myStringArray (___) = "DE"
4   myStringArray (___) = "RU"   : myStringArray (___) = "HR"
5   myStringArray (___) = "COM"  : myStringArray (___) = "FR"
6   myStringArray (___) = "DE"   : myStringArray (___) = "NET"
7   myStringArray (___) = "ME"   : myStringArray (___) = "UK"
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
UK	RU	HR	DE	FR	ME	COM	INFO	NET	EU

12) Why does the following code not work? Correct it. What is the output?

```
1   Function collectionsTest1()
2     Dim col As New Collection
3     Dim num As Integer
4
5     num = 10: col.Add num
6     num = 30: col.Add num
7     num = 88: col.Add num
8     num = 30: col.Remove num
9     num = col.Item(1): Debug.Print num
10    collectionsTest1 = num
11  End Function
```

13) If the following needs changing, change it so that line 11 returns littleArray(1) = 66.

```
1   Function arrayTest2()
2     Dim littleArray(4) As Integer
3     littleArray(0) = 1
4     littleArray(1) = 99
5     littleArray(2) = 5
6     littleArray(3) = 67
7     Erase littleArray
8     littleArray(0) = 1
9     littleArray(1) = 66
10    littleArray(2) = 5
11    arrayTest2 = littleArray(1)
12  End Function
```

14) Write a multi-dimensional array that is called and represents a chessboard that could hold the text queen, king, bishop, knight, rook, pawn.

15) True or false

a. Collections are a string of characters
b. Less memory is used by an object in a collection than an integer in an array
c. Arrays are slower to access than a collection
d. A variant array may hold objects
e. Arrays are instantiated
f. To increase the size of a collection we used ReDim
g. Preserving an array maintains its size and clears the content
h. c = 10 / 2: Dim A() As Integer: ReDim A(5): Debug.Print UBound(A) = c
i. Arrays are instantiated

16) What is special about the Forms collection?

17) SubForm KOL can be found where in relation to Me?

18) Are Strings, by default, dynamic or fixed length arrays of characters?

19) How does VBA implement dynamic arrays for primitive types?

20) If an Double takes up 8 bytes of memory space, and a Float takes by 8 bytes of memory space, and one character of a String takes up 2 bytes of memory space, rank the following in order of size, smallest to largest:

Float(5)	
Double(6)	
String “Foobar”	

Float 6.77	
String(1)	
Double(2,5)	
Double(3,3,3)	
Float(6,1)	
String(10,2)	

Answers

1) Traditionally, an Array has been a block of memory put aside to hold values of a particular type. Its size is set at the time it is initiated and any element within it may be accessed randomly or sequentially.
2) Dynamic can change over time whilst a fixed cannot
3) Yes or no
    a. No
    b. No
    c. Yes
    d. Yes
    e. Yes
4) Yes or no
    a. No
    b. Yes
    c. Yes
    d. Yes
    e. No
5) Line 6: `ReDim Preserve integerArray(10)`
6) Letters below
    a. e
    b. g
    c. e
    d. h
    e. 2
    f. i
7) yes or no
    a. yes
    b. yes
    c. no
    d. yes
    e. yes
    f. yes
    g. yes
    h. yes
8) see below

```
1   myIntegerArray = array(8,9,10,5,3,23,65,99,121,00)
```

myIntegerArray: Array of Integers (0..9)									
0	1	2	3	4	5	6	7	8	9
8	9	10	5	3	23	65	99	121	0

9) see below

```
1   myString = "There,follows,a,party,political,broad,cast,!,?"
    myStringArray = Split(myString, ",")
```

myStringArray: Array of Strings									
0	1	2	3	4	5	6	7	8	9
There	follows	a	party	political	broad	cast	!	?	

10)
```
For Each _a_ In __myStringArray__ : ? a : next
```

11) See below

```
3   myStringArray (_7_) = "INFO" : myStringArray (_3_) = "DE"
4   myStringArray (_1_) = "RU"   : myStringArray (_2_) = "HR"
5   myStringArray (_6_) = "COM"  : myStringArray (_4_) = "FR"
6   myStringArray (_3_) = "DE"   : myStringArray (_8_) = "NET"
7   myStringArray (_5_) = "ME"   : myStringArray (_0_) = "UK"
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
UK	RU	HR	DE	FR	ME	COM	INFO	NET	EU

12) Line 8 causes an out of bounds error
change to num=2
collectionsTest1 = 10

13) Cheeky answer, comment out line 7
otherwise place a ReDim littleArray(3) after line 7

14) Dim Chessboard(8,8) As String

15) True or false

a. False
b. False
c. False
d. True
e. False
f. False
g. False
h. True (sorry)
i. False

16) Forms only contains those forms that are open

17) Me.KOL or Me.Controls("KOL")

18) Dynamic

19) By using ReDim

20) See below

Float(5)	40	4
Double(6)	48	6
String “Foobar”	12	3
Float 6.77	8	2
String(1)	2	1
Double(2,5)	80	8
Double(3,3,3)	216	9
Float(6,1)	48	7
String(10,2)	40	5