Visual Basic - Arrays

An *array* is a named collection of variables of the same data type. Each element of the array is labelled by one or more indices, e.g.

```
candNames(1) = "Smith"
candNames(2) = "Patel"
candNames(3) = "Oddbins"
```

Arrays allow you to group related variables and to manipulate them all at once while still being able to access each variable separately. For arrays we can set up loops using the index numbers.

The size of an array is determined by the number of dimensions (i.e. the number of arguments) and the lower and upper bounds of the index number of each dimension. VB arrays can have 60 dimensions, with computer memory size limiting the number of elements. In the example above `candNames` has three elements and one dimension.
VB allocates space for every element of an array even if you never store a value in it. Thus you should avoid declaring an array larger than you really need. If you know what size you need declare a *static* array and specify the lower and upper bounds of each dimension. On the other hand if the size is truly not known you can declare a *dynamic* array.

You must explicitly declare an array before you use it; implicit declaration is not allowed. You use **Private**, **Public**, **Dim** or **Static** keywords to define its scope and lifetime; use integer values to specify the lower and upper bounds on each dimension; use **As** to specify the data type of the elements (all the same).
You specify lower and upper bounds within brackets after the array name. The bounds can be positive, negative integers or zero. If you give only one value for a dimension it is interpreted as the upper bound. The default lower bound is zero (0) unless you set it to 1 – which is recommended. The statement

```
Dim counters(14) As Integer
```

defines counters as an array with 15 elements going from `counters(0)` through to `counters(14)`. To set the default lower bound to 1 use `Option Base 1` in a module. This is strongly recommended as all Excel collections are 1-base. If you assign a 1-based array to a 0-based, the same data exits in both arrays but each element is offset by one. This makes macros confusing and leads to errors. To specify explicitly both bounds use `lowewrBound To upperBound` as in

```
Dim counters(1 To 15) As Integer
Dim totals(100 To 150) As Single
```

To use an array you access each element using the element's index value. The following example fills an array with 10 random numbers in the range 1 to 20 and displays the 6th.

```
Sub RandomArray()
     Dim n As Integer, randNos(10) As Integer
     For n = 1 To 10
          randNos(n) = Int(Rnd() * 20) +1
```

```
     Next
     MsgBox randNos(6)
End Sub
```

If you want to change the size of an array at run-time, say to free up memory, you use a dynamic array. To declare a dynamic array omit the dimensions from the definition, i.e. `Dim dynamicArray()`. Later in the macro allocate the actual number of elements with the **ReDim** statement, `ReDim dynamicArray(x + 1)` where the value of `x` is an integer. The **ReDim** statement can change the number of elements, lower and upper bounds and the dimensions of the array. For example

```
Option Base 1
Sub ChartSheetNames()
' get names of chart sheets in workbook
     Dim chartNames() As String
     Dim i As Integer, chartCount As Integer
     chartCount = Workbooks("test.xls").Charts.Count
     ReDim chartNames(chartCount)
     For i = 1 To chartCount
          chartName(i) = _
                    Workbooks("test.xls").Charts(i).Name
     Next
End Sub
```

Each time the **ReDim** statement is used all current values in the array are lost and all elements are re-initialised. Sometimes, however, you may want to keep the data. To do this use the **Preserve** keyword. To enlarge the dimensions of `oldArray` by 1 but without losing the data do

`ReDim Preserve oldArray(Ubound(oldArray) + 1 ),`

`where Ubound` is a VB built-in function that gets the value of the upper bound of an array. Only the upper bound of the last dimension in a multi-dimensional array can be changed if the **Preserve** keyword is used.

A multi-dimensional array is defined in the statement

`Dim matrix(10, 10) As Single`

to give a 10 x 10 "square" array with 100 elements from `matrix(1,1)` to `matrix(10, 10)`. The statement

`Dim multiArray(4, 1 To 10, 1 To 15)`

produces a three-dimensional array 4 x 10 x 15 with 600 elements.

Loops (see later) are an efficient way to process arrays. The example initialises each element of a 10 x 10 array to its position in the list

```
Option Base 1
Dim i As Integer, j As Integer
Static matrix(1 To 10, 1 To 10) As Single
For i = 1 To 10
     For j =1 To 10
     matrix(i, j) = 10 * i + j
     Next j
Next i
```