Arrays in VBA

Arrays are basically lists, where each item in the list is an indexed element of the array. Although you can usually avoid using them, especially in Excel where it is easy to store data in a worksheet, most programmers find that arrays provide a power and flexibility that cannot be achieved without them.

Declaration

As with all variables, arrays are declared with a **Dim** statement. As well as the data type, however, you must also specify how many elements are in the array. For example, this statement declares two arrays, Employee and Salary:

```
Dim Employee(100) As String, Salary(100) As Currency
```

This **Dim** statement specifies that

- Employee and Salary are arrays, not single variables
- Employee elements are of type String
- Salary elements are of type Currency
- Both arrays have 100 elements

The Option Base Statement

VBA arrays are 0-based by default. This means that indexing starts from 0. Therefore, the first element in the Employee array declared above is Employee(0) and the last element is Employee(99).

In most cases, it's more natural to use 1-based arrays. This can be specified by using the **Option Base** command at the start of a code module (usually after the **Option Explicit** statement), thus:

```
Option Base 1
```

With this statement, the indexes of the Employee and Salary arrays (and any other arrays declared in the module) will start from 1. So the first element in the Employee array will be Employee(1) and the last will be Employee(100).

You can also specify the first and last indexes in the **Dim** statement. This will overrule default indexing. For example:

```
Dim Employee(1 To 100) As String, Salary(1 To 100) As Currency
```

Array Elements

A value can be assigned to an array element just the same as with any other variable, except the index must be specified. The following statements will store "Smith" in the first element and "Jones" in the last:

```
Employee(1) = "Smith"
Employee(100) = "Jones"
```

An attempt to assign data to an element outside the index range will cause an error:

```
Employee(101) = "James"          'Index out-of-bounds error
```

For loops are often used to access successive elements in an array. This code initialises each element of the Salary array:

```
For i = 1 To 100
      Salary(i) = 0.00
Next i
```

Dynamic Indexing and Redim

Once an array has been declared, it is created in memory with a fixed number of elements that cannot be changed. There are situations, however, when the programmer does not know in advance how many elements will be needed. In this case, a dynamic array would be used.

There are two steps to using dynamic arrays

1. Declare the array with a **Dim** statement as before, but leaving the parenthesis empty:

   ```
   Dim Employee() As String
   ```

2. In the body of the sub, set the size of the array when it is known using a **ReDim** statement. This code is typical:

   ```
   N = InputBox("How many employees are in the company?")
   ReDim Employee(N)
   ```

The array can be redimensioned using **ReDim** as many times as necessary, but any data stored in the array will be lost unless the **Preserve** keyword is used. The following code is typical when you want to add another element to the end of the array without loosing the existing elements:

```
N = N + 1
ReDim Preserve Employee(N)
Employee(N) = "Another"
```

Remember, **Redim** will loose the contents of an array unless the keyword **Preserve** is used.

Multiple Dimension Arrays

The arrays seen so far have just one dimension. Arrays with two dimensions require two indexes. This statement declares a two-dimensional array of integers:

```
Dim Table(10, 100) As Integer
```

This statement causes VBA to set aside storage for 10 x 100 = 1,000 integers. To assign a value to an element requires both indexes to be specified:

```
Table(1, 4) = 1024
```

Think of this as storing 1024 in row 1 column 4 of Table().

VBA arrays can be declared with any number of dimensions, but in practice, arrays with more than two dimensions are rarely used.

Array Functions

The **Array** function provides another way of creating an array. Here is an example:

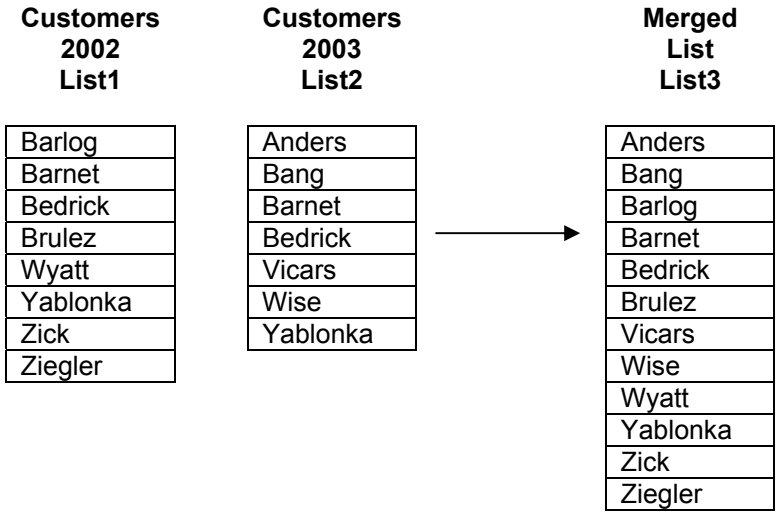
```
Option Base 1

Sub ArrayFunctionExample
     Dim Days As Variant
     Days = Array("Mon", "Tue", "Wed", "Thur", "Fri", "Sat", "Sun")
     MsgBox "The first day in the array is " & Days(1)
End Sub
```

Although Days is an array, it is not declared like a typical array. It must be declared with type **Variant**. You may not use the **Array** function very often, but as this code shows, it provides a convenient way of populating an array. It is useful when you want an array to be initialised with different values for each element.

Merging Two Lists

In this example, two lists of customers are to be merged into a single list in which no customer appears more than once. The two lists are sorted alphabetically, but may be different sizes.



The merge algorithm to do this is:

```
Set Index1 to 1
Set Index2 to 1

Loop while (Index1 <= SizeOfList1) and (Index2 <= SizeOfList2)

        If List1( Index1)  < List2( Index2 )

                Add List1( Index1 ) to List3
                Increment Index1

        Else If List1( Index 1 ) > List2( Index2 )

                Add List2( Index2 ) to List3
                Increment Index2

        Else  ( List1(Index1) = List2(Index2) )

                Add List1( Index1 ) to List3
                Increment Index1
                Increment Index2

        Endif

EndLoop

If there are any names left in List1
        Add them to List3
ElseIf there are any names left in List2
        Add them to List3
Endif
```