

# The Complete Guide to Using Arrays in Excel VBA

 [excelmacromastery.com/excel-vba-array/](http://excelmacromastery.com/excel-vba-array/)

Paul Kelly

06/01/2015



**“A list is only as strong as its weakest link” – Donald Knuth.**

The following table provides a quick reference to **using arrays in VBA**. The remainder of the post provides the most complete guide you will find on the VBA arrays.

Contents [[hide](#)]

- [1 A Quick Guide to VBA Arrays](#)
- [2 Introduction](#)
- [3 Quick Notes](#)
- [4 What are Arrays and Why do You Need Them?](#)
- [5 Types of VBA Arrays](#)
- [6 Declaring an Array](#)
- [7 Assigning Values to an Array](#)
- [8 Using the Array and Split function](#)
- [9 Using Loops With Arrays](#)
  - [9.1 Using the For Each Loop](#)
- [10 Using Erase](#)
- [11 ReDim with Preserve](#)
- [12 Passing an Array to a Sub or Function](#)
- [13 Returning an Array from a Function](#)
- [14 Two Dimensional Arrays](#)
  - [14.1 Using the For Each Loop](#)
- [15 Reading from a Range of Cells to an Array](#)

- [16 How To Make Your Macros Run at Super Speed](#)
- [17 Conclusion](#)
- [18 Arrays Cheat Sheet](#)

## A Quick Guide to VBA Arrays

Task	Static Array	Dynamic Array
Declare	Dim arr(0 To 5) As Long	Dim arr() As Long Dim arr As Variant
Set Size	See Declare above	ReDim arr(0 To 5) As Variant
Increase size (keep existing data)	Dynamic Only	ReDim Preserve arr(0 To 6)
Set values	arr(1) = 22	arr(1) = 22
Receive values	total = arr(1)	total = arr(1)
First position	Ubound(arr)	Ubound(arr)
Last position	LBound(arr)	LBound(arr)
Read all items(1D)	For i = LBound(arr) To UBound(arr) Next i Or For i = LBound(arr,1) To UBound(arr,1) Next i	For i = LBound(arr) To UBound(arr) Next i Or For i = LBound(arr,1) To UBound(arr,1) Next i
Read all items(2D)	For i = LBound(arr,1) To UBound(arr,1) For j = LBound(arr,2) To UBound(arr,2) Next j Next i	For i = LBound(arr,1) To UBound(arr,1) For j = LBound(arr,2) To UBound(arr,2) Next j Next i
Read all items	Dim item As Variant For Each item In arr Next item	Dim item As Variant For Each item In arr Next item
Pass to Sub	Sub MySub(ByRef arr() As String)	Sub MySub(ByRef arr() As String)

Task	Static Array	Dynamic Array
Return from Function	Function GetArray() As Long() Dim arr(0 To 5) As Long GetArray = arr End Function	Function GetArray() As Long() Dim arr() As Long GetArray = arr End Function
Receive from Function	Dynamic only	Dim arr() As Long Arr = GetArray()
Erase array	Erase arr *Resets all values to default	Erase arr *Deletes array
String to array	Dynamic only	Dim arr As Variant arr = Split("James:Earl:Jones",":")
Array to string	Dim sName As String sName = Join(arr, ":")	Dim sName As String sName = Join(arr, ":")
Fill with values	Dynamic only	Dim arr As Variant arr = Array("John", "Hazel", "Fred")
Range to Array	Dynamic only	Dim arr As Variant arr = Range("A1:D2")
Array to Range	Same as Dynamic but array must be two dimensional	Dim arr As Variant Range("A5:D6") = arr

## Introduction

This post provides an **in-depth look at arrays** in the Excel VBA programming language. It covers the important points such as

- Why you need arrays
- When should you use them
- The two types of arrays
- Using more than one dimension
- Declaring arrays
- Adding values
- Viewing all the items
- A super efficient way to read a Range to an array

In the first section we will look at what are arrays and why you need them. You may not understand some of the code in the first section. This is fine. I will be breaking it all down into simple terms in the following sections of the post.

## Quick Notes

Sometimes [Collections](#) are a better option than arrays. You may want to check out my post [The Ultimate Guide To Collections in Excel VBA](#).

Arrays and Loops go hand in hand. The most common loops you use with arrays are the [For Loop](#) and the [For Each Loop\(read-only\)](#).

## What are Arrays and Why do You Need Them?

**A VBA array** is a type of variable. It is used to store lists of data of the same type. An example would be storing a list of countries or a list of weekly totals.

In VBA a normal variable can store only one value at a time.

The following example shows a variable being used to store the marks of a student.

```
' Can only store 1 value at a time
Dim Student1 As Integer
Student1 = 55
```

If we wish to store the marks of another student then we need to create a second variable.

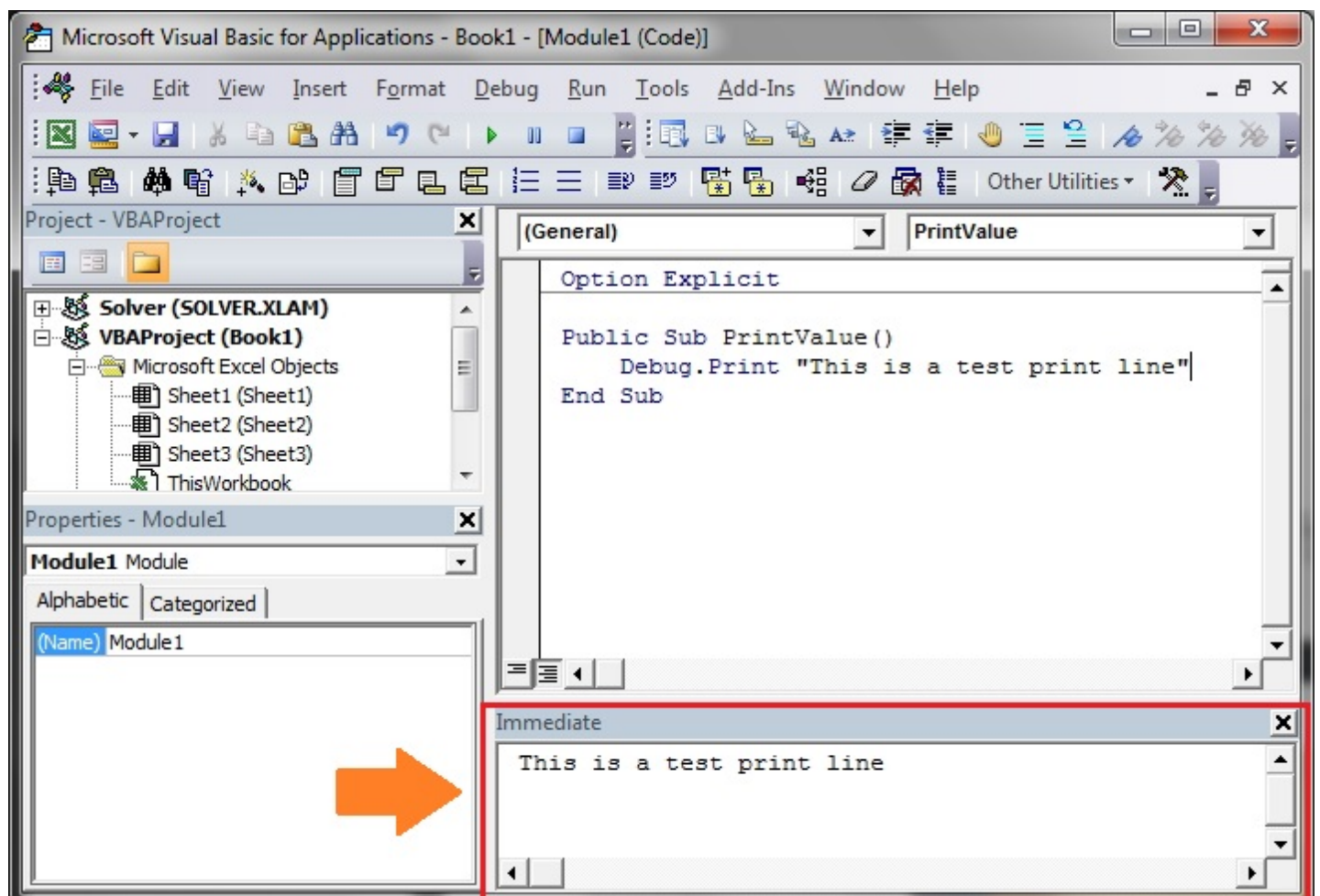
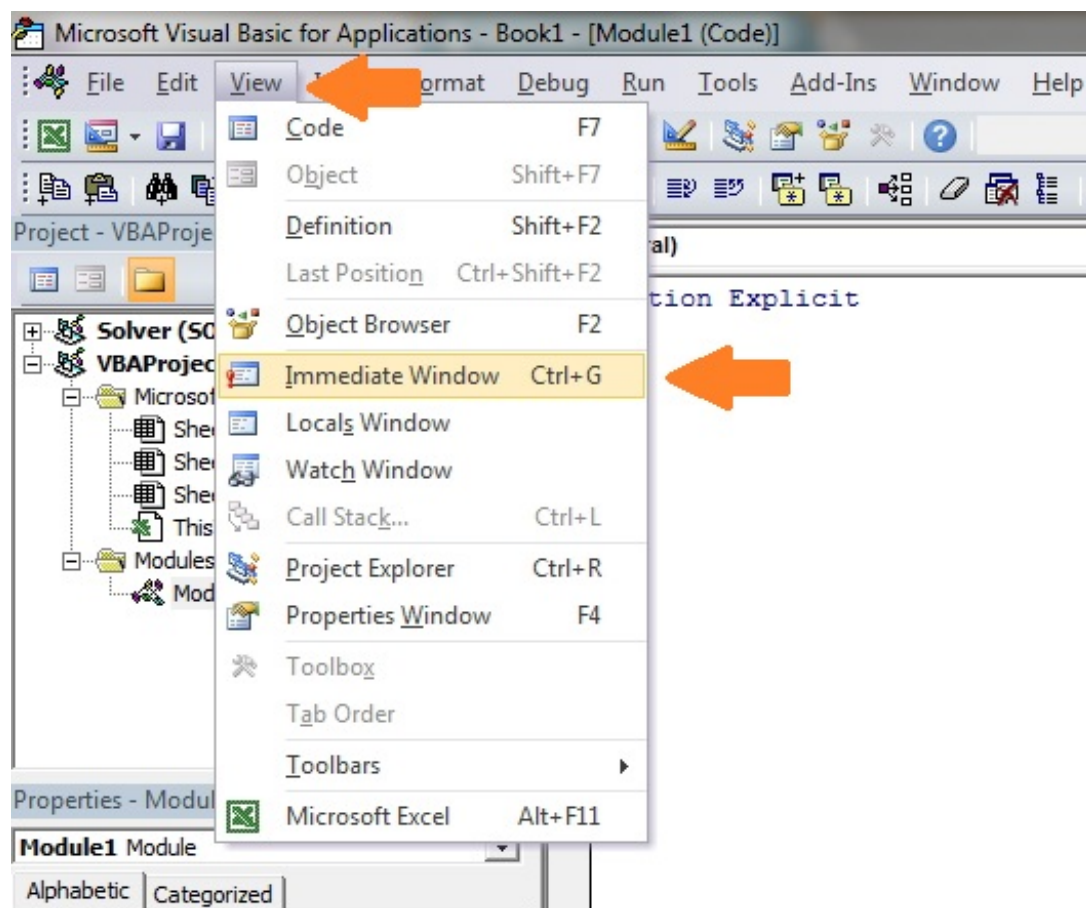
In the following example we have the marks of five students

We are going to read these marks and write them to the Immediate Window.

**Note:** The function `Debug.Print` writes values to the Immediate Window. To view this window select View->Immediate Window from the menu( Shortcut is Ctrl + G)

	A	B	C	D
1				
2		Student ▼	Mark ▼	
3		John	89	
4		Anna	67	
5		Violet	77	
6		Joe	42	
7		Sophia	70	
8				

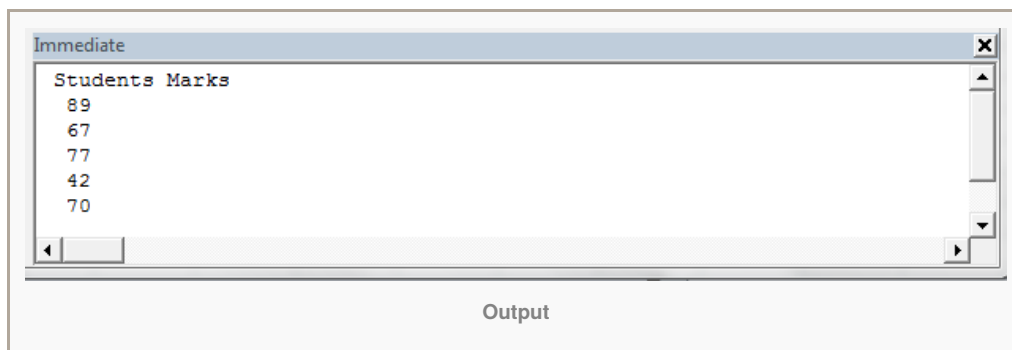
Student Marks



As you can see in the following example we are writing the same code five times – once for each student

```
Public Sub StudentMarks()  
  
    With ThisWorkbook.Worksheets("Sheet1")  
  
        ' Declare variable for each student  
        Dim Student1 As Integer  
        Dim Student2 As Integer  
        Dim Student3 As Integer  
        Dim Student4 As Integer  
        Dim Student5 As Integer  
  
        ' Read student marks from cell  
        Student1 = .Range("C2").Offset(1)  
        Student2 = .Range("C2").Offset(2)  
        Student3 = .Range("C2").Offset(3)  
        Student4 = .Range("C2").Offset(4)  
        Student5 = .Range("C2").Offset(5)  
  
        ' Print student marks  
        Debug.Print "Students Marks"  
        Debug.Print Student1  
        Debug.Print Student2  
        Debug.Print Student3  
        Debug.Print Student4  
        Debug.Print Student5  
  
    End With  
  
End Sub
```

The following is the output from the example



The problem with using one variable per student is that you need to add code for each student. Therefore if you had a thousand students in the above example you would need three thousand lines of code!

Luckily we have arrays to make our life easier. Arrays allow us to store a list of data items in one structure.

The following code shows the above student example using an array

```
Public Sub StudentMarksArr()  
  
    With ThisWorkbook.Worksheets("Sheet1")  
  
        ' Declare an array to hold marks for 5 students  
        Dim Students(1 To 5) As Integer  
  
        ' Read student marks from cells C3:C7 into array  
        Dim i As Integer  
        For i = 1 To 5  
            Students(i) = .Range("C2").Offset(i)  
        Next i  
  
        ' Print student marks from the array  
        Debug.Print "Students Marks"  
        For i = LBound(Students) To UBound(Students)  
            Debug.Print Students(i)  
        Next i  
  
    End With  
  
End Sub
```

The advantage of this code is that it will work for any number of students. If we have to change this code to deal with 1000 students we only need to change the **(1 To 5)** to **(1 To 1000)** in the declaration. In the prior example we would need to add approximately five thousand lines of code.

Let's have a quick comparison of variables and arrays. First we compare the declaration

```
' Variable  
Dim Student As Integer  
Dim Country As String  
  
' Array  
Dim Students(1 To 3) As Integer  
Dim Countries(1 To 3) As String
```

Next we compare assigning a value

```
' assign value to variable  
Student1 = .Cells(1, 1)  
  
' assign value to first item in array  
Students(1) = .Cells(1, 1)
```

Lastly we look at writing the values

```
' Print variable value
Debug.Print Student1

' Print value of first student in array
Debug.Print Students(1)
```

As you can see, using variables and arrays is quite similar.

The fact that arrays use an index(also called a subscript) to access each item is important. It means we can easily access all the items in an array using a For Loop.

Now that you have some background on why arrays are useful lets go through them step by step.

## Types of VBA Arrays

There are two types of arrays in VBA

1. Static – an array of fixed size.
2. Dynamic – an array where the size is set at run time.

The difference between these arrays mainly in how they are created. Accessing values in both array types is exactly the same. In the following sections we will cover both types.

## Declaring an Array

A static array is declared as follows

```
Public Sub DecArrayStatic()

' Create array with locations 0,1,2,3
Dim arrMarks1(0 To 3) As Long

' Defaults as 0 to 3 i.e. locations 0,1,2,3
Dim arrMarks2(3) As Long

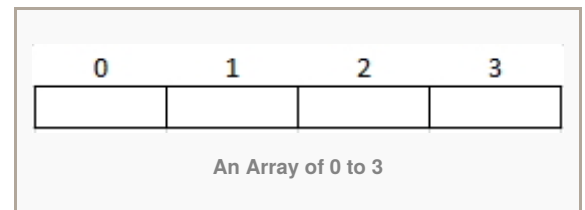
' Create array with locations 1,2,3,4,5
Dim arrMarks1(1 To 5) As Long

' Create array with locations 2,3,4 ' This is rarely used
Dim arrMarks3(2 To 4) As Long
```



End Sub

As you can see the size is specified when you declare a static array. The problem with this is that you can never be sure in advance the size you need. Each time you run the Macro you may have different size requirements.



If you do not use all the array locations then the resources are being wasted. If you need more locations you can use **ReDim** but this is essentially creating a new static array.

The dynamic array does not have such problems. You do not specify the size when you declare it. Therefore you can then grow and shrink as required

```
Public Sub DecArrayDynamic()  
  
    ' Declare dynamic array  
    Dim arrMarks() As Long  
  
    ' Set the size of the array when you are ready  
    ReDim arrMarks(0 To 5)  
  
End Sub
```

The dynamic array is not allocated until you use the ReDim statement. The advantage is you can wait until you know the number of items before setting the array size. With a static array you have to give the size up front.

To give an example. Imagine you were reading worksheets of student marks. With a dynamic array you can count the students on the worksheet and set an array to that size. With a static array you must set the size to the largest possible number of students.

[Need Help Using Arrays? Click here to get your FREE Cheat Sheet](#)

## Assigning Values to an Array

To assign values to an array you use the number of the location. You assign value for both array types the same way.

```
Public Sub AssignValue()  
  
    ' Declare array with locations 0,1,2,3  
    Dim arrMarks(0 To 3) As Long  
  
    ' Set the value of position 0
```

```

arrMarks(0) = 5

' Set the value of position 3
arrMarks(3) = 46

' This is an error as there is no location 4
arrMarks(4) = 99

End Sub

```

The number of the location is called the subscript or index. The last line in the example will give a “Subscript out of Range” error as there is no location 4 in the array example.

0	1	2	3
5	0	0	46

The array with values assigned

## Using the Array and Split function

You can use the **Array** function to populate an array with a list of items. You must declare the array as a type Variant. The following code shows you how to use this function.

```

Dim arr1 As Variant
arr1 = Array("Orange", "Peach", "Pear")

Dim arr2 As Variant
arr2 = Array(5, 6, 7, 8, 12)

```

The array created by the Array Function will start at index zero unless you use **Option Base 1** at the top of your module. Then it will start at index one. In programming it is generally considered poor practice to have your actual data in the code. However sometimes it is useful when you need to test some code quickly. The **Split** function is used to split a string into an array based on a delimiter. A delimiter is a character such as a comma or space that separates the items.

0	1	2
Orange	Peach	Pear

Contents of arr1 after using the Array function

The following code will split the string into an array of three elements.

```

Dim s As String
s = "Red,Yellow,Green,Blue"

Dim arr() As String

```

```
arr = Split(s, ",")
```

The Split function is normally used when you read from a comma separated file or another source that provides a list of items separated by the same character.

0	1	2	3
Red	Yellow	Green	Blue

The array after using Split

## Using Loops With Arrays

Using a [For](#) Loop allows quick access to all items in an array. This is where the power of using arrays becomes apparent. We can read arrays with ten values or ten thousand values using the same few lines of code. There are two functions in VBA called LBound and UBound. These functions return the smallest and largest subscript in an array. In an array arrMarks(0 to 3) the LBound will return 0 and UBound will return 3.

The following example assigns random numbers to an array using a loop. It then prints out these numbers using a second loop.

```
Public Sub ArrayLoops()  
  
    ' Declare array  
    Dim arrMarks(0 To 5) As Long  
  
    ' Fill the array with random numbers  
    Dim i As Long  
    For i = LBound(arrMarks) To UBound(arrMarks)  
        arrMarks(i) = 5 * Rnd  
    Next i  
  
    ' Print out the values in the array  
    Debug.Print "Location", "Value"  
    For i = LBound(arrMarks) To UBound(arrMarks)  
        Debug.Print i, arrMarks(i)  
    Next i  
  
End Sub
```

The functions **LBound** and **UBound** are very useful. Using them means our loops will work correctly with any array size. The real benefit is that if the size of the array changes we do not have to change the code for printing the values. A loop will work for an array of any size as long as you use these functions.

## Using the For Each Loop

You can use the **For Each** loop with arrays. The important thing to keep in mind is that it is **Read-Only**. This means that you cannot change the value in the array.

In the following code the value of **mark** changes but it does not change the value in the array.

```
For Each mark In arrMarks
    ' Will not change the array value
    mark = 5 * Rnd
Next mark
```

The For Each is loop is fine to use for reading an array. It is neater to write especially for a **Two-Dimensional** array as we will see.

```
Dim mark As Variant
For Each mark In arrMarks
    Debug.Print mark
Next mark
```

## Using Erase

The **Erase** function can be used on arrays but performs differently depending on the array type.

For a static Array the Erase function resets all the values to the default. If the array is of integers then all the values are set to zero. If the array is of strings then all the strings are set to "" and so on.

For a Dynamic Array the Erase function DeAllocates memory. That is, it deletes the array. If you want to use it again you must use **ReDim** to Allocate memory.

Lets have a look an example for the static array. This example is the same as the **ArrayLoops** example in the last section with one difference – we use Erase after setting the values. When the value are printed out they will all be zero.

```
Public Sub EraseStatic()

    ' Declare array
    Dim arrMarks(0 To 3) As Long

    ' Fill the array with random numbers
    Dim i As Long
    For i = LBound(arrMarks) To UBound(arrMarks)
        arrMarks(i) = 5 * Rnd
    Next i

    ' ALL VALUES SET TO ZERO
```

```

Erase arrMarks

' Print out the values - there are all now zero
Debug.Print "Location", "Value"
For i = LBound(arrMarks) To UBound(arrMarks)
    Debug.Print i, arrMarks(i)
Next i

End Sub

```

We will now try the same example with a dynamic. After we use Erase all the locations in the array have been deleted. We need to use ReDim if we wish to use the array again.

If we try to access members of this array we will get a “**Subscript out of Range**” error.

```

Public Sub EraseDynamic()

    ' Declare array
    Dim arrMarks() As Long
    ReDim arrMarks(0 To 3)

    ' Fill the array with random numbers
    Dim i As Long
    For i = LBound(arrMarks) To UBound(arrMarks)
        arrMarks(i) = 5 * Rnd
    Next i

    ' arrMarks is now deallocated. No locations exist.
    Erase arrMarks

End Sub

```

## ReDim with Preserve

If we use **ReDim** on an existing array, then the array and it's contents will be deleted.

In the following example, the second **ReDim** statement will create a completely new array. The original array and it's contents will be deleted.

```

Sub UsingRedim()

    Dim arr() As String

    ' Set array to be slots 0 to 2
    ReDim arr(0 To 2)
    arr(0) = "Apple"

    ' Array with apple is now deleted

```

```
ReDim arr(0 To 3)
```

```
End Sub
```

If we want to extend the size of an array without losing the contents, we can use the **Preserve** keyword.

When we use **Redim Preserve** the new array must be bigger and start at the same dimension e.g.

We cannot Preserve from (0 to 2) to (1 to 3) or (2 to 10) as they are different starting dimensions.

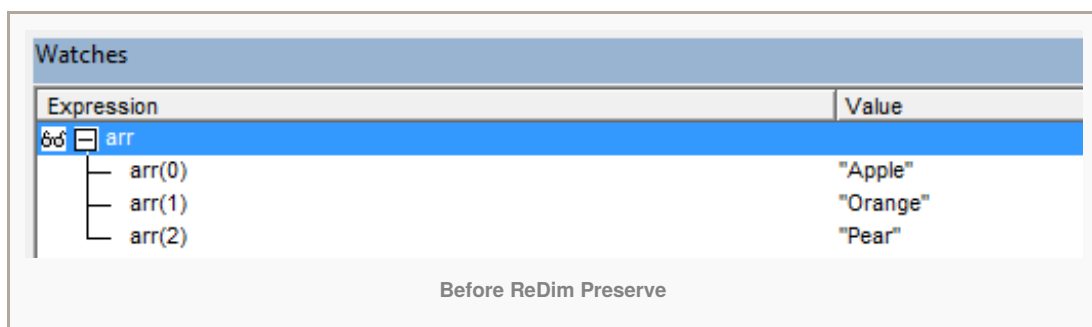
We cannot Preserve from (0 to 2) to (0 to 1) or (0) as they are smaller than original array.

In the following code we create an array using **ReDim** and then fill the array with types of fruit.

We then use **Preserve** to extend the size of the array so we don't lose the original contents.

```
Sub UsingRedimPreserve()  
  
    Dim arr() As String  
  
    ' Set array to be slots 0 to 1  
    ReDim arr(0 To 2)  
    arr(0) = "Apple"  
    arr(1) = "Orange"  
    arr(2) = "Pear"  
  
    ' Resize and keep original contents  
    ReDim Preserve arr(0 To 5)  
  
End Sub
```

You can see from the screenshots below, that the original contents of the array have been “Preserved”.





**Word of Caution:** In most cases you shouldn't need to resize an array like we have done in this section. If you are resizing an array multiple times then you may want to consider using a [Collection](#).

## Passing an Array to a Sub or Function

Sometimes you will need to pass an array to a procedure. You declare the parameter using parenthesis similar to how you declare a dynamic array.

Passing to the procedure using ByRef means you are passing a reference of the array. So if you change the array in the procedure it will be changed when you return.

**Note:** It is not possible to pass an array using ByVal.

```
' Passes array to a Function
Public Sub PassToProc()
    Dim arr(0 To 5) As String
    ' Pass the array to function
    UseArray arr
End Sub

Public Function UseArray(ByRef arr() As String)
    ' Use array
    Debug.Print UBound(arr)
End Function
```

## Returning an Array from a Function

It is important to keep the following in mind. If you want to change an existing array in a procedure then you should pass it as a parameter using ByRef(see last section). You do not need to return the array from the procedure.

The main reason for returning an array is when you use the procedure to create a new one. In this case you assign the return array to an array in the caller. This array cannot be already allocated. In other words you must

use a dynamic array that has not been allocated.

The following examples show this

```
Public Sub TestArray()  
  
    ' Declare dynamic array - not allocated  
    Dim arr() As String  
    ' Return new array  
    arr = GetArray  
  
End Sub  
  
Public Function GetArray() As String()  
  
    ' Create and allocate new array  
    Dim arr(0 To 5) As String  
    ' Return array  
    GetArray = arr  
  
End Function
```

## Two Dimensional Arrays

The arrays we have been looking at so far have been one dimensional arrays. This means the arrays are one list of items.

A two dimensional array is essentially a list of lists. If you think of a single spreadsheet column as a single dimension then more than one column is two dimensional. In fact a spreadsheet is the equivalent of a 2 dimensional array. It has two dimensions – rows and columns.

The following image shows two groups of data. The first is a one dimensional layout and the second is two dimensional.



Class 1		Class 1	Class 2	Class 3
55		58	57	71
67		92	96	68
87		96	59	69
54		69	76	91



One dimensional



Two dimensional

To access an item in the first set of data(1 dimensional) all you need to do is give the row e.g. 1,2, 3 or 4.

For the second set of data(2 dimensional) you need to give the row AND the column. So you can think of 1 dimensional being rows only and 2 dimensional as being rows and columns.

**Note:** It is possible to have more dimensions in an array. It is rarely required. If you are solving a problem using a 3+ dimensional array then there probably is a better way to do it.

You declare a 2 dimensional array as follows

```
Dim ArrayMarks(0 To 2,0 To 3) As Long
```

The following example creates a random value for each item in the array and the prints the values to the Immediate Window.

```
Public Sub TwoDimArray()

    ' Declare a two dimensional array
    Dim arrMarks(0 To 3, 0 To 2) As String

    ' Fill the array with text made up of i and j values
    Dim i As Long, j As Long
    For i = LBound(arrMarks) To UBound(arrMarks)
        For j = LBound(arrMarks, 2) To UBound(arrMarks, 2)
            arrMarks(i, j) = CStr(i) & ":" & CStr(j)
        Next j
    Next i

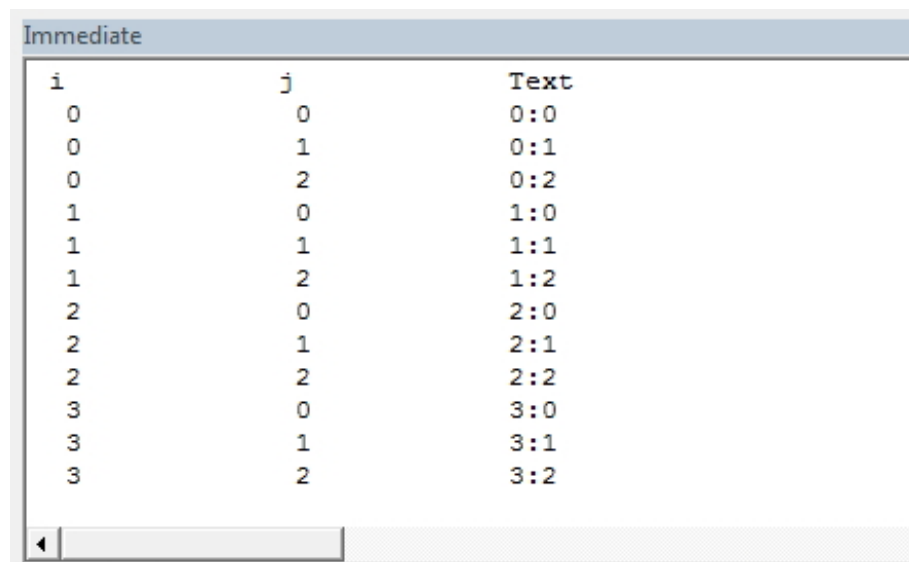
    ' Print the values in the array to the Immediate Window
    Debug.Print "i", "j", "Value"
    For i = LBound(arrMarks) To UBound(arrMarks)
        For j = LBound(arrMarks, 2) To UBound(arrMarks, 2)
            Debug.Print i, j, arrMarks(i, j)
        Next j
    Next i
End Sub
```

```
Next i
```

```
End Sub
```

You can see that we use a second [For](#) loop inside the first loop to access all the items.

The output of the example looks like this:



i	j	Text
0	0	0:0
0	1	0:1
0	2	0:2
1	0	1:0
1	1	1:1
1	2	1:2
2	0	2:0
2	1	2:1
2	2	2:2
3	0	3:0
3	1	3:1
3	2	3:2

How this Macro works is as follows

- Enters the **i** loop
- **i** is set to 0
- Enters **j** loop
- **j** is set to 0
- **j** is set to 1
- **j** is set to 2
- Exit **j** loop
- **i** is set to 1
- **j** is set to 0
- **j** is set to 1
- **j** is set to 2
- And so on until **i**=3 and **j**=2

You may notice that **LBound** and **UBound** have a second argument of 2. This specifies that it is the upper or lower bound of the second dimension. That is the start and end location for **j**. The default value 1 which is why

we do not need to specify it for the i loop.

## Using the For Each Loop

Using a For Each is neater to use when reading from an array.

Let's take the code from above that writes out the two-dimensional array

```
' Using For loop needs two loops
Debug.Print "i", "j", "Value"
For i = LBound(arrMarks) To UBound(arrMarks)
    For j = LBound(arrMarks, 2) To UBound(arrMarks, 2)
        Debug.Print i, j, arrMarks(i, j)
    Next j
Next i
```

Now let's rewrite it using a For each loop. You can see we only need one loop and so it is much easier to write

```
' Using For Each requires only one loop
Debug.Print "Value"
Dim mark As Variant
For Each mark In arrMarks
    Debug.Print mark
Next mark
```

Using the For Each loop gives us the array in one order only – from LBound to UBound. Most of the time this is all you need.

## Reading from a Range of Cells to an Array

If you have read my previous post on [Cells and Ranges](#) then you will know that VBA has an extremely efficient way of reading from a Range of Cells to an Array and vice versa

```
Public Sub ReadToArray()

    ' Declare dynamic array
    Dim StudentMarks As Variant

    ' Read values into array from first row
    StudentMarks = Range("A1:Z1").Value

    ' Write the values back to the third row
    Range("A3:Z3").Value = StudentMarks
```

End Sub

The dynamic array created in this example will be a two dimensional array. As you can see we can read from an entire range of cells to an array in just one line.

The next example will read the sample student data below from C3:E6 of Sheet1 and print them to the Immediate Window.

```
Public Sub ReadAndDisplay()  
  
    ' Get Range  
    Dim rg As Range  
    Set rg = ThisWorkbook.Worksheets("Sheet1").Range("C3:E6")  
  
    ' Create dynamic array  
    Dim StudentMarks As Variant  
  
    ' Read values into array from sheet1  
    StudentMarks = rg.Value  
  
    ' Print the array values  
    Debug.Print "i", "j", "Value"  
    Dim i As Long, j As Long  
    For i = LBound(StudentMarks) To UBound(StudentMarks)  
        For j = LBound(StudentMarks, 2) To UBound(StudentMarks, 2)  
            Debug.Print i, j, StudentMarks(i, j)  
        Next j  
    Next i  
  
End Sub
```

	A	B	C	D	E	F
1						
2		Student ▼	Maths ▼	History ▼	Biology ▼	
3		Noah	41	22	82	
4		Jane	57	58	44	
5		Sophia	16	19	84	
6		Pat	38	25	33	
7						

Sample Student data

As you can see the first dimension (accessed using `i`) of the array is a row and the second is a column. To demonstrate this take a look at the value 44 in E4 of the sample data. This value is in row 2 column 3 of our data. You can see that 44 is stored in the array at **StudentMarks(2,3)**.

Immediate		
i	j	Value
1	1	41
1	2	22
1	3	82
2	1	57
2	2	58
2	3	44
3	1	16
3	2	19
3	3	84
4	1	38
4	2	25
4	3	33

Output from sample data

## How To Make Your Macros Run at Super Speed

If your macros are running very slow then you may find this section very helpful. Especially if you are dealing with large amounts of data. The following is a well kept secret in VBA

**Updating values in arrays is exponentially faster than updating values in cells.**

In the last section, you saw how we can easily read from a group of cells to an array and vice versa. If we are updating a lot of values then we can do the following

1. Copy the data from the cells to an array.
2. Change the data in the array.
3. Copy the updated data from the array back to the cells.

For example, the following code would be much faster than the code below it

```
Public Sub ReadToArray()

    ' Read values into array from first row
    Dim StudentMarks As Variant
    StudentMarks = Range("A1:Z20000").Value

    Dim i As Long
    For i = LBound(StudentMarks) To UBound(StudentMarks)
        ' Update marks here
        StudentMarks(i, 1) = StudentMarks(i, 1) * 2
        ' ...
    Next i

    ' Write the new values back to the worksheet
```

```

Range("A1:Z20000").Value = StudentMarks

End Sub

```

```

Sub UsingCellsToUpdate()

    Dim c As Variant
    For Each c In Range("A1:Z20000")
        c.Value = ' Update values here
    Next c

End Sub

```

Assigning from one set of cells to another is also much faster than using Copy and Paste

```

' Assigning - this is faster
Range("A1:A10").Value = Range("B1:B10").Value

' Copy Paste - this is slower
Range("B1:B1").Copy Destination:=Range("A1:A10")

```

The following comments are from two readers who used arrays to speed up their macros

*“A couple of my projects have gone from almost impossible and long to run into almost too easy and a reduction in time to run from 10:1.” – Dane*

*“One report I did took nearly 3 hours to run when accessing the cells directly — 5 minutes with arrays” – Jim*

## Conclusion

The following is a summary of the main points of this post

1. Arrays are an efficient way of storing a **list of items** of the same type.
2. You can access an array item directly using the number of the location which is known as the **subscript or index**.
3. The common error “**Subscript out of Range**” is caused by accessing a location that does not exist.
4. There are two types of arrays: **Static** and **Dynamic**.
5. **Static** is used when the size of the array is always the same.
6. **Dynamic** arrays allow you to determine the size of an array at run time.
7. **LBound** and **UBound** provide a safe way of find the smallest and largest subscripts of the array.
8. The basic array is **one dimensional**. You can also have multi dimensional arrays.
9. You can only pass an array to a procedure using **ByRef**. You do this like this: ByRef arr() as long.

10. You can **return an array** from a function but the array, it is assigned to, must not be currently allocated.
11. A worksheet with it's rows and columns is essentially a **two dimensional** array.
12. You can read directly **from a worksheet range** into a two dimensional array in just one line of code.
13. You can also write from a two dimensional **array to a range** in just one line of code.

I hope you enjoyed this post and found it beneficial. You may want to check out one of our most popular posts [The Ultimate Guide to the VBA String](#)

## Arrays Cheat Sheet

[Need Help Using Arrays? Click here to get your FREE Cheat Sheet](#)

**Note: I periodically archive comments to maintain the page speed.**