EXCEL VBA FUNCTIONS – MIKE STAUNTON

The key advantage of VBA as a programming language is its simplicity but first a few words of warning about the programming environment. Firstly, it does not take kindly to errors, as for example accidentally misspelling the name of one of your variables. Like a stubborn donkey, the whole code sheet will refuse to compile until the mistake is rectified and the blue square reset button pressed. The moral of this is to type carefully in the knowledge that "right first time" should be your objective. Secondly, be prepared for the occasional unwitting path that leaves one at the end of a cul-de-sac. I prefer to keep all the module sheets and code internal to each spreadsheet file but sometimes Excel believes that your functions are instead taken from module sheets in another file. If this happens, your only hope is to copy your function code onto the clipboard, delete the module sheet, save your spreadsheet, exit Excel and reopen your spreadsheet, create a new module sheet and then paste the function code into it. The remaining hurdle is to ensure that your functions can run when the accompanying spreadsheet file is opened. Your functions will not be able to run if the macro security level is set to high or the security level is set to medium and you press the disable macros button when prompted. And, when working in Office 12, you will need to save your file with the correct extension.

Now that we have got the government health warnings out of the way, a few brief thoughts on my programming style. I'm concerned with writing code that is straightforward to understand and yet that will also run as quickly as possible. Along the way, I use short names for variables, delight in the use of type declaration characters, take especial care to distinguish between vectors and matrices, adopt a style that will facilitate the translation of my code into other programming languages and am relatively sparing in my use of comments. I encourage the use of creating new variables that reduce the number of repeated calculations (particularly important when writing code that will be run many times within loops) and try to retain as much consistency with the choice of variable names across all the functions that I write. In the past, I probably tended to retain too much of the modular approaching to programming by retaining too many calls to separate functions - now, I prefer to eliminate repeated calls to too many other single-line functions by keeping more code in the original function and making fewer calls to other functions. It is worth spending some time on getting from the code that gives the correct answer to the code that in addition is the quickest to run.

Within the model sheet, you should follow the following order of use: first VBA syntax then the VBA math functions and finally any Excel functions that can be used in VBA.

To confirm the simplicity of the VBA language, I propose to detail the ten functions that contain all the programming syntax that I will need to write all the functions in this book. You should spend time understanding what these ten functions do and why, safe in the knowledge that you can safely ignore almost all of the rest of the VBA language. Any additional syntax that you need can be found in the VBA help.

The VBA language is not case sensitive and so one useful technique is to predominantly type in lower case and allow VBA to automatically recognize commands and add the appropriate capitals. You will of course need to use upper case characters when wished within the names of functions, arguments and variable names.

The Option Explicit statement at the top of my module sheets forces me to declare all arguments and variables that I use, thus giving me warning when I misspell names. The Option Base 0 statement means that new size arrays that I create in VBA will have a lower bound of zero as opposed to the lower bound of one in Excel.

Nearly all of my variables will have a type that is either Double (or # symbol) or Integer (or % symbol) or Long (or & symbol) with the rest taking the default Variant type. Most variables in VBA take their size from the assignment statement that gives them their value but others do need to be given a size and I will call these size variables.

Function vaF01 illustrates the very simplest function, that accepts a single argument (tyr) and assigns the value of its square root, using the VBA Math Sqr function. The equals sign ensures that the calculation on the right-hand side is merely assigned to the variable on the left-hand side. The argument and the value that the function returns are given my favourite type of Double via the type declaration character #. When Excel is searching for functions to use, it will select first any user-defined functions that match the name followed next by any of the Math functions such as Sqr. There are only twelve members of the VBA Math class and using the Object Browser (F2) from the VB Editor you can see them all listed.

```
Function vaF01(tyr#) As Double
   vaF01 = Sqr(tyr)
End Function
```

Function vaF02 shows how we can call the new function that we've just written whilst also using another one of the VBA Math functions, Log. Note that since arguments in VBA are passed by reference, the value of arguments can be changed within functions. If you do not want the value of an argument to be changed, you must pass the argument by value by prefixing the argument name with the ByVal keyword. I prefer to call my VBA functions from Excel by using the Function Wizard as this ensures that you can point to the arguments in the cells and hence minimise the number of errors that you make. Remember that you will have to scroll down to include the sixth input argument as the wizard only displays boxes for the first five arguments.

```
Function vaF02(S#, K#, r#, q#, tyr#, sigma#) As Double
   vaF02 = (Log(S / K) + (r - q + 0.5 * sigma * sigma) * tyr) / (sigma * vaF01(tyr))
End Function
```

Function vaF03 creates a new intermediate variable through the Dim statement. It then uses one of the Excel functions, NormSDist, that must be prefaced with WorksheetFunction. (or Application. for earlier versions of Excel). You can see all the members of the WorksheetFunction class within the Excel library that can be used within VBA by looking at the Object Browser. In my style of programming, when you need to call the vaF03 function from another function, the arguments are called by position in preference to name (so that I could have changed the names of the arguments in the calling function as long as I preserved the correct order).

```
Function vaF03(S#, K#, r#, q#, tyr#, sigma#) As Double
   Dim d1
   d1 = vaF02(S, K, r, q, tyr, sigma)
   vaF03 = WorksheetFunction.NormSDist(d1 - sigma * Sqr(tyr))
End Function
```

Function vaF04 introduces the first of the formal programming elements, here using the Select Case statement to return the appropriate value of y corresponding to the value of the single argument x.

```
Function vaF04(x#) As Double
   Dim y#
   Select Case x
   Case 0.5, 1.4
      y = 8.5
   Case Is > 2
      y = 6.3
   Case Else
      y = -10
   End Select
   vaF04 = y
End Function
```

Function vaF05 demonstrates the If statement syntax that can cope with more complex correspondences. You can include as many ElseIf statements as you reasonably wish, use more a single variable as well as a wide variety of conditions that you wish to test. For very simple conditions, you can use the single-line If Then statement.

```
Function vaF05(i%, j%) As Double
   Dim y#
   If i = 1 Or j <= 3 Then
      y = 3.2
   ElseIf i > 2 And j <> 5 Then
      y = 1.6
   Else
      y = -1
   End If
   vaF05 = y
End Function
```

Function vaF06 adopts the powerful For loop syntax to sum the elements of a two-dimensional matrix, where matrix is passed into the VBA function with the default type (Variant). Each of the dimensions can have any size and the VBA function will adapt by assigning the appropriate upper bounds to the variables nr1 and nc1. Note that the VBA language refers to matrices by rows then columns in contrast to Excel that refers to call addresses by columns then rows. It is good practice to set the variable to be used to store the sum to a suitable value (here 0) before entering the calculation loop. The summation will start with the first value of the counter variable i for the outside loop (here 1) then proceed through the values of the counter variable j for the inside loop (here 1 then 2 as the default increment is 1). Almost none of the For loops that I write have more than three nested counter variables and it is good practice to ensure that counter variables have type Integer or Long.

```
Function vaF06 (Amat1) As Integer
  Dim nr1%, nc1%, i%, j%, sumij%
  nr1 = Amat1.Rows.Count
  nc1 = Amat1.Columns.Count
  sumij = 0
  For i = 1 To nr1
    For j = 1 To nc1
      sumij = sumij + Amat1(i, j)
    Next j
  Next i
  vaF06 = sumij
End Function
```

Function vaF07 illustrates the Do loop syntax by generating uniform random numbers until they are greater than the input argument xmin and returns both the last random value and the number of random drawings in an array of fixed length. This also illustrates how to create a variable with fixed size that here, with lower bound zero, will have size two.

```
Function vaF07(xmin#)
  Dim rnd01#
  Dim i%
  Dim avec(1) As Double
  Application.Volatile
  i = 0
  Do
    rnd01 = Rnd
    i = i + 1
  Loop Until rnd01 > xmin
  avec(0) = rnd01
  avec(1) = i
  vaF07 = avec
End Function
```

Function vaF08 accepts two matrices as arguments are returns their matrix product as a Variant back into the spreadsheet (and so you will need to mark out the necessary dimensions for the answer range before using the function wizard and remember to hold down the Ctrl+Shift+Enter key stroke combination to return the complete answer). It took me many months to appreciate that you could use the Matrix functions from Excel in VBA. The trick is to declare the VBA variable that holds the answer from the matrix multiplication as default type Variant, without the brackets that would give either a fixed or dynamic size array.

```
Function vaF08(Amat1, Amat2)
  Dim Amat12
  Amat12 = WorksheetFunction.MMult(Amat1, Amat2)
  vaF08 = Amat12
End Function
```

Function vaF09 builds an array with dynamic size, employing first the Dim statement with empty brackets and then the ReDim statement that gives the variable the required upper bound. Remember that since our module sheet has lower bound zero, the dynamic vector will have size ncol+1. My preference is not to declare a lower bound, partly because I prefer to use the Option Base statement to set the lower bound and partly for compatibility with other languages.

```
Function vaF09(pn%, ncol%)
    Dim j%
    Dim bvec0() As Double
    ReDim bvec0(ncol)

    For j = 0 To ncol
        bvec0(j) = pn^j
    Next j
    vaF09 = bvec0
End Function
```

Function vaF10 is my attempt to sidestep the debugging tools that VBA provides by writing intermediate values of chosen variables straight to a text output file (where here #1 refers to file number 1). Life is too short to step slowly through calculations, much better to write it all to a file and diagnose carefully where the problem is. Note the negative Step increment in the For statement. The text file will typically be created in the directory last used to open an Excel file but can also be found using the Windows Search function.

```
Function vaF10()
    Dim i%, j%
    Open "SourceFile10.txt" For Output As #1
    Write #1, "i then j then i*j"
    For i = 1 To 11 Step 2
    For j = 15 To 3 Step -3
        Write #1, i, j, i * j
    Next j
    Next i
    Close #1
End Function
```

Mike Staunton: Selected parts of VBA Help

Create a Procedure

Code within a module is organized into procedures. A procedure tells the application how to perform a specific task. Use procedures to divide complex code tasks into more manageable units.

To create a procedure by writing code

1. Open the module for which you want to write the procedure.
2. You can create a **Sub**, **Function**, or **Property** procedure.
3. Type **Sub**, **Function**, or **Property**.

   Press F1 to get Help with syntax, if necessary.

4. Type code for the procedure.

   Visual Basic concludes the procedure with the appropriate **End Sub**, **End Function**, or **End Property** statement.

I will concentrate on writing user-defined Functions

Writing a Function Procedure

A **Function** procedure is a series of Visual Basic statements enclosed by the **Function** and **End Function** statements. A **Function** procedure is similar to a **Sub** procedure, but a function can also return a value. A **Function** procedure can take arguments, such as constants, variables, or expressions that are passed to it by a calling procedure. If a **Function** procedure has no arguments, its **Function** statement must include an empty set of parentheses. A function returns a value by assigning a value to its name in one or more statements of the procedure.

In the following example, the **Celsius** function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the **Main** procedure, a variable containing the argument value is passed to the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub Main()
    temp = Application.InputBox(Prompt:= _
        "Please enter the temperature in degrees F.", Type:=1)
    MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub

Function Celsius(fDegrees)
    Celsius = (fDegrees - 32) * 5 / 9
End Function
```

Function Statement Example

This example uses the **Function** statement to declare the name, arguments, and code that form the body of a **Function** procedure. The last example uses hard-typed, initialized **Optional** arguments.

```
' The following user-defined function returns the square root of the
' argument passed to it.
Function CalculateSquareRoot(NumberArg As Double) As Double
    If NumberArg < 0 Then    ' Evaluate argument.
        Exit Function    ' Exit to calling procedure.
    Else
        CalculateSquareRoot = Sqr(NumberArg)    ' Return square root.
    End If
End Function
```

Function Statement

Declares the name, arguments, and code that form the body of a **Function** procedure.

Syntax

[**Public** | **Private | Friend**] [**Static**] **Function** *name* [(*arglist*)] [**As** *type*]
[*statements*]
[*name* = *expression*]
[**Exit Function**]
[*statements*]
[*name* = *expression*]

End Function

The **Function** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Function** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private**, the procedure is not available outside the project.
Private	Optional. Indicates that the **Function** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Function** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Function** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Function**, even if they are used in the procedure.
name	Required. Name of the **Function**; follows standard variable naming conventions.
arglist	Optional. List of variables representing arguments that are passed to the **Function** procedure when it is called. Multiple variables are separated by commas.
type	Optional. Data type of the value returned by the **Function** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, or (except fixed length), Object, Variant, or any user-defined type.
statements	Optional. Any group of statements to be executed within the **Function** procedure.
expression	Optional. Return value of the **Function**.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. It may not be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (not currently supported) **Date**, **String** (variable length only), **Object**, **Variant**, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Function** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure does't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

Caution **Function** procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow. The **Static** keyword usually isn't used with recursive **Function** procedures.

All executable code must be in procedures. You can't define a **Function** procedure inside another **Function**, **Sub**, or **Property** procedure.

The **Exit Function** statement causes an immediate exit from a **Function** procedure. Program execution continues with the statement following the statement that called the **Function** procedure. Any number of **Exit Function** statements can appear anywhere in a **Function** procedure.

Like a **Sub** procedure, a **Function** procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a **Sub** procedure, you can use a **Function** procedure on the right side of an expression in the same way you use any intrinsic function, such as **Sqr**, **Cos**, or **Chr**, when you want to use the value returned by the function.

You call a **Function** procedure using the function name, followed by the argument list in parentheses, in an expression. See the **Call** statement for specific information on how to call **Function** procedures.

To return a value from a function, assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to *name*, the procedure returns a default value: a numeric function returns 0, a string function returns a zero-length string (""), and a **Variant** function returns Empty. A function that returns an object reference returns **Nothing** if no object reference is assigned to *name* (using **Set**) within the **Function**.

The following example shows how to assign a return value to a function named `BinarySearch`. In this case, **False** is assigned to the name to indicate that some value was not found.

```
Function BinarySearch(. . .) As Boolean
. . .
    ' Value not found. Return a value of False.
    If lower > upper Then
        BinarySearch = False
        Exit Function
    End If
. . .
End Function
```

Variables used in **Function** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

Understanding Named and Optional Arguments

I call functions where the arguments are supplied by position

When you call a **Sub** or **Function** procedure, you can supply arguments positionally, in the order they appear in the procedure's definition, or you can supply the arguments by name without regard to position.

For example, the following **Sub** procedure takes three arguments:

```
Sub PassArgs(strName As String, intAge As Integer, dteBirth As Date)
    Debug.Print strName, intAge, dteBirth
End Sub
```

You can call this procedure by supplying its arguments in the correct position, each delimited by a comma, as shown in the following example:

```
PassArgs "Mary", 29, #2-21-69#
```

You can also call this procedure by supplying named arguments, delimiting each with a comma.

```
PassArgs intAge:=29, dteBirth:=#2/21/69#, strName:="Mary"
```

A named argument consists of an argument name followed by a colon and an equal sign (**:=**), followed by the argument value.

Named arguments are especially useful when you are calling a procedure that has optional arguments. If you use named arguments, you don't have to include commas to denote missing positional arguments. Using named arguments makes it easier to keep track of which arguments you passed and which you omitted.

I will not use optional arguments

Optional arguments are preceded by the **Optional** keyword in the procedure definition. You can also specify a default value for the optional argument in the procedure definition. For example:

```
Sub OptionalArgs(strState As String, Optional strCountry As String = "USA")
. . .
End Sub
```

Declaring Variables

When declaring variables, you usually use a **Dim** statement. A declaration statement can be placed within a procedure to create a procedure-level variable. Or it may be placed at the top of a module, in the Declarations section, to create a module-level variable.

The data types I use will be Double and Integer (for single-cell variables) and Variant (for vectors and matrices). I use type-declaration characters: # for Double and % for Integer.

The following example creates the variable `strName` and specifies the String data type.

```
Dim strName As String
```

If this statement appears within a procedure, the variable `strName` can be used only in that procedure. If the statement appears in the Declarations section of the module, the variable `strName` is available to all procedures within the module, but not to procedures in other modules in the project. To make this variable available to all procedures in the project, precede it with the **Public** statement, as in the following example:

```
Public strName As String
```

Variables can be declared as one of the following data types: **Boolean**, **Byte**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Date**, **String** (for variable-length strings), **String** * *length* (for fixed-length strings), **Object**, or **Variant**. If you do not specify a data type, the **Variant** data type is assigned by default.

You can declare several variables in one statement. To specify a data type, you must include the data type for each variable. In the following statement, the variables `intX`, `intY`, and `intZ` are declared as type **Integer**.

```
Dim intX As Integer, intY As Integer, intZ As Integer
```

In the following statement, `intX` and `intY` are declared as type **Variant**; only `intZ` is declared as type **Integer**.

```
Dim intX, intY, intZ As Integer
```

You don't have to supply the variable's data type in the declaration statement. If you omit the data type, the variable will be of type **Variant**.

Using the Option Explicit Statement

You can implicitly declare a variable in Visual Basic simply by using it in an assignment statement. All variables that are implicitly declared are of type **Variant**. Variables of type **Variant** require more memory resources than most other variables. Your application will be more efficient if you declare variables explicitly and with a specific data type. Explicitly declaring all variables reduces the incidence of naming-conflict errors and spelling mistakes.

If you don't want Visual Basic to make implicit declarations, you can place the **Option Explicit** statement in a module before any procedures. This statement requires you to explicitly declare all variables within the module. If a module includes the **Option Explicit** statement, a compile-time error will occur when Visual Basic encounters a variable name that has not been previously declared, or that has been spelled incorrectly.

You can set an option in your Visual Basic programming environment to automatically include the **Option Explicit** statement in all new modules. See your application's documentation for help on how to change Visual Basic environment options. Note that this option does not change existing code you have written.

Note You must explicitly declare fixed arrays and dynamic arrays

Data Type Summary

The following table shows the supported data types, including storage sizes and ranges.

Data type	Storage size	Range
Byte	1 byte	0 to 255
Boolean	2 bytes	**True** or **False**
Integer	2 bytes	-32,768 to 32,767
Long (long integer)	4 bytes	-2,147,483,648 to 2,147,483,647
Single (single-precision floating-point)	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double (double-precision floating-point)	8 bytes	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency (scaled integer)	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point; +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/-0.0000000000000000000000000001
Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any **Object** reference
String (variable-length)	10 bytes + string length	0 to approximately 2 billion
String (fixed-length)	Length of string	1 to approximately 65,400
Variant (with numbers)	16 bytes	Any numeric value up to the range of a **Double**
Variant (with characters)	22 bytes + string length	Same range as for variable-length **String**
User-defined (using **Type**)	Number required by elements	The range of each element is the same as the range of its data type.

Note Arrays of any data type require 20 bytes of memory plus 4 bytes for each array dimension plus the number of bytes occupied by the data itself. The memory occupied by the data can be calculated by multiplying the number of data elements by the size of each element. For example, the data in a single-dimension array consisting of 4 **Integer** data elements of 2 bytes each occupies 8 bytes. The 8 bytes required for the data plus the 24 bytes of overhead brings the total memory requirement for the array to 32 bytes.

A **Variant** containing an array requires 12 bytes more than the array alone.

Declaring Arrays

Arrays are declared the same way as other variables, using the **Dim**, **Static**, **Private**, or **Public** statements. The difference between scalar variables (those that aren't arrays) and array variables is that you generally must specify the size of the array. An array whose size is specified is a fixed-size array. An array whose size can be changed while a program is running is a dynamic array.

Whether an array is indexed from 0 or 1 depends on the setting of the **Option Base** statement. If **Option Base 1** is not specified, all array indexes begin at zero.

Declaring a Fixed Array

In the following line of code, a fixed-size array is declared as an **Integer** array having 11 rows and 11 columns:

```
Dim MyArray(10, 10) As Integer
```

The first argument represents the rows; the second argument represents the columns.

As with any other variable declaration, unless you specify a data type for the array, the data type of the elements in a declared array is **Variant**. Each numeric **Variant** element of the array uses 16 bytes. Each string **Variant** element uses 22 bytes. To write code that is as compact as possible, explicitly declare your arrays to be of a data type other than **Variant**. The following lines of code compare the size of several arrays:

```
' Integer array uses 22 bytes (11 elements * 2 bytes).
ReDim MyIntegerArray(10) As Integer

' Double-precision array uses 88 bytes (11 elements * 8 bytes).
ReDim MyDoubleArray(10) As Double

' Variant array uses at least 176 bytes (11 elements * 16 bytes).
ReDim MyVariantArray(10)
```

Declaring a Dynamic Array

By declaring a dynamic array, you can size the array while the code is running. Use a **Static**, **Dim**, **Private**, or **Public** statement to declare an array, leaving the parentheses empty, as shown in the following example.

```
Dim sngArray() As Single
```

Note You can use the **ReDim** statement to declare an array implicitly within a procedure. Be careful not to misspell the name of the array when you use the **ReDim** statement. Even if the **Option Explicit** statement is included in the module, a second array will be created.

In a procedure within the array's scope, use the **ReDim** statement to change the number of dimensions, to define the number of elements, and to define the upper and lower bounds for each dimension. You can use the **ReDim** statement to change the dynamic array as often as necessary. However, each time you do this, the existing values in the array are lost. Use **ReDim Preserve** to expand an array while preserving existing values in the array. For example, the following statement enlarges the array `varArray` by 10 elements without losing the current values of the original elements.

```
ReDim Preserve varArray(UBound(varArray) + 10)
```

Note When you use the **Preserve** keyword with a dynamic array, you can change only the upper bound of the last dimension, but you can't change the number of dimensions.

I distinguish between arrays with one dimension (vectors) and those with two dimensions (matrices).

So the following are fixed arrays: Dim avec%(10) or Dim Amat#(3,5)

There is a two-stage process for dynamic arrays: Dim Bmat() as Variant then on another line ReDim Bmat(nstep,nstep)

Using Select Case Statements

Use the **Select Case** statement as an alternative to using **ElseIf** in **If...Then...Else** statements when comparing one expression to several different values. While **If...Then...Else** statements can evaluate a different expression for each **ElseIf** statement, the **Select Case** statement evaluates an expression only once, at the top of the control structure.

In the following example, the **Select Case** statement evaluates the `performance` argument that is passed to the procedure. Note that each **Case** statement can contain more than one value, a range of values, or a combination of values and comparison operators. The optional **Case Else** statement runs if the **Select Case** statement doesn't match a value in any of the **Case** statements.

```
Function Bonus(performance, salary)
    Select Case performance
        Case 1
            Bonus = salary * 0.1
        Case 2, 3
            Bonus = salary * 0.09
        Case Is > 8
            Bonus = 100
        Case Else
            Bonus = 0
    End Select

End Function
```

Using If...Then...Else Statements

You can use the **If...Then...Else** statement to run a specific statement or a block of statements, depending on the value of a condition. **If...Then...Else** statements can be nested to as many levels as you need. However, for readability, you may want to use a **Select Case** statement rather than multiple levels of nested **If...Then...Else** statements.

Running Statements if a Condition is True

To run only one statement when a condition is **True**, use the single-line syntax of the **If...Then...Else** statement. The following example shows the single-line syntax, omitting the **Else** keyword:

```
Sub FixDate()
    myDate = #2/13/95#
    If myDate < Now Then myDate = Now
End Sub
```

To run more than one line of code, you must use the multiple-line syntax. This syntax includes the **End If** statement, as shown in the following example:

```
Sub AlertUser(value as Long)
    If value = 0 Then
        AlertLabel.ForeColor = "Red"
        AlertLabel.Font.Bold = True
    End If
End Sub
```

Running Certain Statements if a Condition is True and Running Others if It's False

Use an **If...Then...Else** statement to define two blocks of executable statements: one block runs if the condition is **True**, the other block runs if the condition is **False**.

```
Sub AlertUser(value as Long)
    If value = 0 Then
        AlertLabel.ForeColor = vbRed
        AlertLabel.Font.Bold = True
    Else
        AlertLabel.Forecolor = vbBlack
        AlertLabel.Font.Bold = False
    End If
End Sub
```

Testing a Second Condition if the First Condition is False

You can add **ElseIf** statements to an **If...Then...Else** statement to test a second condition if the first condition is **False**. For example, the following function procedure computes a bonus based on job classification. The statement following the **Else** statement runs if the conditions in all of the **If** and **ElseIf** statements are **False**.

```
Function Bonus(performance, salary)
    If performance = 1 Then
        Bonus = salary * 0.1
    ElseIf performance = 2 Then
        Bonus = salary * 0.09
    Else
        Bonus = 0
    End If
End Function
```

Using For...Next Statements

Nearly all my loops are For … Next

You can use **For...Next** statements to repeat a block of statements a specific number of times. **For** loops use a counter variable whose value is increased or decreased with each repetition of the loop.

The following procedure makes the computer beep 50 times. The **For** statement specifies the counter variable `x` and its start and end values. The **Next** statement increments the counter variable by 1.

```
Sub Beeps()
    For x = 1 To 50
        Beep
    Next x
End Sub
```

Using the **Step** keyword, you can increase or decrease the counter variable by the value you specify. In the following example, the counter variable `j` is incremented by 2 each time the loop repeats. When the loop is finished, `total` is the sum of 2, 4, 6, 8, and 10.

```
Sub TwosTotal()
    For j = 2 To 10 Step 2
```

```
        total = total + j
    Next j
    MsgBox "The total is " & total
End Sub
```

To decrease the counter variable, use a negative **Step** value. To decrease the counter variable, you must specify an end value that is less than the start value. In the following example, the counter variable `myNum` is decreased by 2 each time the loop repeats. When the loop is finished, `total` is the sum of 16, 14, 12, 10, 8, 6, 4, and 2.

```
Sub NewTotal()
    For myNum = 16 To 2 Step -2
        total = total + myNum
    Next myNum
    MsgBox "The total is " & total
End Sub
```

You can exit a **For...Next** statement before the counter reaches its end value by using the **Exit For** statement. For example, when an error occurs, use the **Exit For** statement in the **True** statement block of either an **If...Then...Else** statement or a **Select Case** statement that specifically checks for the error. If the error doesn't occur, then the **If…Then…Else** statement is **False**, and the loop will continue to run as expected.

Using Do...Loop Statements

I very rarely use Do … Loop

You can use **Do...Loop** statements to run a block of statements an indefinite number of times. The statements are repeated either while a condition is **True** or until a condition becomes **True**.

Repeating Statements While a Condition is True

There are two ways to use the **While** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop, or you can check it after the loop has run at least once.

In the following `ChkFirstWhile` procedure, you check the condition before you enter the loop. If `myNum` is set to 9 instead of 20, the statements inside the loop will never run. In the `ChkLastWhile` procedure, the statements inside the loop run only once before the condition becomes **False**.

```
Sub ChkFirstWhile()
    counter = 0
    myNum = 20
    Do While myNum > 10
        myNum = myNum - 1
        counter = counter + 1
    Loop
    MsgBox "The loop made " & counter & " repetitions."
End Sub

Sub ChkLastWhile()
    counter = 0
    myNum = 9
    Do
        myNum = myNum - 1
        counter = counter + 1
    Loop While myNum > 10
    MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Repeating Statements Until a Condition Becomes True

There are two ways to use the **Until** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop (as shown in the `ChkFirstUntil` procedure), or you can check it after the loop has run at least once (as shown in the `ChkLastUntil` procedure). Looping continues while the condition remains **False**.

```
Sub ChkFirstUntil()
    counter = 0
    myNum = 20
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
    Loop
    MsgBox "The loop made " & counter & " repetitions."
End Sub

Sub ChkLastUntil()
    counter = 0
    myNum = 1
    Do
        myNum = myNum + 1
        counter = counter + 1
    Loop Until myNum = 10
    MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Exiting a Do...Loop Statement from Inside the Loop

You can exit a **Do...Loop** using the **Exit Do** statement. For example, to exit an endless loop, use the **Exit Do** statement in the **True** statement block of either an **If...Then...Else** statement or a **Select Case** statement. If the condition is **False**, the loop will run as usual.

In the following example, `myNum` is assigned a value that creates an endless loop. The **If...Then...Else** statement checks for this condition, and then exits, preventing endless looping.

```
Sub ExitExample()
    counter = 0
    myNum = 9
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
        If myNum < 10 Then Exit Do
    Loop
    MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Use the Object Browser

The **Object Browser** allows you to browse through all available objects in your project and see their properties, methods and events. In addition, you can see the procedures and constants that are available from object libraries in your project.

To navigate the Object Browser

1. Activate a module.

2. From the **View** menu, choose **Object Browser** (F2). Select the name of the project or library you want to view in the **Project/Library** list.

3. Use the **Class** list to select the class; use the **Member** list to select specific members of your class or project.

4. View information about the class or member you selected in the **Details** section at the bottom of the window.

Set and Clear a Breakpoint

You set a breakpoint to suspend execution at a specific statement in a procedure; for example, where you suspect problems may exist. You clear breakpoints when you no longer need them to stop execution.
To set a breakpoint

1. Position the insertion point anywhere in a line of the procedure where you want execution to halt.

   On the **Debug** menu, click **Toggle Breakpoint** (F9), click next to the statement in the **Margin Indicator Bar**

If you set a breakpoint on a line that contains several statements separated by colons (**:**), the break always occurs at the first statement on the line.
To clear a breakpoint

1. Position the insertion point anywhere on a line of the procedure containing the breakpoint.

2. From the **Debug** menu, choose **Toggle Breakpoint** (F9), or click next to the statement in the **Margin Indicator Bar** (if visible.)

3. The breakpoint is cleared and highlighting is removed.

To clear all breakpoints in the application

- From the **Debug** menu, choose **Clear All Breakpoints** (CTRL+SHIFT+F9).

Trace Code Execution

- **Step Into**: Traces through each line of code and steps into procedures. This allows you to view the effect of each statement on variables.

- **Step Over**: Executes each procedure as if it were a single statement. Use this instead of **Step Into** to step across procedure calls rather than into the called procedure.

- **Step Out**: Executes all remaining code in a procedure as if it were a single statement, and exits to the next statement in the procedure that caused the procedure to be called initially.

To trace execution from the beginning of the program

- From the **Debug** menu, choose **Step Into** (F8), **Step Over** (SHIFT+F8), **Step Out** (CTRL+SHIFT+F8), or **Run To Cursor** (CTRL+F8).