VBA Variables

It is a good practice to declare your data types. In part it ensures that correct value types are used in your data. Also, it can improve the performance of your VBA applications.

The Variant type is the default type which is provided if a type declaration is omitted for a variable. The following code shows how in certain circumstances, such as those that involve repetitive tasks performed on large amounts of data, greatly decreased performance can result when variables are not declared.

Using the Variant type can also lead to type mismatch errors so it is advised in the general case to declare variables to appropriate types.

In the following example, run the code with declarations commented out and then again with the declarations in and compare the time of execution.

Test Performance With and Without Dims

Sub Performance( )

' comment variable declarations in and out and compare performance

' Dim x As Integer, y As Long
' Dim J As Double, K As Double, L As Double
' Dim v As Long, t As Long
' Dim Start, Finish As Date

' record start

    Start = Timer

' Perform arithmetic over many iterations in nested loops

    x = 0
    y = 0
For v = 1 To 10000
    x = x + 1
    y = x + 1
Next v

For t = 1 To 10000
    J = x + y + v
    K = y - x - v
    L = x / y * v
Next t

Finish = Timer

' Display total time in seconds
Debug.Print Format(Finish - Start, "0.0")
End Sub

Excel VBA Variable Types

Ref: ExcelFunctions.net http://www.excelfunctions.net/VBA-Variables-And-Constants.html

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>1 byte, an unsigned number, 0 to 255</td>
</tr>
<tr>
<td>Boolean</td>
<td>2 bytes, True or False. 1 or 0</td>
</tr>
<tr>
<td>Integer</td>
<td>2 bytes, signed, integral whole number, -32,768 to +32,767</td>
</tr>
<tr>
<td>Long</td>
<td>4 bytes, signed integral whole number, -2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>Single</td>
<td>4 bytes, signed, single precision floating point number, -3.4e38 to +3.4e38</td>
</tr>
<tr>
<td>Double</td>
<td>8 bytes, signed, double precision floating point number, -1.8e308 to +1.8e308</td>
</tr>
<tr>
<td>Currency</td>
<td>8 bytes, signed floating point number with fixed number decimal place -922,337,203,685,477.5808 to +922,337,203,685,477.5807</td>
</tr>
<tr>
<td>Date</td>
<td>8 bytes, Date &amp; Time - represented internally by a floating point number. The integer part of the number represents the date the decimal portion represents the time. 1st January 100 to 31st December 9999</td>
</tr>
<tr>
<td>Object</td>
<td>4 bytes A reference to an object</td>
</tr>
<tr>
<td>String</td>
<td>Fixed or variable length, fixed to 65K characters, variable to ~ 2 billion characters</td>
</tr>
<tr>
<td>Variant</td>
<td>Varies in that it can hold any type. Useful if type is not known.</td>
</tr>
</tbody>
</table>

The following code shows all the Variable types in code where the Variant type is used to define the array. Notice in this array definition the variables are passed into the array. The array in Variant form is powerful as it can store dissimilar variable types.
Macro Showing the VBA Variable Types

Sub Types()
Dim bite As Byte
Dim truefalse As Boolean
Dim inty As Integer
Dim longo As Long
Dim smallFloat As Single
Dim largeFloat As Double
Dim dollars As Currency
Dim today As Date
Dim text As String
Dim variantRay As Variant
Dim j As Integer
Dim rango As Range
Set rango = Range("A1:A10")

Dim obj As Object
Set obj = Application

bite = 255  'unsigned
truefalse = True
inty = 32767
longo = 2147483647
smallFloat = 12345678.1234568
largeFloat = 1.234567E+77
dollars = 1000000.0001
today = Now
text = "Excalibur"

' setting cell format of "A7" to currency and "A8" to time
Range("A7").NumberFormat = "$#,##0.00_);($#,##0.00)"
Range("A8").NumberFormat = "hh:mm"

variantRay = Array(bite, truefalse, inty, longo, smallFloat, largeFloat, dollars, today, obj, text)

j = 0
For Each cell In rango
    cell.Value = variantRay(j)
    j = j + 1
Next cell
End Sub

Once again we have used an array and now is a good time to learn more details about VBA arrays.
VBA Arrays

An array holds an indexed collection of elements. If the array is declared as type Variant it can hold different types of objects. The array can also be typed specifically to a particular variable type. An array typically is indexed with an 'offset' value starting at zero. Excel arrays have an added feature in that the index can be specified via a lower and upper range.

A Diagram of an VBA Array Indexed from Zero

<table>
<thead>
<tr>
<th></th>
<th>“cat”</th>
<th>“bird”</th>
<th>“fish”</th>
<th>“frog”</th>
<th>“fly”</th>
<th>“worm”</th>
<th>“spider”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Array Example in a Macro

Sub Animals()
Dim zoo As Variant
Range(“A1:G1”) = zoo

'Reference the individual elements in the array.
Range(“A3”).Value = zoo(0)
Range(“D3”).Value = zoo(3)
Range(“G3”).Value = zoo(6)
End Sub

Literal Form of Array // a special form that only works with the Variant Type

In the above macro, the array is declared in a literal form where each element in the array is specified at declaration. The variable zoo, that is receiving the reference to this collection is declared as type 'variant'.

Arrays Declared As Variant Type

When an array will hold different sorts of values, such as a combination of strings and numbers, the Variant type is appropriate for the array.

Example

Dim objectArray As Variant
Because the Variant type can represent any type even an array, no special form has to be provided.

**Literal Form Seems Only to Work With Variant Type**

It does not appear that the literal form of creating an array shown above can be used with other data types as is shown in the next example.

**Example**  // *not an adequate array declaration*

Dim stringArray as String

The above example doesn't work as an array variable, as the compiler doesn't 'see' this is as a possible array reference, rather it appears to be just a simple string variable declaration.

**Example Demonstrating the Above Doesn't Work**

Sub ArrayBug()
    Dim stringArray As String  'only variant type suffices in this form
    stringArray = Array("cat", "man", "do")
End Sub

// *doesn't compile → yields a type mismatch error*

The following variations does not fix it either, as the error arises, 'cannot assign to an array'.

**Doesn't Fix It**

Dim stringArray(3) As String

stringArray = Array("Kath", "Man", "Du")

// *cannot assign to an array.*

It seems this literal assignment, form is unique and must be declared using the Variant type.
Standard VBA Array Declaration

But good news! In the standard VBA array form, different types can be used. The values are just assigned to the array in a different way.

The standard form of an array declaration involves specifying the array with round braces and the number of elements that will be in the array.

Example

myArray(6) As String

Note in this style we are free to tailor the array to a specific type if appropriate. In the default array form, the classic offset indexing is used, here from 0 to 5.

Specifying Array Ranges

VBA though is very flexible and allows arrays to escape being indexed starting at 0.

VBA allows you to specify the array’s ranges so you can instead specify, for instance, a range of 1 to 6 which is more consistent with the VBA’s Range referencing system and perhaps can make your code easier to work with.

In the following example the array is declared to string type with an index from 11 to 13. Notice how the values are being assigned to each array element. The array then is easily assigned to a range.

Example  // varying the bounds of the array

Sub ArrayForms()

Dim stringRay(11 To 13) As String
stringRay(11) = "Tic"
stringRay(12) = "Tac"
stringRay(13) = "Toe"
Range("C4:E4") = stringRay
End Sub

If the values of the array will all be of one consistent type the array can be declared to that specific type such as Integer.
The Option Base Statement

An array can also be specified to start at an index value, using the Option Base statement. The statement has the following syntax with the default being 0.

Syntax
Option Base { 0 | 1}

The Option Base statement can only appear once in a module before any procedures, that is, before any sub. This is the first time we have a statement has been encountered that is outside the Sub).

Example

Option Base 1  ' placed outside module before array declarations

Sub OptionBase()
    Dim ray(3) As Integer
    For x = 1 To 3
        ray(x) = x
        Debug.Print x
    Next
    Range("A1:C1") = ray
End Sub

Array Row Column Orientation

There is a row column orientation we have encountered before when applying arrays to ranges. Arrays map to rows naturally. A special 'Transpose' property has to be applied to change the orientation of an array from vertical to horizontal or visa versa.

The following table provides details on the Transpose Method and also sheds some light on the inner working of the CSE function.

<table>
<thead>
<tr>
<th>WorksheetFunction.Transpose Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a vertical range of cells as a horizontal range, or vice versa. TRANSPOSE must be entered as an array formula in a range that has the same number of rows and columns, respectively, as an array has columns and rows. Use TRANSPOSE to shift the vertical and horizontal orientation of an array on a worksheet.</td>
</tr>
</tbody>
</table>
Multi-Dimensional Arrays

Arrays can also be multi-dimensional in VBA. The following declaration is a short form of a declaration of an array as a variant type. This array will create three arrays each which have three elements.

Example

Dim xyRay (1 to 3, 1 to 3)

Here is a short macro showing the standard referencing into a two dimensional array. For a long list you would device means to eliminate the redundancy however this is adequate to demonstrate the use of two dimensional arrays.

Example

Sub TwoDRay()

Dim xyRay(1 To 5, 1 To 3)  ' rows & columns

xyRay(1, 1) = "Name"
xyRay(1, 2) = "Grade"
xyRay(1, 3) = "ID"

xyRay(2, 1) = "Roy Rogers"
xyRay(2, 2) = 11
xyRay(2, 3) = "12357"

xyRay(3, 1) = "Dale Evans"
xyRay(3, 2) = 10
xyRay(3, 3) = "11131"

xyRay(4, 1) = "Gabby Hayes"
xyRay(4, 2) = 4
xyRay(4, 3) = "71923"

xyRay(5, 1) = "Trigger"
xyRay(5, 2) = 0
xyRay(5, 3) = "54321"

Range("A1:C1").HorizontalAlignment = xlCenter
Range("A1:C5") = xyRay

End Sub
**Ubound( array )**

Ubound can be used on a Array type to find the upper bound of an array. In this classic array that value will be the offset value 2.

```vba
Sub ub( )
    Dim ary As Variant
    ary = Array(1.1, 2.2, 3.3)
    For j = 0 To UBound(ary)
        Debug.Print ary(j)
    Next j
End Sub
```

**Array To Range**

An array can be dropped into a range very easily. In the next code sample you see array cars is simply assigned to an appropriately size range object.

**Example**

```vba
Range("A1:E1") = Cars
```

**Range To Array**

To load a classic array from a range, we can loop through the values of the range and put the values one by one into the array.

Note the following code is a little confusing as we build an array using the literal form we first discussed and then assign it to a range. Then we pick up the values from the range in a loop and put them into a second array CarNames, loading each element of that array one at a time. Finally we put values from the second array into selected ranges.

**Array To Range and Range To Array Example**

```vba
Sub A2R_n_R2A()
    Dim CarNames(5) As String
    Dim Cars As Variant
    Dim j As Integer
```
the first array is built using the literal form of the array

Cars = Array("Ford", "GM", "Chrysler", "Toyota", "Honda")

array loads a range

Range("A1:E1") = Cars

filling a second array called CarNames, this time a string array, with values from the range

j = 0

For Each cell In Range("A1:E1")
    CarNames( j ) = cell.Value
    j = j + 1
Next cell

taking values from the standard array and putting them into range locations

Range("A3:E3") = CarNames
Range("A5") = "North American Car Makers"
Range("A6") = CarNames(0)
Range("B6") = CarNames(1)
Range("C6") = CarNames(2)

End Sub

Simple Approach To Putting An Range into an Array

The question becomes is there a simpler way to get range values into an array. Here is an example from online. It sure looks simpler!

Example


Sub QuickFillMax()
    Dim myArray As Variant
    myArray = Worksheets("Sheet1").range("B2:C17")
    MsgBox "Maximum Integer is: " & WorksheetFunction.Max(myArray)
End Sub

Dynamic Arrays

The arrays we have created so far are fixed in size. Array may also be created that vary in size and can grow as is required.

A dynamic array is declared with the size not stipulated.
Example

```
Dim myArray() As String ' no size in declaration, will be set later with Redim
```

Redim

The term Redim, (meaning re-dimension), is then used to set the size of the array. Redim allows you to set the size at a later time in the program when pertinent information is available.

The following code from Mr. Excel’s book shows the dynamic array, deferring setting the size of the array to a later point in the macro, sometimes after the declaration. The following is modified to run on the current workbook. See the original in the Resource downloads.

Example // modified from VBA & Macros' Microsoft Excel 2010, Bill Jelen & Tracy Syrstad

```
Sub MySheets()
    Dim myArray() As String
    Dim myCount As Integer, NumShts As Integer
    NumShts = ActiveWorkbook.Worksheets.Count
    ReDim myArray(1 To NumShts)
    For myCount = 1 To NumShts
        myArray(myCount) = ActiveWorkbook.Sheets(myCount).Name
        Debug.Print (myArray(myCount)) ' added Debug statement
    Next myCount
End Sub
```

// The Arrays chapter in the text has some additional information on arrays.

Error Handling in VBA

If an error is encountered in VBA and there is no error handling code the ‘Continue, End, Debug, Help’ error window pops up from where you are invited to enter Debug mode. The offending line that caused the error will be highlighted in yellow. (Hovering the cursor over a variable, in break mode, will show you it’s current value.)
Error Handling

Error handling code enables a programmer to branch their program to a block of code that handles the error condition should it arise and the permit the program to proceed normally.

The following summarizes handling components that are built into VBA and the code that is used to deal with errors.

Error Handling Components

- On Error . . . GoTo Label
  - used to enclose a block of code that might yield an error

- Exit Sub
  - prevents Error Handling Code to Automatically Execute

- The Label
  - used after after Exit
  - a name followed by a colon
  - labels the error handling branch code

- Error handling code
  - this is the code that decides what to do with an error condition

- Add 'Resume Next'
  - to return control to the next line of the macro

The following code creates a classic error condition and handles it permitting the macro to complete rather than to exit to Debug mode. The key features of the error handling code are highlighted.

Example

Sub Error()
  On Error GoTo ZeroDiv  ' begins error protected scope
  x = 1 / 0  "div by 0"
  Debug.Print ("OK, Won't do that again!")
  Exit Sub  ' Don't do error handling code if no error occurs
  ZeroDiv:  ' error handling block marked by label
  Answer = MsgBox("Divide By Zero")  ' pressing the OK button makes vbOK true
  If vbOK Then
    Resume Next  ' return to macro after line that threw error
  End If
End Sub
The following example adds an error. We kept the earlier error handling code in it to show more than one error type being handled.

Example

Sub Error2()
    On Error GoTo ZeroDiv
    x = 1 / 0 'div by 0
    Debug.Print("OK, Won't do that again!")

    On Error GoTo NoFileFound
    Open "Notta.txt" For Input As #1 'no file found
    Debug.Print("No Notta File!")
    Exit Sub

ZeroDiv:
    Answer = MsgBox("Divide By Zero")
    If vbOK Then
        Resume Next
    End If

NoFileFound:
    Answer = MsgBox("File Not Found")
    If vbOK Then
        Resume Next
    End If
End Sub

Built In Error Handling Using Err Description and Number

VBA has a built in Err object that has both a number and a description property. These are described in some detail in the box below. These built in error types can be utilized to handle errors to avoid going to the Debug dialog box. Notice in the following example we report the error back into the spreadsheet. Notice also the use of the built in VB constant vbOK which can be used to process the condition where the OK button is pressed.

Example Using Err.Description and Err.Number

Sub Error3()
    On Error GoTo NoFileFound
    Open "Notta.txt" For Input As #1
    Debug.Print("No Notta File!")
    Exit Sub

NoFileFound:
    Answer = MsgBox(Err.Description & ": Error Number " & Err.Number)
' reporting errors into spreadsheet

Range("A5").Value = Err.Description & ": Error Number " & Err.Number
If vbOK Then
    Resume Next
End If
End Sub


Frequently when a call on a function of sub doesn't work out one of the values highlighted below shows up in a spreadsheet. Pearson consulting supplies the following.

"VBA provides a function called CVErr that takes a numeric input parameter specifying the error and returns a real error value that Excel will recognize as an error. The values of the input parameter to CVErr are in the XLCVError Enum and are as follows:

- xlErrDiv0 (= 2007) returns a #DIV/0! error.
- xlErrNA (= 2042) returns a #N/A error.
- xlErrName (= 2029) returns a #NAME? error.
- xlErrNum (= 2036) returns a #NUM! error.
- xlErrRef (= 2023) returns a #REF! error.
- xlErrValue (= 2015) returns a #VALUE! Error."

// the above are the seven built in VBA error types

- On Error GoTo 0 → cancel Error trapping
- On Error Resume Next → on error just keep going
- Err.Clear → reset error to clear

Ignoring Errors

In the Box above we see 'Resume Next' which we have already used in code, but also we see 'Go To 0' Mr Excel has an example on page 554 where the Kill command is called on a file. This could result in an error if the file does not exist. In this circumstance where the file is being replaced anyway the error can be ignored. Hence the 'Go To 0' directive is used.

Example

Go To 0  // used to ignore an error should it arise
Example Macro That Ignores Errors

Sub Apathy()
    On Error Resume Next
    x = 1 / 0
    On Error GoTo 0
    Debug.Print "The world ended and Macro doesn't care!"
End Sub

There are more details to be learned on errors on pages 555 to 562 in the 2010 text and pages 525 to 531 in the 2013 text which are left to the reader to investigate.

Assignment

This assignments aims to provide sample artifacts for each of the topics covered in this note. Work from a clean sheet or use the examples in the notes (or both) to come up with your responses.

Types

1 ) Create a macro that outputs each VBA type in the first column of a spreadsheet, an example of the type in the second column and a description of the type in the third column. (you can use the description for the types supplied in the table at the beginning of this note or some facsimile.) Capture the output in a screenshot and the macro that was used to create the output.

The next question asks you to do essentially the same thing that is done in the Sub A2R_n_R2A() sub where an array is created using literal form. It's values are then passed into a range. A second array is created using standard form. The values of the newly created range are passed into this second array. Finally the second arrays values are put into a second range.

2 ) Create an literal array with a a set of any seven, eight or nine items that are related such as products from a store. ( i.e. VCR, DVD player, TV etc. ) (This will entail using the Variant type and array literal notation, i.e. Array(1,2,3).

Transfer six of these values to the range A1:F1 by assignment or increase the range size to take all items in your array.

We are now going to transfer this range into a second array. Create a loop to do this. Prove the capture worked by assigning the new array to a new range in a new row.
Finally access the second array selectively using array notation, a selection of the values of this array and show them in a new range of a spreadsheet.

**Error Handling**

3 ) Create three short macros. The first is called DoNothingOnError. Simply assign to a variable a divide by zero.

Record code and the Debug message in a screenshot.

Second create a macro called HandlingError. In this one include all the classic features of code protected by VBAs handling system. Highlight the key handling features. Again do from scratch or use code provided as a model.

Finally, create a macro called IgnoringError and use the error handling form that utilizes 'GoTo 0' to ignore an error in a macro.

Provide code and screenshot outputs.

---

**Last Weeks Anagram**

**Prove the Anagram**

Sub Anagram_C3_Active( )

' Before running have cell cursor at C3 where the "START" cell will be copied in
' Also you must be in R1C1 style referencing in the Spreadsheet
' File → Options → Formulas → check R1C1

Dim ray(0 To 27)
Dim rango As Range
Dim s As String
Dim tv As String

'convert numbers to letters in loops
'add a few special values, START and XYZ

For j = 0 To 11
ray(j) = Chr(j + 65)
Next j

ray(12) = "START"

For k = 12 To 23
ray(k + 1) = Chr(k + 65)
Next k

ray(24) = "XYZ"
i = 0
For w = 1 To 5
    For z = 1 To 5
        Cells(w, z).Value = ray(i)
        i = i + 1
    Next z
Next w


For Each cell In rango
    tv = Trim(cell.Value)
    s = s & tv
Next cell

Range("A7").Value = "Answer"
Range("B7").Value = s

End Sub

Sub EAF()
FinalRow = Cells(Rows.Count, 1).End(xlUp).Row
Cells(FinalRow + 1, 3).FormulaArray = "=SUM(R2C[-2]:R[-1]C[-2]*R2C[-1]:R[-1]C[-1]*R2C:R[-1]C)"
End Sub