Session 14

Using Macros II

Function Macros for Arrays in Visual Basic for Applications

Last revised: July 6, 2011

Review of last time: Using Macros I

- Macros reduce maintenance costs, reduce errors, and speed development
- Two kinds of macros: function macros and command macros
- Two languages — VBA and XLM
- Basic VBA macro structure
  - Variable declarations
  - Computations
  - Returning values
- Objects have properties and methods
- Methods and properties use postfix syntax
  - Caller, Column, Row, Columns, Rows, Count,
  - Application object

Array function macros

- Last time we discussed scalar function macros — macros that return a single value
- Array function macros return a rectangular array of values
- Array function macros are powerful
  - Manage blocks of data at high levels
  - Support array computations
  - Reduce maintenance costs and development time
- But they require more machinery
  - Iteration
  - Understanding Excel's object model
  - Dynamic allocation
- They're worth it
- Let's start with a few simple examples

Example 1: Add two 3x3 arrays

- Returns the sum of two 3x3 ranges

```
Function ArraySum(range1 As Range, range2 As Range) As Variant
    Dim i As Integer, j As Integer   'iteration variables
    Dim answerArray(3, 3)

    For i = 1 to 3
        For j = 1 to 3
            answerArray(i, j) = range1(i, j) + range2(i, j)
        Next j
    Next i
    ArraySum = answerArray
End Function
```

- This works, but it's unnecessary: Excel can already do this
- Let's see how we built this

ArraySum: How to build it

- Create a module
- Insert module options
- Write a function macro in the module.

  Start with function statement, which includes the function name and its argument list

```
Function ArraySum(range1 As Range, range2 As Range) As Variant
    <Variable declarations>
    <Computations>
End Function
```

- As usual, the body of the function definition contains two parts:
  - Variable declarations
  - Computations

Since Variant is the default data type, this is optional

ArraySum: Create the answer array

- Use Dim statement to declare the answer array

```
Function ArraySum(range1 As Range, range2 As Range) As Variant
    Dim answerArray(3, 3)
    <Other variable declarations>
    <Computations>
    ArraySum = answerArray
End Function
```

- Use an assignment statement to return the value

How to use iteration

- Iteration is a repeat form
- Many varieties of iteration in VBA; we use a simple one:

```
For <var> = <min> To <max>
        <sequence-of-statements>
Next <var>
```

- Iteration can be nested; that's how we handle two-dimensional arrays

```
For <varx> = <minx> To <maxx>
        For <vary> = <miny> To <maxy>
                <sequence-of-statements>
        Next <vary>
Next <varx>
```

- You must include variable declarations for iteration variables

Now iterate through the argument array

- Notice that declarations are needed for the iteration variables

```
Function ArraySum(range1 As Range, range2 As Range) As Variant
    Dim answerArray(3, 3)
    Dim i As Integer, j As Integer
    For i = 1 To 3
        For j = 1 To 3
            answerArray(i, j) = range1.Cells(i, j) + range2.Cells(i, j)
        Next j
    Next i
    ArraySum = answerArray
End Function
```

Example 2: Multiply an array by a constant

- The macro takes two arguments:
  - array of any size
  - constant
- It returns an array equal to array * constant
- Plan of calculation:
  - Create an array to hold the answer
  - Iterate through the argument array:
    - Pick up the array element
    - Multiply it by the constant
    - Insert the result into the corresponding place in the answer array
- This one is trickier because we don't know the size of the array
- This is really useless, because Excel can do this without a macro

Now build a macro to do this

- Create a module as before (if needed)
- Write a function macro in the module.

  Start with function statement, which includes the function name and its argument list

```
Function Multiply(argRange As Range, factor As Double) As Variant
        <Variable declarations>
        <Computations>
End Function
```

- As usual, the body of the function definition contains two parts:
  - Variable declarations
  - Computations

Since Variant is the default data type, this is optional

Dynamic arrays

- We can't always know the size of an array when we're writing the program—it can depend on the runtime environment.
- We need the ability to allocate space for it on the fly: use ReDim
- ReDim declares the array's size at execution time
- To declare a two dimensional array 5x10:

```
ReDim answerArray(5,10)
```

- Sometimes the size depends on the sizes of arguments:

```
inputRows = argRange.Rows.Count
inputColumns = argRange.Columns.Count
ReDim answerArray(inputRows,inputColumns)
```

Create the answer array

- Steps:
  - At compile time:
    - Declare the array
  - At run time:
    - Extract the sizes of the argument array
    - Resize the answer array

```
Function Multiply(argRange As Range, factor As Double) As Variant
    Dim inputRows As Integer, inputColumns As Integer
    Dim answerArray
    inputRows = argRange.Rows.Count
    inputColumns = argRange.Columns.Count
    ReDim answerArray(inputRows, inputColumns)
    <Computations>
End Function
```

Now iterate through the argument array

Notice that declarations are needed for the iteration variables

```
Function Multiply(argRange As Range, factor As Double) As Variant
    Dim i As Integer, j As Integer
    Dim inputRows As Integer, inputColumns As Integer
    Dim answerArray
    inputRows = argRange.Rows.Count
    inputColumns = argRange.Columns.Count
    ReDim answerArray(inputRows, inputColumns)

    For i = 1 To inputRows
        For j = 1 To inputColumns
            answerArray(i, j) = argRange.Cells(i, j) * factor
        Next j
    Next i

End Function
```

Finally, return the value


```
Function Multiply(argRange As Range, factor As Double) As Variant
    Dim i As Integer, j As Integer
    Dim inputRows As Integer, inputColumns As Integer
    Dim answerArray
    inputRows = argRange.Rows.Count
    inputColumns = argRange.Columns.Count
    ReDim answerArray(inputRows, inputColumns)

    For i = 1 To inputRows
        For j = 1 To inputColumns
            answerArray(i, j) = argRange.Cells(i, j) * factor
        Next j
    Next i
    Multiply = answerArray
End Function
```

More useful example

- You're writing a plan for producing a product line of hose couplings.
- You must consider materials requirements and sales projections.
- The materials requirements are given as an array of Coupling-Type by Diameter [name: BrassContent]
- Expected shipments are given as an array of Coupling-Type-and-Size by Month [name: ProjectedSales]
- The dimensions of these tables are incompatible.
- We need to "unwind" the BrassContent range.

Hose coupling data

Brass Content (kgm)	4 Inch	6 Inch	8 Inch
T-Joint	2.5	3.2	5.1
L-Joint	3.6	4.3	6.2
Y-Joint	4.3	5.4	8.7

Projected Shipments	*Oct*	*Nov*	*Dec*	*Jan*	*Feb*	*Mar*
T-Joint 4 Inch	515	170	436	419	528	357
L-Joint 4 Inch	543	205	570	276	140	395
Y-Joint 4 Inch	371	369	133	489	566	215
T-Joint 6 Inch	572	149	138	314	506	592
L-Joint 6 Inch	229	579	413	295	228	568
Y-Joint 6 Inch	531	166	594	458	416	218
T-Joint 8 Inch	217	236	311	291	566	404
L-Joint 8 Inch	462	511	306	259	458	574
Y-Joint 8 Inch	273	167	374	544	405	257

Unwinding the brass content data

T-Joint 4 Inch	2.5
L-Joint 4 Inch	3.6
Y-Joint 4 Inch	4.25
T-Joint 6 Inch	3.2
L-Joint 6 Inch	4.3
Y-Joint 6 Inch	5.44
T-Joint 8 Inch	5.1
L-Joint 8 Inch	6.2
Y-Joint 8 Inch	8.67

T-Joint 4 Inch	=INDEX(BrassContent,0,1)
L-Joint 4 Inch	=INDEX(BrassContent,0,1)
Y-Joint 4 Inch	=INDEX(BrassContent,0,1)
T-Joint 6 Inch	=INDEX(BrassContent,0,2)
L-Joint 6 Inch	=INDEX(BrassContent,0,2)
Y-Joint 6 Inch	=INDEX(BrassContent,0,2)
T-Joint 8 Inch	=INDEX(BrassContent,0,3)
L-Joint 8 Inch	=INDEX(BrassContent,0,3)
Y-Joint 8 Inch	=INDEX(BrassContent,0,3)

Now build a macro to do this

- Create a module as before (if needed)
- Insert module options
- Write a function macro in the module.

  Start with function statement, which includes the function name and its argument list

```
Function VLineUp(argRange As Range) As Variant
        <Variable declarations>
        <Computations>
End Function
```

- As usual, the body of the function definition contains two parts:
  - Variable declarations
  - Computations

Since Variant is the default data type, this is optional

Overall approach

- Receive the BrassContent array as an argument (3x3)
- Assemble a 9x1 array by stacking the columns of BrassContent
- Return the 9x1 array
- We will iterate through the cells of BrassContent, inserting its values into the result

Planning the iteration

- Where will the (i, j) element of the input end up?
- (j-1) * 3 + i

```
Function VLineUp(argRange As Range) as Variant
    Dim answerArray(9), i As Integer, j As Integer

    For j = 1 To 3
        For i = 1 To 3
            answerArray((j - 1) * 3 + i) = argRange.Cells(i, j)
        Next i
    Next j
    VLineUp = answerArray
End Function
```

- This isn't quite it, though: it's a 1x9 array.
- We have to transpose it

Finally


```
Function VLineUp(argRange As Range) as Variant
    Dim answerArray, i As Integer, j As Integer

    For j = 1 To 3
        For i = 1 To 3
            answerArray((j - 1) * 3 + i) = argRange.Cells(i, j)
        Next i
    Next j
    VLineUp = Application.WorksheetFunction.Transpose(answerArray)
End Function
```

Limitations of VLineUp

- It assumes that the input is 3x3
- It can produce only a 1x9
- In real problems, it's much more useful if you can avoid assumptions about array sizes
  - Things might change
  - You might want to use the macro for another project
- A more useful unwinder wouldn't assume 3x3
- It would unwind any square range into a single column
- How can we do that? Use dynamic arrays

Dynamic unwinder


```
Function VLineUp(argRange As Range) As Variant
    Dim answerArray, i As Integer, j As Integer
    Dim inputColumns As Integer, inputRows As Integer

    inputRows = argRange.Rows.Count
    inputColumns = argRange.Columns.Count
    ReDim answerArray(inputRows*inputColumns)
    For j = 1 To inputColumns
        For i = 1 To inputRows
            answerArray((j - 1) * inputRows + i) = argRange Cells(i, j).Value
        Next i
    Next j
    VLineUp = answerArray
End Function
```

Resize and Offset

- The functionality of the worksheet function OFFSET is available in VBA in a slightly different form
- Offset is a Range method that works like the worksheet function OFFSET, but it takes only the first two arguments.

```
theRange.Offset(i, j)
```

- Resize is a Range method that returns a range with a different shape and size, but 0,0 offset.

```
theRange.Resize(i, j)
```

- You can chain them together:

```
theRange.Offset(3,2).Resize(1,4)
```

A tip about arrays

- Excel worksheet functions (mostly) work for ranges as well as individual cells.
- When you use them inside a VBA macro, you don't need to iterate (see previous slide).
- If you assign their return value to the function name, you're done

The main points

- Iteration
- Dynamic arrays
- Using the Set statement for objects

Reference readings

- Rob Bovey, Stephen Bullen, John Green, Robert Rosenberg, *Excel 2002 VBA Programmers Reference*. Birmingham, UK: 2001. Wrox Limited.
  - This is a whole lot more than you need for this course. Don't even think of looking at this unless you want to dive into programming. But if you want to, it's a solid reference.
- On line help for VB takes some getting used to, but it is serviceable.
- Readings: Excel Macros in Visual Basic for Applications

Preview of next time: Spreadsheet Tools for Managers

- Rough out your command macros using the macro recorder
- Many commands can't be recorded
- Separate the presentation function from the maintenance function
- Using the Set statement for objects
- Use templates to collect data from the organization
- Avoiding putting macros in templates
- Distribute macro collections as add-ins