VBA Programming

Visual Basic is an object-oriented programming language. A limited (but still very powerful) version of Visual Basic, called VBA, is incorporated into Word, Excel, Access, and other Microsoft Office application programs. VBA is the engine needed to create macros. In some cases, the application program does an excellent job of writing the VBA code for you. When you create an Access form that searches for data, an underlying piece of VBA code is written that makes the form operate. Most people never even see the code - they don't need to. In other situations, however, you will need to read and understand the code in order to make it do exactly what you want. In some cases it may be more convenient to write Excel macros directly in VBA.

This section provides an introduction to programming in Visual Basic for Applications (VBA) programming language. It covers several features and keywords used in VBA.

The material in this document is based on Ron William's material for the Tech 394 course at Minnesota State University.

VBA Programming Table of Contents

Programming Concepts

Objects

VBA is an "object-oriented language". This means that it identifies items on the screen as objects, and then takes action to modify those objects as instructed. The objects can be an entire form, a box or line on the form, a worksheet, or a cell in a worksheet. Each object can have related properties. The kinds of properties associated depend on the object identified. A text box on a form will have a property called "Text," which represents the text to be printed in the box. An input box also allows the user to show text on the screen, but the text will appear in a property called "Contents".

A typical line of VBA code might read:

```
ActiveCell.Offset(2,3).Value = "Hello"
```

The object in this line is "ActiveCell", which identifies the cell where the cellpointer (cursor) rests. If it were in cell B2, you could also identify it as Range("B2"). Then you could get the same results with:

```
Range("B2").Offset(2,3).Value = "Hello"
```

Note that if the cellpointer moves to a new location, the results would no longer be the same.

Variables

In many situations, you want to be able to refer to a string of text or a number by a name that can be called again and again. VBA allows you to store the information in memory in a variable. VBA allows you to identify these variables by any name that is not a reserved word (naming a variable "Exit" would be a problem, because VBA understands that word to be an instruction).

The values in variables will go away when the program terminates. If you want to save the value for future reference, you must tell VBA to print it, store it in a cell on the spreadsheet, etc. If you don't, when the macro is complete, the data is lost.

A typical piece of VBA code using variables might be:

```
Dim vNumber
vNumber = 0.23
Range("D6").Value = Range("B6").Value*vNumber
```

These instructions would tell VBA to look up the value in cell B6, multiply it by the value stored in a variable named vNumber, and write the result in cell D6. You could have written the number directly in the last line (rather than using a variable), but if that same number needs to be used in several places, you could use the same variable name in any other line of code, and VBA will always substitute the same value.

Variable Types

In the above example, we did not specify a variable type when we used the Dim (dimension) statement to declare the vNumber variable. If a type has not been specified, the variable is considered to be **variant** type. This means that the variable will take on the type of the first thing that is stored in it (in the above example, a floating point number).

There are many variable types in VBA, but here is a list of those that you are most likely to use in CSCA01. Notice that we start the name of the variable with a lowercase letter that indicates the type (this is what we call a naming convention – it is not required, but it is a good practice).

Integer

```
Dim iNum As Integer
iNum = 55
Range(“A1”).Value = iNum
```

Floating-Point (decimal)

```
Dim dNum As Double
dNum = 15.47
Range(“A1”).Value = dNum
```

String (text)

```
Dim sText As String
sText = “Hi Mom!”
Range(“A1”).Value = sText
```

Methods

VBA also allows you to run pre-defined processes that modify the data shown. Since the word "procedure" is used in another way, VBA identifies these as "methods". A typical method in spreadsheet use is "Offset", which allows the user to identify a cell at some location relative to the object cell.

A period (dot) follows the object, on which the **Offset** method is to be run (in this case, the active cell).

ActiveCell.Offset(2,3)

The above method means, "from the referenced cell, move down 2 rows, and over 3 columns". The method will only work with spreadsheet cell objects, since it needs to be able to move through columns and rows. You can get fancy with this by replacing the number 2 or 3 with statements that yield a number. For instance, if the number 4 was stored in cell B6, the method statement:

ActiveCell.Offset(2, Range("B6").Value)

would offset from the referenced cell by 2 rows and 4 columns, since it would find the number 4 in cell B6.

Properties

Following the object or method is the property. In the example above, the property identified is "Value". It means the value of whatever is stored in the cell. A similar property is "Contents". You must be careful how you store things in cells, and how you identify them. If you somehow manage to store the number 1 as a text string, VBA will identify it as a character, just like any other letter. Its numeric value will be "0", since you can't add a letter to a number. We'll spend some time looking at functions that help us keep these straight.

The last part of the statement is the action to be taken. Many statements list a property followed by an equals sign, and then a number or text, in this case, '= "Hello"'. These statements tell VBA to replace whatever is currently in that property for the object with whatever follows the equals sign. Therefore

ActiveCell.Offset(2,3).Value = "Hello"

says to replace whatever value may currently be stored in the cell 2 rows down and 3 columns to the right of the active cell with the text string "Hello". Likewise,

ActiveCell.FontName = "MS San Serif"

would change the font used for text in the active cell, from whatever was currently used to MS San Serif.

Functions

Functions are built-in processors that let us modify data in a specific way. The function **Ucase( )** transforms whatever string is in the parentheses so that it has all upper case characters. Functions can appear in the middle of a VBA code line. For instance

```
Range("D3").Value = Ucase(Range("B23").Value)
```

determines the value in cell B23, converts it to upper case, and then places it in cell D3.

Functions in VBA may be somewhat confusing since Excel has built-in functions as well. If the function

```
Upper(+B23)
```

was placed in cell D3, it would do the same thing. If the value in cell B23 was to change, Excel would update cell D3 automatically. The VBA statement would only replace the value in cell D3 once. When planning a problem solution, you often need to consider whether you will let the application (in this case, Excel) do the work, or whether you will do the work in VBA. Keep in mind, there are some things you can only do in VBA.

In addition to using the functions that are built-in within VBA, it is also possible to create your own user-defined custom functions. Custom functions will not be discussed in this document.

Subroutines

VBA processes a set of statements that are grouped together in what it call a "subroutine." Each subroutine starts with the word "Sub," followed by a name and then the open and closed parentheses. It ends with the statement "End Sub". VBA will try to process the subroutine from top to bottom, reading each line as a statement to be processed.

- Any line that starts with a singe quote (‘) will be highlighted in green as a comment - VBA will not attempt to process anything on that line.
- Also, if any statement is too long to fit conveniently on one line, it can be broken into two or more lines, with each line ending in an underscore (_). The underscore cannot occur within a text string, because VBA will try to make it a part of the string.

A typical VBA program (or subroutine) might look like this:

```
Sub MyMacro( )
```

```
        Range("B3").Activate
        Range("B3:B18").Select
        Selection.Copy
        Range("E3").Activate
        ActiveSheet.Paste

    End Sub
```

This subroutine moves the cellpointer to cell B3, select the range from B3 to B18, copies that selection to the clipboard, and pastes it into a range starting at cell E3. This is the same effect as if you had used the mouse to highlight the range, used Ctrl-C to copy, used the cellpointer to select E3, and pressed Ctrl-V to paste.

You can call one macro from inside another, simply by inserting the macro name. For example, if you wrote:

```
    Sub NewMacro( )

        Range("D6").Activate
        ActiveCell.Value = Now
        MyMacro

    End Sub
```

The macro would move the cell pointer to cell D6 and enter the date ("Now" is a VBA function that looks at the computers internal calendar, and return's the current date). It would then run the macro named "MyMacro", as recorded above. When MyMacro was done running, VBA would return to NewMacro, read the next line - End Sub - and quit.

Getting Started

The easiest way to write a VBA program is by recording a macro of the key stokes used to accomplish the task. To do this, use **Tools→Macro→Record New Macro…**, choose a name for the new subroutine, and OK to begin recording. When the steps you want are completed, stop the recording. You can view the VBA code by selecting **Tools→Macro→Macros**, highlight the macro you recorded, and press **Edit**. You can go directly to the VBA editor by pressing Alt-F11.

Next, try to determine how to make the code do what you want, rather than what it recorded. For example, if you wanted to know how to get VBA to input data to a cell, you might turn on the recorder, enter a cell, type in a message, press return, and turn off the recorder. The code would look like:

```
Range("B3").Activate
ActiveCell.FormulaR1C1 = "A message"
Range("B4").Activate
```

If you were unclear as to what the code meant, you could use VBA's context-sensitive help. If you recognized that FormulaR1C1 was a property, but weren't sure how it worked, you'd put the cursor anywhere on that word, and press **F1**. It would bring up the help file for that keyword, including examples.

Input Boxes

If your purpose is to offer the user a chance to enter some data into a cell, rather than inserting a set message each time, you would replace the text string "A message" (in the example above) with a call to the InputBox function. This function pops up a Windows dialog box with a sentence prompting the user to type in some information. When the user presses **Enter**, the function returns whatever the user typed as a string. The standard form is:

```
InputBox(prompt [, title] [, default])
```

Notice that the optional parameters are indicated with square brackets around them. The **prompt**, **title**, and **default** arguments must be text strings, so they would have double quotes around them. If you leave off the double quotes, VBA will look in memory to find a variable with the name you typed, and an error would be displayed since it would not be found.

The **prompt** is the message that appears in the box, asking for information. The **title** appears in the upper left-hand corner of the box. The **default** value is displayed in the text box as the default response if no other input is provided (the text box is displayed empty if this parameter is omitted). If you were asking for a user's home province, and you knew that 90% of the users were from Ontario, you could enter **"Ontario"** as the **default** value.

Message Boxes

You may also pop up a dialog box that simply displays information. The MsgBox function allows you to stop the macro, deliver a message, and let the user respond by clicking on a button. In its simplest form, it appears as a single line:

```
MsgBox message
```

This will simply stop operation and pop up a standard dialog box containing 2 things - the **message**, and a button to click that says **OK**. A message box can be used to tell the user when an error has occurred. After clicking OK, the program could then branch to a section of code that deals with the error.

In its more complex form, MsgBox is a function that returns a value. The value returned will correspond to whichever button the user clicked (you can have other buttons besides just the OK button). The value returned is actually a number, but VBA has several built-in constants (a constant is the same as a variable, but its contents cannot change), with easy-to-remember names, for the possible numbers that might be returned. For example, if the user clicks a button marked **Yes**, MsgBox will return the number **6**. You don't need to remember that number because the vbYes constant (which is much easier to remember than 6) stores this number. Likewise, there are other built-in constants (vbNo, vbOK, vbCancel, etc) that store the numbers that correspond to the other buttons that the user might click.

The standard form for this use of MsgBox is:

```
MsgBox(prompt [, buttons] [, title])
```

As with InputBox, the **prompt** and **title** are text strings. The **prompt** is the message that is printed on the dialog box, and the **title** is printed at the top of the window. **Buttons** is a number corresponding to which buttons (yes, no, cancel, OK) and what icons you want to appear in the box. The best way to decide what value to enter is to look for help on the MsgBox key word. The help file contains several good examples. The output from this form of MsgBox can be used to control the branching of code.

Note that when this form of MsgBox is used, VB interprets it as a function which returns a value (notice the parentheses). You can't simply set it in a line of code by itself - VB wants to know what you want to do with the value it returns. A typical use is to ask the user a yes/no question, and have the code in the macro respond based on the answer. This requires the use of an If statement:

```
If MsgBox("Do you want to quit now?", vbYesNo, "Quit") = vbYes Then
    Quit
End If
```

In the statements above, the MsgBox function pops up a window with the text **"Do you want to quit now?"**, and two buttons, **Yes** and **No**. VBA recognizes **vbYesNo** as a constant corresponding to the correct number to make those buttons appear. If the user presses **Yes**, the function returns the number 6 (**vbYes**), while pressing the No button returns the number 7 (**vbNo**). So the first line in effect says, "If the user presses a button in this box, and the value returned is equal to the value stored in vbYes, the condition is true - process the next line (i.e., Quit)". If they click No, or cancel out of the box, the value will not equal 6 (**vbYes**), so nothing else happens.

Branching

A decision (a condition that is true or false) causes VBA code to branch, meaning that the code may no longer execute sequentially. If the user makes one decision, one set of code statements must be read and interpreted. If another choice is made, different instructions must be followed. VBA allows several methods for code branching.

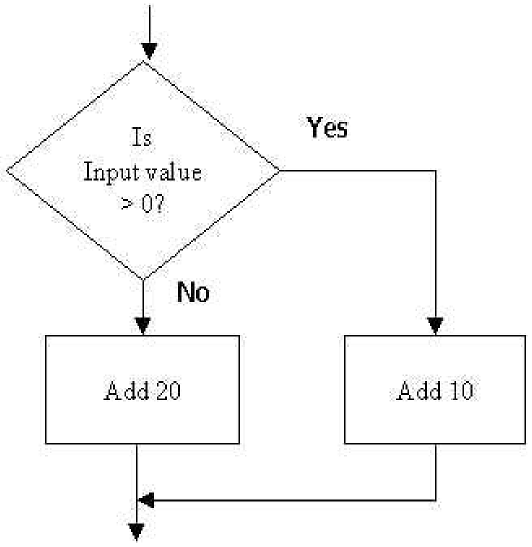
If condition Then

When the choice of two (or more) alternatives must be selected, and the results will follow directly from the choice made, the **If ..Then** structure is typically used. Its form is:

> **If** *condition* **Then**
>   Statements to run if the *condition* is true
> **Else**
>   Statements to run if the *condition* is false
> **End If**

The words in bold font are keywords to the process, and must appear. The word **Else** and the statements to follow if the condition is false are optional. Many conditional branches don't require an alternative action.

The **If ..Then** structure is often used where the flowchart branches out and downward. For instance, if the flowchart showed the following:



The condition to be considered is whether the input value is greater than zero. If that condition is true, then the input value is to be increased by 10. If not, the input value is to be increased by 20. If the input value is stored in cell B3, the code to accomplish this would read:

```
If Range("B3").Value > 0 Then
      Range("B3").Value = Range("B3").Value + 10
Else
      Range("B3").Value = Range("B3").Value + 20
End If
```

Note that the replacement of the value doesn't happen until after the entire line is read, so you can add to a value by replacing it with the existing value plus some amount.

You can use this type of structure to respond to the user's selection from a message box. If, for instance, you had run through a set of calculations, and wanted the user to decide whether to print the results, you might include the function:

```
MsgBox("Do you want to print these results?", vbYesNo, "Print?")
```

This would pop up a message box that asks "Do you want to print these results?", and offers the user two buttons - one marked Yes, the other No. If the user clicked Yes, the function would return the number 6 (**vbYes**). If they clicked No, it would return the number 7 (**vbNo**).

But the message box alone is not enough. VBA would be confused by a line of code that simply caused it to tell itself a number. How is VBA supposed to respond? An **If ..Then** statement tells it what to do:

```
If MsgBox("Do you want to print these results?", vbYesNo, "Print ?") _
   = vbYes Then
      ActiveSheet.Printout
End if
```

Note that everything between **If** and **Then** is a conditional expression which compares two values. The user supplied the value to the left of the equals sign by clicking on Yes (6) or No (7). The value to the right of the equals sign is the vbYes constant that VBA interprets to be the number 6. So, if the user clicks on Yes, VBA interprets this line to read:

```
If 6 = 6 Then
```

The conditional statement here is true - 6 is equal to 6, and so any statements up to the next key word (End If) will be processed, and the active sheet will be printed. If the user had clicked No, the statement would be interpreted:

```
If 7 = 6 Then
```

Here the conditional expression is false - 7 is not equal to 6, so VBA would skip to the next key word - End If - without processing any intervening statements. In this case, no alternative action is needed, so the key word Else is left off. Either the user wants the current sheet to be printed, which they indicate by clicking Yes, or they don't.

An **If ..Then** statement can be used to select from more than two choices by adding another key word - ElseIf. The form is:

```
If condition 1 Then
    Statements to be processed if condition 1 is true
ElseIf condition 2 Then
    Statements to be processed if condition 2 is true
Else
    Statements to be processed if neither condition is true
End If
```

You can include any number of alternate conditions in consecutive **ElseIf** statements.

Do Loops

PLEASE NOTE: Loops will not be covered at all in CSCA01. The remaining information in this document is provided for your interest only.

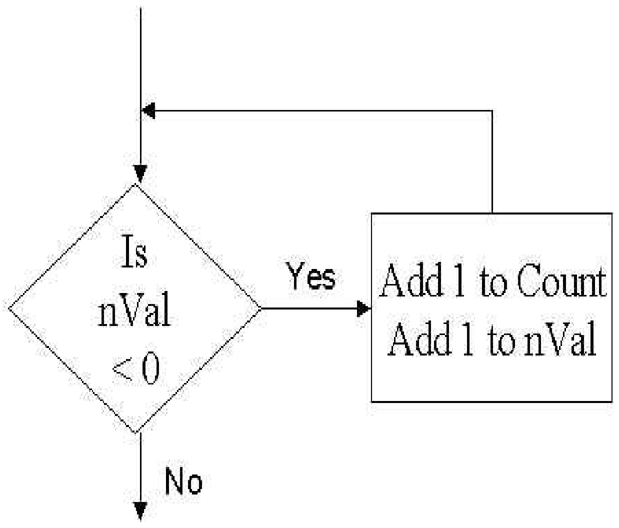
In many situations, you need the program to repeat the same set of steps until a specific condition is satisfied. The Do command can be stated in four ways, each offering a slightly different meaning.

```
Do While condition
   Statements to be processed so long as condition is true
Loop

Do Until condition
   Statements to be processed until condition becomes true
Loop

Do
   Statements to process regardless
Loop While condition

Do
   Statements to process regardless
Loop Until condition
```

The first two forms determine if the condition is true or false *before* processing any statements. In a "Do While" statement, if the condition is not true initially, VB will say, "The condition was never true, therefore the statements should never be processed." Likewise, a "Do Until" statement where the condition is initially true will be passed through without processing any internal statements - the condition must be false before VB will process the statements.

The last two forms call for the statements to be processed at least once. They will be processed immediately after VB reads "Do". When VB is done processing the statements, it evaluates the condition and determines if it should loop back and start over. The "Do...Loop While" form will loop back if the condition is true, while the "Do...Loop Until" form stops looping and moves on when the condition becomes true.

In flowcharting, the Do loop appears when a conditional response takes you backwards in the flowchart:



The VB statements that make this process occur are:

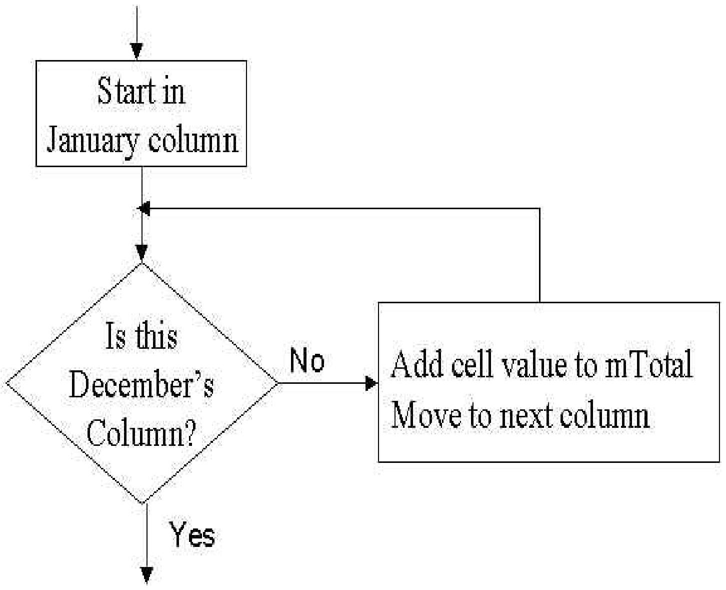
```
Do While nVal < 0
    Count = Count + 1
    nVal = nVal + 1
Loop
```

Note that this can be a dangerous process. If you've defined a condition that can never be satisfied, the program will loop forever (or until you get fed up and reboot)! In this case, if the condition is "Is nVal = 0?", you might have a problem. What if nVal = -0.5? The steps that follow only add 1 to the existing values, and nVal will never reach 0 exactly.

For Next Loops

In some situations, you need to have the program repeat a step for a known number of cycles. The For-Next loop allows you to name a counter variable, define the start and stop values for the counter, and repeat the steps for each counter value in that range.

If you had monthly totals listed in twelve adjacent columns, and wanted to total those values into a variable named *mTotal*, the flow chart might look like this:



Again, you could do this with the If-Then statement and some creativity, but the For-Next loop makes it easy. If the monthly total for January can be found in cell B16, and the totals for the other months are found in the next 11 columns of row 16, the code might look like this:

```
Cell("B16").Activate
For i = 1 to 12
    mTotal = mTotal + ActiveCell.Value
    ActiveCell.Offset(0,1).Activate
Next
ActiveCell.Value = mTotal
```

These instructions tell VBA to make cell B16 the active cell. Then it repeats two instructions 12 times: Add the value in the active cell to the variable mTotal, and move one cell to the right. If it starts in the January column, after 11 moves it ends up in the December column. The 12th step adds the value for December to the total, and moves to the right. The last instruction writes the total in the cell to the right of the monthly totals.

The counter value itself can actually be used in the steps to be performed. If the monthly totals are located in B16 - M16, and the yearly total is to be located in N16, the same process could be performed with these instructions:

```
Cell("N16").Activate
For i = 1 to 12
    ActiveCell.Value = ActiveCell.Value + ActiveCell.Offset(0,-i).Value
Next
```

This set of instructions makes the grand total cell active, and then stepping through the loop, adds the value in a cell "i" columns to the left to the existing value.

Either of the above methods will work, but your job as a programmer is to make them work as efficiently as possible. You decide which makes most sense to you, and which you can explain or modify most easily to meet the needs of the people who use your macros.