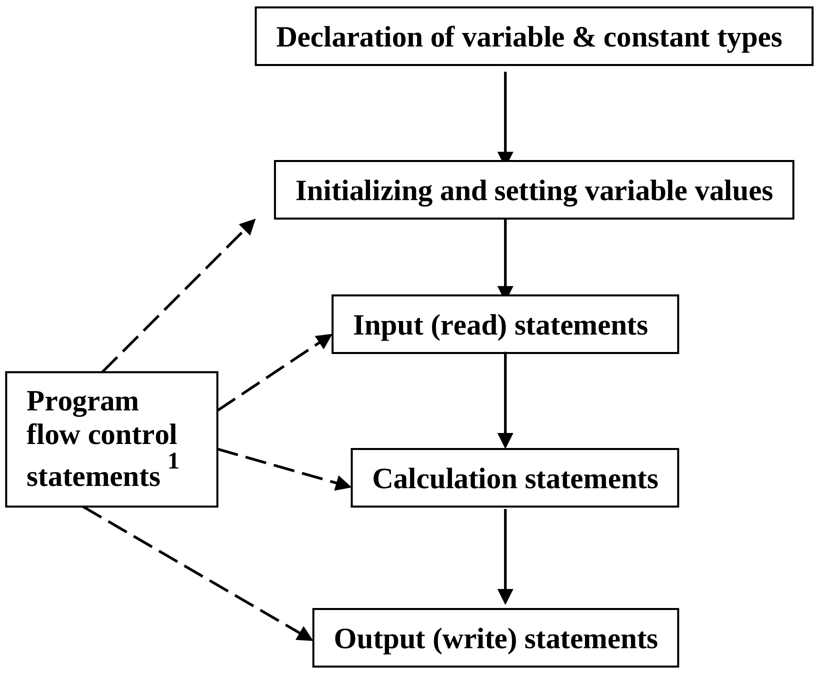
CHAPTER 1 INTRODUCTION TO PROGRAMMING

1.1 The Basic Program Structure

The programming of a scientific application involves re-formulating the analysis which you'd normally perform with pen, paper, and calculator as a set of instructions that follow a logical sequence. The collection of instructions is often referred to as a procedure or routine ; we'll simply call it a program. Different programming languages are available that allow us to communicate the set of instructions to the computer, and like all languages there are certain rules that must be followed when converting the general instructions into a form suitable for the particular programming language. These grammatical rules and structures are known as the syntax of the language.

The programming language which we'll be using is **Visual Basic for Applications** (**VBA**) which is a subset of the **Visual Basic** language. It was chosen for two reasons : i) simplicity -- it's not a difficult language, and can be learned quite quickly, and ii) availability -- **VBA** is included with all versions of **Excel** and **Word** since 1995. Once you've mastered the common underlying logic and approach to programming, you'll find learning to program in other languages a much easier process.

The basic structure of all of our programs can be described as follows :


Declaration of variable & constant types
Initializing and setting variable values
Input (read) statements
Calculation statements
Output (write) statements
Program flow control statements [1]

Each of these program components will be described later in the chapter and throughout the notes. At this time it will be more instructive for you to enter your first program and encounter a few of the program components in a practical environment. Do not be distressed if you don't understand a number of the concepts presented here. There is a vast amount of interrelated ideas and information that will hopefully become clear by the end of the course. The main thing *is to **"plow through", to keep on reading, and re-reading.*** Very often something that is not clear at first will become more palatable once you've been worn down through repeated exposure !

[1] Program flow control statements alter the sequence in which the program executes (sometimes in response to decisions that are made, or due to interaction with the user).

Before considering how programs are constructed, it is useful to understand how the program is accessed by the user. One of the major concerns here is that the user should not be able to inadvertently corrupt or alter the code; it follows that some sort of "barrier" or interface between the code and the user is highly desirable. The approach used by **Visual Basic** (**VB**), and other "visual" programming languages is to "attach", or associate, code with various **controls** such as buttons, scrollbars, check boxes, text boxes, option buttons, etc. The user then interacts with these controls, rather than directly with the actual program code. When a particular control is activated, such as by clicking on a button, moving a scrollbar, selecting an item from a menu, making a selection from a set of check boxes, etc. the code attached to that control is executed. In many cases the user does not need to interact with the program beyond specifying certain input values and problem parameters and pressing a **Run** button; the program then executes without interruption until the required output is produced. However, where a greater degree of interaction is required, such as when a user needs to make choices and express preferences, or when a simulation is being run and a value needs to be changed, additional controls are used (some of which are made to be visible only as required).

Professional and commercial **VB** applications are constructed on **User Forms** which hold the various controls. A user form in effect organizes, groups, and packages the controls. A good example of a user form based program is the **Chart Wizard** where the user is presented with a series of windows each of which is a **VB** type user form containing a variety of controls. The individual windows (or dialog boxes) guide the user through the interaction with the **Chart Wizard** program by prompting the user to make certain choices and selections. The user never sees the actual code[2] but nevertheless interacts with the program as it executes. **VB** programs are typically accessed via icons located either on the desktop or in the toolbar of a particular program such as **Word** or **Excel**.

Most engineers and scientific researchers working on a technical analysis or simulation that will not be widely distributed prefer to avoid the formalism of creating a user form and instead <u>place the controls directly onto an **Excel** worksheet</u>. This makes sense as scientific and engineering applications usually involve the input or generation of data on a worksheet. This approach will be used throughout the course.

These notes were created in reference to Excel 97; slight differences may occur in later versions.

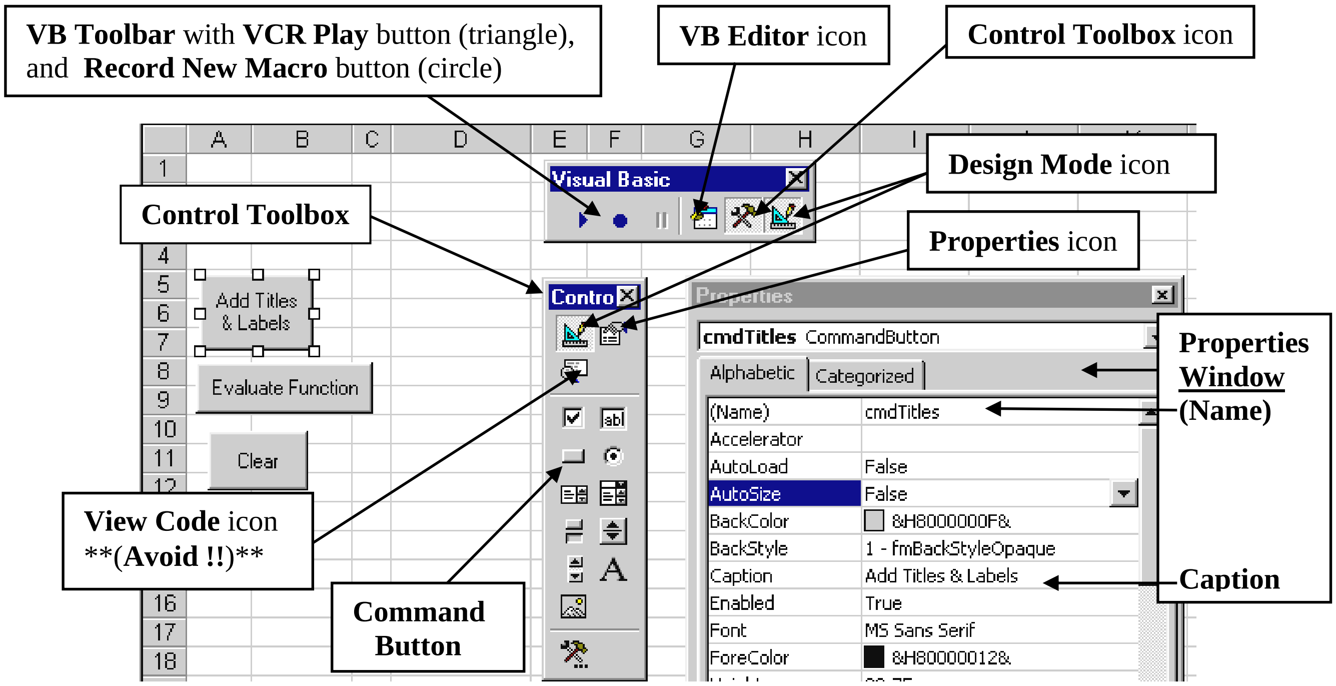
1.2 Example # 1 (Function Evaluation) -- Adding Controls

This example is designed to introduce you to creating a **VB** application; it constructs a very simple program to evaluate the function of two variables $\mathbf{z = -x^2 + ay + 2.1}$ where **a** is a known parameter. Each of the program components will be examined in detail in **section 1.4**. Don't worry too much about what you're doing for the time being, just follow the steps to create the program and get it working.

Open and save a new **Excel** file with some convenient name.
Display the **VB Toolbar** by going into the **View menu** ⇒ **Toolbars** ⇒ **Visual Basic**.
[The **VB Toolbar** for **Excel 2000** is slightly different than given here and can be seen on pg 1-7]
Open the **Control Toolbox** by clicking on the **Toolbox** icon on the **VB Toolbar** (the hammer & wrench icon -- see diagram below). [Note the distinction between Tool**bar** and Tool**box.**]

[2] The **Macro Recorder** [see **Appendix 19 : Legal Plagiarism**] provides a means of recording the actual **VB** code associated with most operations and activities occurring on the worksheet, including the **Chart Wizard** program, as we'll examine later.

A control is added by “drawing” it on the worksheet. Hover the cursor over the **Control Toolbox** until you find the **Command Button** icon, then click on the button icon and release. Move the cursor to a blank area of the worksheet, and click and drag to draw the button.



cmdTitles CommandButton

(Name)	cmdTitles
Accelerator	
AutoLoad	False
AutoSize	False
BackColor	&H8000000F&
BackStyle	1 - fmBackStyleOpaque
Caption	Add Titles & Labels
Enabled	True
Font	MS Sans Serif
ForeColor	&H80000012&

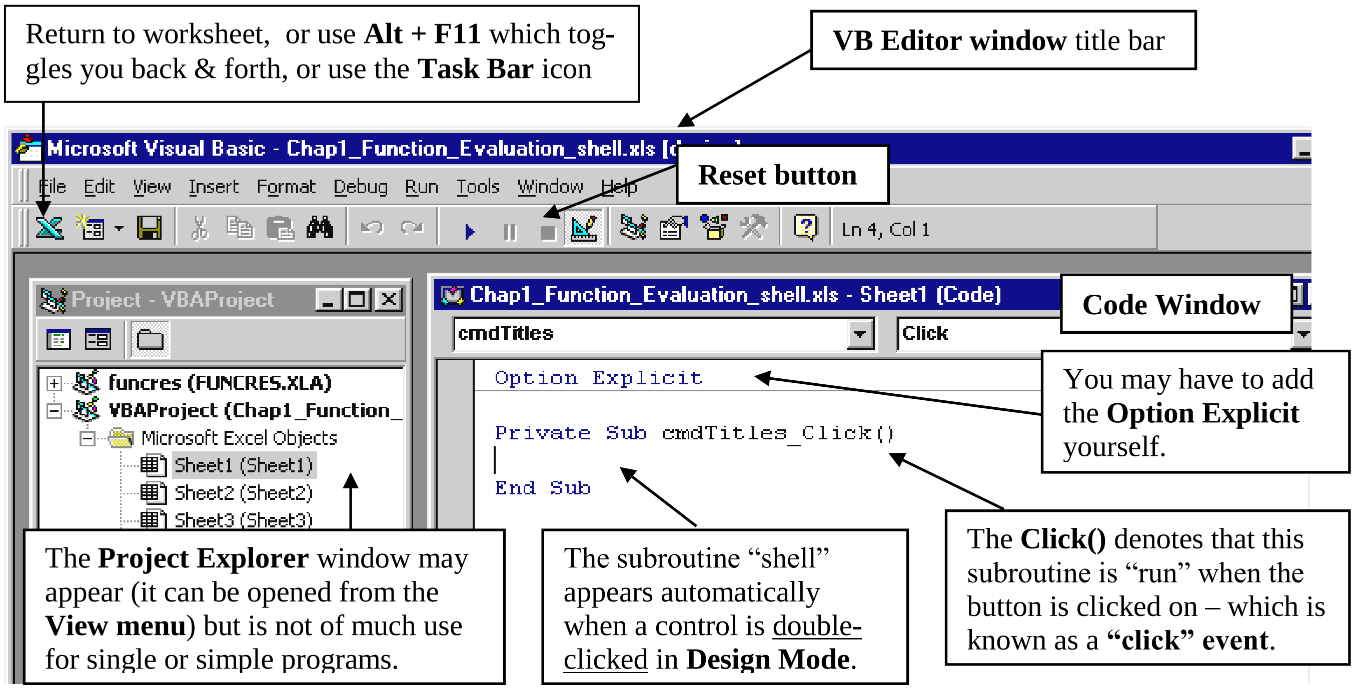
Every control has a set of associated properties ranging from the prosaic such as **colour**, **style**, **font** to the useful such as **caption**, and **left** and **top** location to the sophisticated such as **visible** (which allows the program designer to make the control invisible when not required). In addition, the control has a name (default name = commandbutton1) that is used to link it with its associated code which will be assigned the same name. Display the **Properties window** by clicking on the **Properties icon** found at the top portion of the **Control Toolbox**. Now, click on the **Command Button** that you’ve just added; small white boxes should appear around its periphery (if not, make certain that the **Design Mode icon** is active; that is, has been pressed). Change the **(Name)** property in the **Properties Window** to **cmdTitles** , and the **Caption** property to **Add Titles & Labels**. If you want the caption to appear on more than 1 line change the **WordWrap property** to **True**. Add two more command buttons : **Name = cmdEvaluate**, **Caption = Evaluate Function** ; **Name = cmdClear**, **Caption = Clear** . The “cmd” prefix in each name is an abbreviation of “command” and reminds the user that the control is a command button. Control prefixes are most useful in really large programs to remind the programmer of the type of control being referenced. A summary of control prefixes is given in **Appendix 5**.

Design mode refers to the occasion when the program is being designed, or constructed. When the **VB Editor** is in design mode a control’s properties can be set and the control can be moved and resized without causing it to be activated (and thus “running” the program). Clicking on a control in the **control toolbox** automatically puts the **VB Editor** in design mode. A program cannot be run if the editor is in design mode.

Caution : Older versions of VBA have a nasty tendency toward screen freezes !! Save frequently

1.3 Entering Code for Example # 1

Code is entered into either **subroutines** or **functions** created in the **Code Window** of the **VB Editor** (which is the window behind the worksheet where the program is stored). The easiest way to get into the **VB Editor** and the **Code Window** is by double-clicking on a control during **Design Mode**[3]. Something similar to the following set of windows should appear on your screen :


Microsoft Visual Basic - Chap1_Function_Evaluation_shell.xls [d
File Edit View Insert Format Debug Run Tools Window Help
Ln 4, Col 1
Project - VBAProject
funcres (FUNCRES.XLA)
VBAProject (Chap1_Function_
Microsoft Excel Objects
Sheet1 (Sheet1)
Sheet2 (Sheet2)
Sheet3 (Sheet3)
Chap1_Function_Evaluation_shell.xls - Sheet1 (Code)
cmdTitles
Click
VB Editor window title bar
Reset button
Code Window

```
Option Explicit

Private Sub cmdTitles_Click()

End Sub
```

Return to worksheet, or use **Alt + F11** which toggles you back & forth, or use the **Task Bar** icon

You may have to add the **Option Explicit** yourself.

The **Project Explorer** window may appear (it can be opened from the **View menu**) but is not of much use for single or simple programs.

The subroutine "shell" appears automatically when a control is double-clicked in **Design Mode**.

The **Click()** denotes that this subroutine is "run" when the button is clicked on – which is known as a **"click" event**.

Type **Option Explicit** (refer to pg 1-10) at the very top of the **Code Window** (as in the diagram above). Double-click on the **Add Titles & Labels** button; the following subroutine shell should be created :

The **Click()** automatically appended to the control name indicates what type of **event** triggers or causes the subroutine to run. Whenever the user clicks on this button the commands in the subroutine will be executed. Consult **Events** in **Appendix** 3 for more details.

Do not type in the subroutine shell; it should appear automatically when the control is double-clicked on the worksheet.

Private Sub cmdTitles_Click()

End Sub

Code will be entered inside the subroutine shell

******It is preferable to create subroutine shells automatically by double clicking on a control in design mode rather than by typing to avoid the possibility of mis-typing either the name or the event.***

[3] The **VB Editor** can also be reached from the worksheet by **Alt + F11** which toggles you back & forth with the worksheet, clicking on the **VB Editor icon** on the **VB Toolbar**, or clicking on the **Task Bar** icon that appears after the editor has been entered once. The **Code Window** can also be opened from **the View menu** as well as by double_clicking on **Sheet1 (Sheet1)** in the Project Explorer.

Add the following code inside the subroutine shell :

If you have a French version of Excel use Feuil1 instead of Sheet1. Refer to the note on pg 1-12 .

```
Private Sub cmdTitles_Click()
Worksheets("Sheet1").Range("B1") = " INPUTS"
Worksheets("Sheet1").Cells(1, 4) = " OUTPUTS"
Worksheets("Sheet1").Cells(2, 1) = " x =  "
Worksheets("Sheet1").Range("A3") = " y =  "
End Sub
```

WRITE statements. The text in quotes on the right side is written into the cell addresses given on the left side.

The subroutine shell can be thought of as equivalent to the front and back covers of a book : the first line, **Private Sub cmdTitles_Click()**, names the subroutine with the same name as the associated control and appends the type of event that activates the subroutine to the name, **_Click**() in this case. The last line, **End Sub**, identifies the point at which the subroutine code is finished. When the **cmdTitles** button is clicked (and the **VB Editor** is not in design mode) the compiler checks for a subroutine of the same name and upon finding it executes the 4 write statements given above.

Cell Addresses : The write statements contained in the **cmdTitles** subroutine referred to worksheet cell locations in two ways : i) the **city-map grid** description (eg **B4**) is the standard designation on a worksheet consisting of an alphabetic letter representing the column followed by a number representing the row, and ii) a **double numeric**, or **matrix type**, specification in which the columns are also represented by numeric values. With the matrix type designation the row number is given first, and the column number second (opposite to the conventional specification). For example, cell **B4** has an array-type location of **(4, 2)** meaning the cell in the 4th row and 2nd column. The matrix type description is particularly useful when loops are used to perform repetitive operations and arrays of data are either being read in or written out.

Double-click on the **Clear** button and enter the following code into its subroutine shell:

```
Private Sub cmdClear_Click()
Worksheets("Sheet1").Range("A2:A3").ClearContents
Worksheets("Sheet1").Range(Cells(1, 4), Cells(4, 4)).ClearContents
Worksheets("Sheet1").Range("B1").ClearContents
End Sub
```

Alternative is **Cells(1, 2).ClearContents**

Test the subroutines by exiting **Design Mode** (click on the icon) and clicking on the two buttons (but obviously not the **Run** button as no code has been attached to it). Two sets of titles and labels should appear and then be deleted by the **Clear** button. If an error message occurs press the **Debug** button on the message box to identify the offending statement. Any errors will likely be the result of typing errors ! The program must be **Reset** after the bugs have been corrected. There are two ways of resetting the program : i) pressing the **Reset** button (the **VCR Stop** type square button in the toolbar at the top of the **VB Editor**, ii) double-clicking on the **Design Mode** button which takes you in and out of Design Mode and ready to run again. If the program doesn't run after you've corrected the bugs, verify that you are, in fact, out of **Design Mode**.

Enter the following code into the **Evaluate Function** button subroutine shell :

```
Private Sub cmdEvaluate_Click()

Dim z As Double, x As Double, y As Double
Const a As Double = 6.2
Dim String1 As String

String1 = "The value of z is "

x = Worksheets("Sheet1").Range("B2").Value
y = Worksheets("Sheet1").Range("B3")

z = -x ^ 2 + a * y + 2.1

Worksheets("Sheet1").Range("D2") = z
Worksheets("Sheet1").Cells(3, 4) = String1 & z
Worksheets("Sheet1").Range("D4") = " x = " & x & " y = " & y

End Sub
```

Declaration of variable & constant types

Set the value of String1.

Read-in values of **x** & **y** from the worksheet

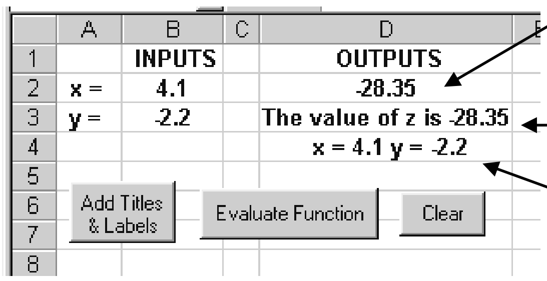
Calculation statement

Write **z** back on worksheet.

******* Include a single space before and after each ampersand **&**

Return to the worksheet, enter the value **4.1** in cell **B2** and **–2.2** in cell **B3**. Exit **Design Mode**, click on the **Evaluate Function** button, and verify that the answer for **z** is **-28.35** . Your worksheet should appear similar to the following :

Worksheets("Sheet1").Range("D2") = z



	A	B	C	D
1		INPUTS		OUTPUTS
2	x =	4.1		-28.35
3	y =	-2.2		The value of z is -28.35
4				x = 4.1 y = -2.2
5				
6				
7				
8				

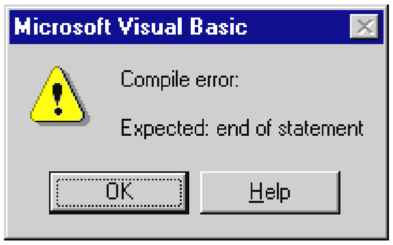
Worksheets("Sheet1").Cells(3, 4) = String1 & z

Worksheets("Sheet1").Range("D4") = " x = " & x & " y = " & y

Order of Program Execution : As there are no decision statements or branching statements (explained later) the code lines in the subroutine are executed one after the other in a linear fashion starting from the top. The values of **x** and **y** must clearly be read-in **before** calculation statement is reached. If one or both of the read statements are erroneously located after the calculation statement an error message will unfortunately **not** generally occur as the compiler will assume that the values are zero.

Upper & Lower Case Letters : VBA commands (but not variable, constant, and control names) will always appear with the letters having the same combination of upper and lower cases regardless of how they were typed in. For example, the expression **Worksheets("Sheet1"). Range("B2").Value** may be typed with any combination of upper and lower cases but will always appear in the form just given once the cursor is moved to another line. [The exception is the cell address which will remain as it was typed -- either **b2** or **B2** .] Variables, constants, and control names will always revert to the form that they were first declared either in a **Dim** statement in the case of variables and constants, or in the Properties window name box in the case of controls. **String1** was declared with a capital **"S"** in the **Dim** statement; if it was subsequently typed with a lower case "s", it would automatically be converted to its appearance in the **Dim** statement. Typing code can thus be sped up by using a single case and letting the compiler sort things out.

Spacing of typed in characters. The VB compiler is reasonably forgiving about how the code is typed in. For instance, if **C=2** was entered two spaces would automatically be added around the equal sign to give **C = 2**. That being said, there are a few situations in which a space must not be placed, and other places, such as before and after an ampersand, where it's absolutely essential. Leaving 1 or more spaces after the period and before the **Range** in **x = Worksheets("Sheet1"). Range("B2").Value** will produce the error message on the right :

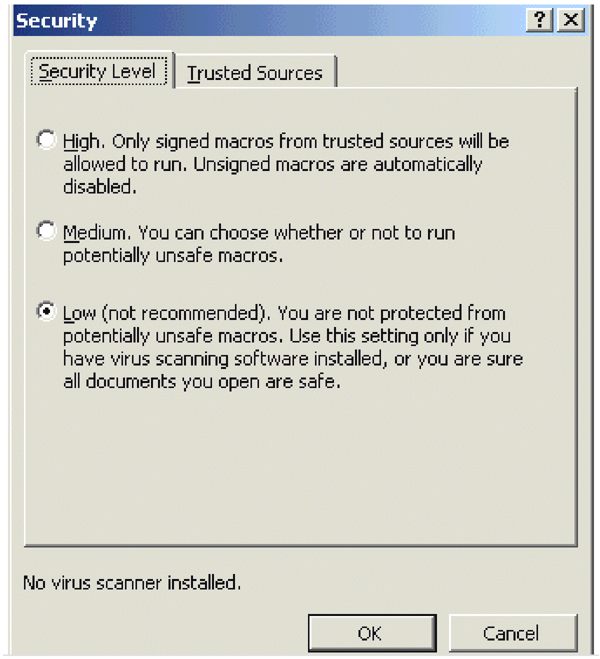

Microsoft Visual Basic

Compile error:

Expected: end of statement

Security Issues : Opening Files Created on Other Computers

Excel 2003 contains a security feature that prevents a file created on another machine from being run unless the security level is set at **Low**. (The file can be opened; it just can't be run.) Security levels are changed in the **Security** dialog box which can be accessed from the **VB Toolbar**, or via the **Tools Menu ⇒ Macro** . However, a change to the security level when a file is open will not register for that particular file. Instead, close the file, open a new workbook, change the security to low, and then open the desired file.


Security

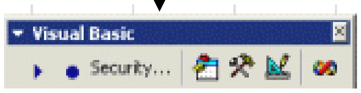
High. Only signed macros from trusted sources will be allowed to run. Unsigned macros are automatically disabled.

Medium. You can choose whether or not to run potentially unsafe macros.

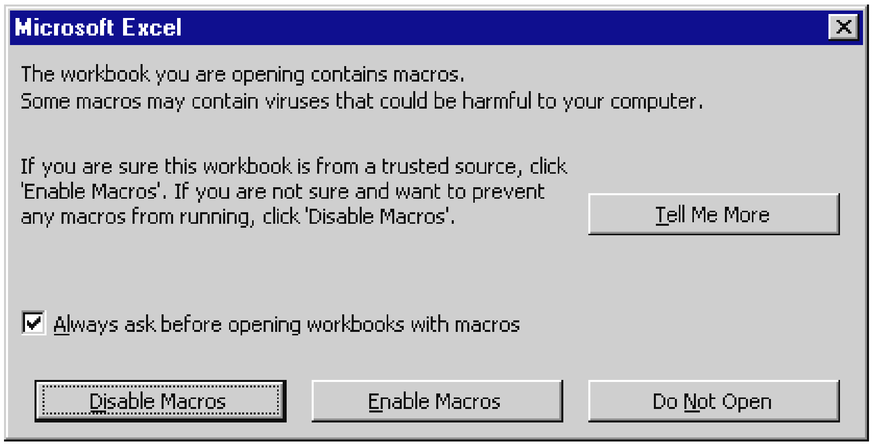
[✓ Low (not recommended). You are not protected from potentially unsafe macros. Use this setting only if you have virus scanning software installed, or you are sure all documents you open are safe.]

No virus scanner installed.

Clicking on **Security** on the **VB Toolbar** produces the dialog box on the right. Select Low.



If you attempt to open a VBA file using **Excel 97** you may encounter the dialog box given. The subroutines you've created are also referred to as **Macros**,or macro programs. A macro program uses a programming language, VBA in this case. The individual code statements are then converted into micro instructions that tell the processor exactly how to execute each code statement. A simple multiplication, for example, would involve a number of micro instructions.


Microsoft Excel

The workbook you are opening contains macros.
Some macros may contain viruses that could be harmful to your computer.

If you are sure this workbook is from a trusted source, click 'Enable Macros'. If you are not sure and want to prevent any macros from running, click 'Disable Macros'.

[✓] Always ask before opening workbooks with macros

Select **Enable Macros** which means to make them operative.

Do not be tempted to create a virus. As noted in the course outline, the department has a zero tolerance for such matters no matter how cute or funny it may seem to you. The sanctions will be swift and severe: at the very least you will be out of the course with a letter on your file.

This warning automatically results from a check-off in the Excel worksheet: **Tools menu ⇒ Options ⇒ General** tab check off **Macro Virus Protection**.

1.4 Examining the Program Components

1.4.1 Declaration of Variables & Constants :

Although you probably think only in terms of integer and decimal numbers, there are in fact many other types of data that need to be recognized by a computer (refer to the table below). We'll generally be using the **Double**, **Integer**, **Boolean**, and **String** types. **Double** is an abbreviation of **Double Precision** which was an old **Fortran** term used to describe large numbers that carry about 14 decimal places. Since **Double** is the default type for numerical data we'll use it for all decimal (**floating point**) numbers. The **Integer** data type is used in a few specialized applications such as to represent an index, or refer to a cell location (which can only be an integer), or with integer arithmetic. **Boolean** data types describe variables that have only two states **True** or **False** (or 1 or 0) and are generally used as "flags" to indicate when some condition or event has occurred . **Strings** hold alphanumeric values which are combinations of alphabetic letters, numerals, and other keyboard characters.

Data Type	Storage Requirement[4]	Range of Values
Byte	1 byte	0 to 255 (2^8 values available in an 8 bit binary number)
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to + 32,76**7** (16 bits : 1 bit used for + or - leaving 2^{15} = 32, 768 values. Zero uses one of the positive values)
Long (Big Integers)	4 bytes	-2,147,483,648 to 2,147,483,647 (32 bits : 1 bit used for + or - leaving 2^{31} = 2,147,483,648)
Single (single precision)	4 bytes	**-** 3.402823E38 to **-**1.401298E-45 (negative values) 1.401298E-45 to 3.402823E38 (positive values)
Double (double precision)	8 bytes	4.94065645841247E-324 to 1.79769313486232E308 (pos) -1.79769313486232E308 to -4.94065645841247E-324 (neg)
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Date	8 Bytes	January 1, 100 to December 31, 9999
String (variable length)	10 bytes plus string length	0 to about 2 billion
String (fixed length)	Length of string	1 to approximately 65,400.
Variant (with numbers)	16 bytes	Any numeric number within range of a Double
Variant (with characters)	22 bytes plus string length	0 to about 2 billion
Object	4 bytes	Any object reference
User defined	Varies	Varies by element

[**Excel 2000** also has a **Decimal** data type that provides 28 places to the right of the decimal point.]

---

[4] Computers work with **binary digits**(or **bits**) each of which has a value of either **1** or **0**. A collection of **8** bits is known as a **byte**.

[This table has been adapted from a table given in the **Help file** of the **VB Editor** under **Data types** ⇒ **Data Type Summary** or under **% (data type)** ⇒ **Data Type Summary** .]

It is good practice to explicitly declare the nature, or type, of a variable in order that the processor knows exactly the amount of storage to reserve to hold the value, but more importantly, to avoid certain errors (see **Error Trapping** below). If this is not done the variable is assumed to be of the default **Variant** type (which is referred to as **Implicit Declaration**). **Explicit Declaration** is made by means of **Dim, Public**, or **Static** statements (**Dim** is a shortened form of the earlier **Dimension** statement from the **Fortran** language). Constants are explicitly declared via the **Const** statement .

Examples of Explicit declaration

```
Dim z As Double, I As Integer, y As Double
Dim bSleep As Boolean, String1 As String
Const a As Double = 6.2
Static Index As Integer
Public Const Graph_Title As String = "Y versus X"
```

When a variable (or constant) is declared by a **Dim** or **Const** statement a portion of the computer memory is "named" with the variable name. Variable names thus represent storage locations. Whenever a variable is encountered in a code statement it is useful to interpret it as the value in the storage location having that variable name. Algebraic expressions therefore involve operations on the values contained in various storage locations. The expression **2*x – 5* y** should really be understood as twice the value stored in memory location **x** minus five times the value stored in memory location **y** .

A discussion of the implications of **Implicit** declaration as well the significance of **Public** versus **Private** declarations is given in the **Appendices 8 and 10** at the end of the notes.

Error Trapping. A variable that has had its data type explicitly declared cannot inadvertently be given a corrupted value without an error message been produced. For example, suppose that variable **x** in **Example # 1** had not been explicitly declared as **Double** and that the read statement for **x** had erroneously referred to cell **B1** instead of **B2**. The text **INPUTS** would have then been stored in the **x** memory location. When the statement to evaluate **z** was executed the text would have been converted to some numerical equivalent, and the computation would have occurred without the user realizing that anything was wrong. A typing mistake such as entering **4.X** instead of **4.1** in cell **B2** would similarly go undetected; the text **4.X** would be converted to some numerical value during the evaluation of **z**, with the user completely unaware of the error. If, however, **x** had been explicitly declared as **Double** an error message would have been produced in each case, and the user would be aware that something was wrong with the data.

Option Explicit : The **Option Explicit** statement is used to keep the programmer "honest"; it is placed at the very top of the code window before any other code. It acts by producing an error message each time an undeclared variable is encountered and thus forces the programmer to declare all variables. **Appendix 9** gives a check-off that causes **Option Explicit** to appear automatically.

1.4.2 Initializing and Setting Variable Values

Equal signs are used to assign values to variables. **String1** in **Example # 1** was assigned a value in this way with the statement **String1 = "The value of z is "**. [String data is always enclosed in quotes so that the compiler doesn't try to interpret the string as a single variable, or group of variables (which usually produces an error message).] Instead of reading the values of **x** and **y** off the worksheet it would have been possible to define their values by using the statements **x = 4.1** and **y = –2.2** . Although these appear to be normal algebraic statements, the equal sign here has an entirely different connotation (see the box below).

Initializing variables means to give them some starting, or initial, value (with the expectation that this value will change). In Chapter 3 you will create a counter that counts by threes beginning with the number 2 (2, 5, 8, 11, …). The value of the variable **Count** which is to hold the counted values must therefore be initialized to 2 . Initializing variables is generally only necessary when a running total is being computed (refer to chapter 3).

The meaning of equal signs. Equal signs have an entirely different connotation when used in computer programs. The use of an equal sign creates what is known as an **assignment statement**. Values are given to variables by entering numbers, text, or strings on the right side of an equal sign; these values are then assigned, or stored, into the specific storage location or object referenced on the left side of the equal sign. The statement **x = x + 3** (which is nonsensical according to rules of normal algebra) has a perfectly legitimate interpretation in the realm of computer math : evaluate the right hand side by adding **3** to the value currently found in storage location **x**, and then store the result back in location **x** !! Expressions of this type will prove enormously useful in running total calculations (refer to Chapter 3). Exceptions : equal signs actually denote equality in 3 situations: when part of conditions in **If** statements, **Select …Case** statements, and **Loop Until** statements (part of **Do…Loop**s) .

1.4.3 Read & Write Statements [Also known as **Input & Output (I/O)** statements]

In the context of our applications it will often be convenient to read data and variable values from cells on the worksheet directly into the program [although occasionally, and mostly for fun, we'll use an **Input Box** (refer to **Example # 2b)** in this chapter]. Similarly, once the numerical analysis has been completed it will be necessary to "write" or display the values somewhere, and again the worksheet is the most logical place. In **Example # 1** the following read and write statements were used to bring in values for **x** and **y** from the worksheet and to transfer the value of **z** back to two locations on the worksheet :

```
x = Worksheets("Sheet1").Range("B2").Value
y = Worksheets("Sheet1").Range("B3")
```

Basic read statements

```
Worksheets("Sheet1").Range("D2") = z
Worksheets("Sheet1").Cells(3, 4).Value = z
```

Basic write statements

Observe that the cell location is on opposite sides of the equal sign for the two situations. In view of the interpretation of the equal sign given above, the **read statement** takes the value stored in the cell location described on the right and assigns it to the memory location of the variable on the left. Conversely, the **write statement** takes the value stored in the memory space referenced on the right side and assigns it to the cell location on the left.

French versions of Excel use **Feuil1** instead of **Sheet1**, and so on, to identify the various sheets. This will cause problems if you use the full version of read or write statements : **Worksheets(“Sheet1”).Range (“C3”)** since **Sheet1** will not exist. The error message **Run-Time error ‘9’ : Subscript out of range** will be produced (pressing the **Debug** button will highlight the offending statement). There are 3 ways to work around this : i) try using the simplified version of the read or write statement --- **Range(“C3”)** so that the reference to **“Sheet1”** is eliminated, ii) replace **Sheet1** with **Feuil1** so that the instruction becomes **Worksheets(“Feuil1”).Range (“C3”)**, iii) go to the bottom of your worksheet, right-click on **Feuil1** tab, select **Rename** (Renom ??) and change the name from **Feuil1** to **Sheet1**. Fortunately there are not too many of such conflicts; the only other that I’ve seen involves **Graphique** being used for **Chart** .

Default conditions. A cell on the worksheet is an object that has a variety of properties such as colour, font, bold, value, etc. The default property of a cell is its value so that the **.Value** reference can be omitted and the compiler will still recognize that the cell value is being referred to. Similarly, for most versions of **Excel** the reference **Worksheets("Sheet1").** is a default condition which can be omitted . Therefore, any of the following could be used as read and write statements :

Read : y = Range("B3") or **y = Cells(3,2)** **Write : Range("D2") = z** or **Cells(3, 4) = z**

Caution : Some versions of **Excel** occasionally give error messages if **Worksheets("Sheet1").** is omitted.

Adding descriptive text to write statements. In many situations it is helpful to include some sort of description of the number being written into a cell. Text can be combined with a value by enclosing the text in quotes or defining it as a string. An ampersand **&** is used to link the text contained in the quotes or in the string with the variable value. For example, in **Example # 1** the statements

```
Worksheets("Sheet1").Cells(3, 4) = String1 & z
Worksheets("Sheet1").Range("D4") = " x = " & x & " y = " & y
```

led to **The value of z is -28.35** being written into cell **D3** and **x = 4.1 y = - 2.2** being written into cell **D4**. Units such as **m/s** can be attached to values in this way.

Formatting : The function value determined in **Example # 1** did not extend beyond two decimal places for the given values of **x**, **y** and **a**. In most situations, however, considerably more decimal places may be present. The way in which an output value is presented is referred to as its format. Formatting functions **Format( )** and **FormatNumber()** are available which allow a value to be presented with a specific number of decimal places and in a certain way such as in scientific notation, or as a currency, or a percentage, etc. For example, the following format specifications produce the output shown to the right of each statement :

```
Worksheets("Sheet1").Range("D5") = "z =  " & Format(z, "0.000")         ──→  z = - 28.350
Worksheets("Sheet1").Range("D5") = "z = " & Format(z, "scientific")     ──→  z = -2.84E+01
Worksheets("Sheet1").Range("D5") =  "z = " & Format(z, "0.00%")         ──→  z = -2835.00%
Worksheets("Sheet1").Range("D5") = "z = " & Format(z, "0.000E+00")      ──→  z = -2.835E+01
Worksheets("Sheet1").Range("D5") = "z = " & FormatNumber(z, 3)          ──→  z = -28.350
```

Note : I have been unable to obtain more than 2 decimal places using **Format(z, "0.000E+00")** ; the output above was obtained with **Excel 97**.

1.4.4 Calculation Statements & Operator Precedence

The familiar spreadsheet arithmetic operators are also used by the **VB Editor** and are given below in the order of their **precedence** (that is, in the order of which operations occur before which other operations).

Operator	Type	Use	Description
^	Exponentiation	x^y	Raises x to the power y
–	Negation	– x	Negates x
*	Multiplication	x*y	Multiplies x and y
/	Division	x/ y	Divides x by y
\	Integer division	x\ y	Divides x by y and returns the integer result
Mod	Modulo operator	x **Mod** y	Divides x by y and returns the remainder
+	Addition	x + y	adds x and y
–	Subtraction	x – y	Subtracts y from x

[The **Int()** function also performs integer division by returning the integer part of the argument.]

Special attention must be given to the precedence, or hierarchy, of the operations so that the exp-ressions are evaluated in the desired way. It is usually quotients that lead to an improper sequence of operations. For example, suppose you wished to evaluate $z = \frac{4 + 6}{2 + 3}$.
Writing this quotient as z = 4 + 6 / 2 + 3 would produce the incorrect result of 10 since according to the hierarchy the division 6/ 2 would be evaluated first, and then added to 4 and to 3. Parentheses can be used to force the evaluation of whatever is inside the parentheses first (of course inside the parentheses the hierarchy of operations will still be in effect unless additional parentheses are included). However, adding parentheses only to the numerator of the quotient in this example would still give an incorrect answer : z = (4 + 6)/ 2 + 3 $\Rightarrow$ 8 [the 4 and 6 are added first because of the parentheses to give 10, which is then divided by 2, and finally added to 3]. The correct approach here is to write z = (4 + 6)/ (2 + 3) which gives 2 .

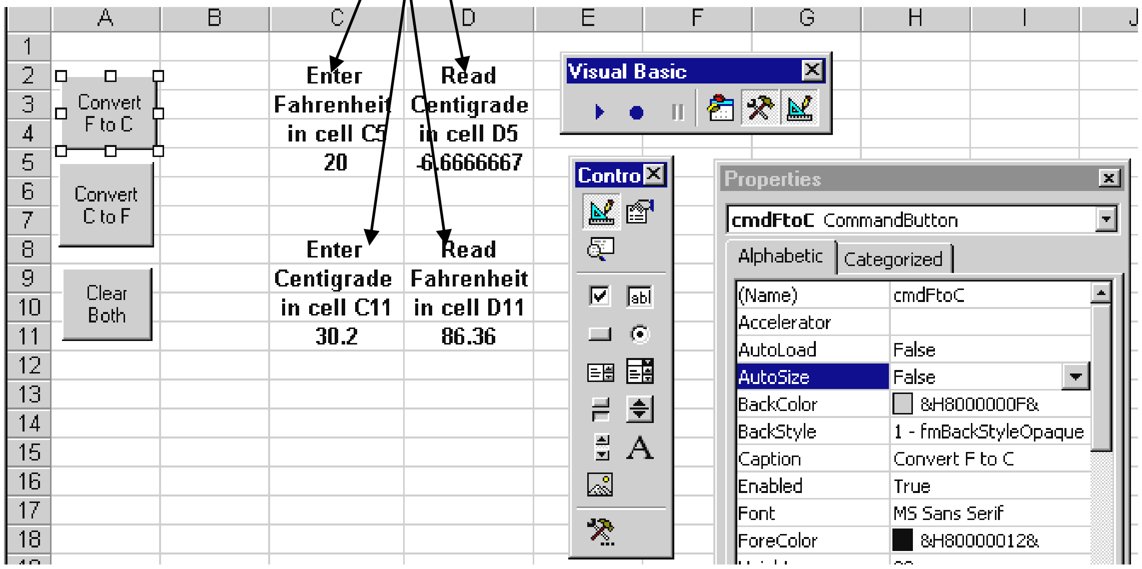
Disturbingly, operator precedence on the worksheet has negation occurring before exponentiation. This means that if you wish to evaluate exp(-x^2) then entering the formula =Exp(-(x^2)) into a cell produces a correct answer while entering = Exp(-x^2) does not ! Check this out in Excel Help : Operator Precedence $\Rightarrow$ About calculation operators $\Rightarrow$ The order in which Excel performs operations in formulas. Compare with VBA Help under Operator Precedence.

1.4.5 Program Flow Control Statements

Example # 1 presented a simple linear program; the code lines are executed one after the other. Complex programs require a more sophisticated type of program flow which is directed by program flow control statements of the following types : **conditional or decision statements** (such as **If…Then…Else** or **Select Case** statements) **, looping statements** (**For…Next** or **Do…Until** statements)**,** and **branching statements** (such as **GoTo** statements). These statements allow program execution to assess a condition and chose an outcome, to repeat certain operations, and to skip around specific code. **Looping** statements are examined in Chapter 3 (and in the **Appendix 14 & 15**), **GoTo** statements are introduced in **section 2.3** (and periodically thereafter), and **If** statements will be considered below in **section 1.6**. [**Select Case** statements are presented in the **Appendix 23**].

<u>1.5</u> **<u>Temperature Converter</u>**. **<u>Example # 2 a</u>) :** This example is a lead-in to two more interesting versions . Open and save a new Workbook with appropriate name. Add 3 buttons : **Name = cmdFtoC**, **Caption = Convert F to C** (set the **Word Wrap** property to **True**) ; **Name = cmdCtoF**, **Caption = Convert C to F** ; **Name = cmdClear**, **Caption = Clear Both**.

Type in the titles indicated. Or, if you wish, you could use a **cmdTitles** subroutine with 12 write statements (as in **Example #1**).



	A	B	C	D
1				
2			Enter	Read
3	Convert F to C		Fahrenheit	Centigrade
4			in cell C5	in cell D5
5			20	-6.6666667
6	Convert C to F			
7				
8			Enter	Read
9	Clear Both		Centigrade	Fahrenheit
10			in cell C11	in cell D11
11			30.2	86.36

Properties

cmdFtoC CommandButton

(Name)	cmdFtoC
Accelerator	
AutoLoad	False
AutoSize	False
BackColor	&H8000000F&
BackStyle	1 - fmBackStyleOpaque
Caption	Convert F to C
Enabled	True
Font	MS Sans Serif
ForeColor	&H80000012&

Alternative: Read ***F*** *value directly into the formula and store the result directly into the output cell.*
Range("D5") = (5 / 9) * (Range("C5") – 32)

```
Option Explicit

Private Sub cmdFtoC_Click()
Dim F As Double, C As Double
F = Worksheets("Sheet1").Range("C5")
C = (5 / 9) * (F – 32)
Worksheets("Sheet1").Range("D5") = C
End Sub

Private Sub cmdCtoF_Click()
Dim F As Double, C As Double
C = Worksheets("Sheet1").Range("C11")
F = (9 / 5) * C + 32
Worksheets("Sheet1").Cells(11, 4) = F
End Sub
```

Alternative (3 lines replaced by 1) :
Cells(11, 4) = (9 / 5) * Range("C11") + 32

```
Private Sub cmdClear_Click()
Worksheets("Sheet1").Range("C5:D5").ClearContents
Worksheets("Sheet1").Range("C11:D11").ClearContents
Beep
End Sub
```