Visual Basic Lecture 2

In Maple programming, we extensively used and manipulated variables in our programs with little or no regard to how the computer accessed or viewed those variables. In this set of lecture notes, we will study how the use of variables and their types in VB.

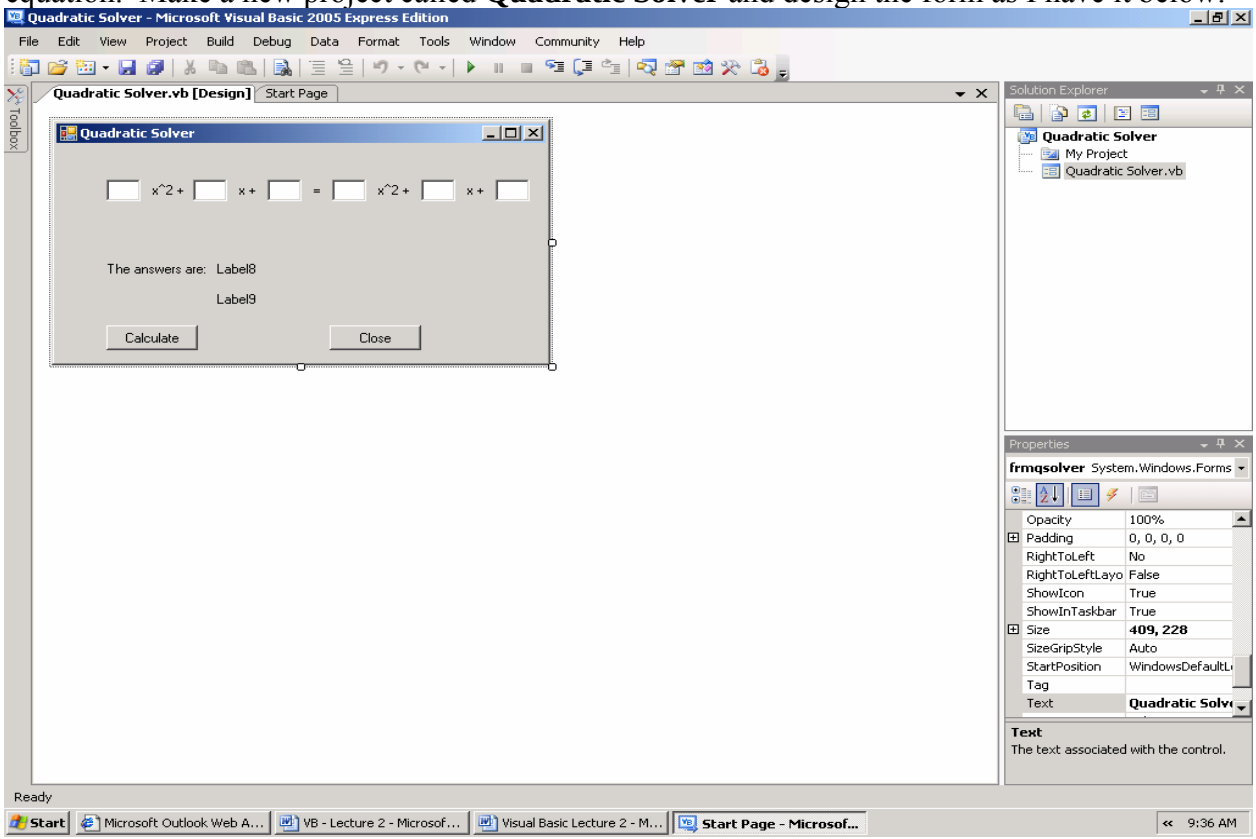
When you want to use a variable, you must first tell the computer to create a space for it in memory. This call must tell the computer exactly how much memory to use for this variable. The size of a variable is based upon its dimension. For instance, if you wanted a variable **j** that represented a counter for a loop, then you would probably dimension **j** as integer. The call to the computer would read like this:

```
Dim j as integer
```

The type integer tells the computer that the variable **j** can only be an integer, and the computer sets aside a pre-determined amount of memory to store the variable **j**. If the number that **j** represents gets too large and exceeds the memory size that has been allotted for it, then the program will crash with a "**Stack Overflow"**.

When performing computations that will result in floating point decimals, you will want to dimension those variables as Single or Double. These types allow the variables to be floating-point decimals of different length. The type Double will allow a much larger floating point number, but uses much more memory than Single.

Let's work now on creating a program that will solve quadratic equations. We know from the quadratic formula that equations of the form $ax^2 + bx + c = 0$ have the two solutions $x = (-b \pm \sqrt{(b^2 - 4ac)}) / (2a)$. Let's create a program that takes equations of the form $ax^2 + bx + c = dx^2 + ex + f$, converts the equation to the standard form of $ax^2 + bx + c=0$, and then solves the equation. Make a new project called **Quadratic Solver** and design the form as I have it below:



We see that we will have six text boxes that the user will type in that values of ***a, b, c, d, e,*** and ***f***. I have named these **txta, txtb, txtc, txtd, txte,** and **txtf.** When they click on the **Calculate** button, the computer will need to move all of the terms on the right side of the equation to the left, add the like terms, and have the new variables ***a, b,*** and ***c*** that it will use in the quadratic equation. Double-click the button **Calculate** to generate the sub-procedure that handles that button being clicked.

We need to create variables that we will use to retrieve the user input. We could name these of ***a, b, c, d, e,*** and ***f***, but a slight problem arises in that one of the arguments that the procedure receives goes by the variable name ***e***. Also, **VB is not case sensitive!!** What I have done is names them of ***aa, bb, cc, dd, ee,*** and ***ff***. You will probably want to dimension them as Single. We want the variable ***aa*** to represent the text that was typed into the text box **txta**. To make this assignment, use the command:

```
Dim aa as Single
aa = txta.Text
```

You could also have created the variable and made the assignment on one line:

```
Dim aa as Single = txta.Text
```

However, since all six variables need to be made with the same dimensions, you might want to use the following:

```
Dim aa, bb, cc, dd, ee, ff as Single
aa = txta.Text
bb = txtb.Text
cc = txtc.Text
dd = txtd.Text
ee = txte.Text
ff = txtf.Text
```

We also will need the variables ***a, b,*** and ***c*** that we will use in the quadratic equation. Simple algebra shows that:

```
a = aa – dd
b = bb – ee
c = cc – ff
```

You can include these variables on the same line that you used to dimension the other variables. We will also need two variables to represent the two solutions to this quadratic equation. So create the variables **x1** and **x2** along with the rest. If you want to create a variable **D** that represents the discriminant, which is the quantity $\mathbf{b^2 - 4ac}$, this might simplify the problem. Your first lines of code in the sub-procedure should read:

```
Dim aa, bb, cc, dd, ee, ff, a, b, c, x1, x2, D as Single
aa = txta.Text
bb = txtb.Text
cc = txtc.Text
dd = txtd.Text
```

ee = txte.Text
ff = txtf.Text
a = aa – dd
b = bb – ee
c = cc – ff
D = b^2 – 4*a*c

Now we're ready to solve the quadratic equation. The only thing we're missing is the exact method of taking square roots in VB. All mathematical operations in VB can be accessed through **Math.Procedurename(value)**. For instance, to take the square root of a number **n**, you would type **Math.Sqrt(n)**.

With this in mind, we solve the equations for the two x values:

x1 = (-b + Math.Sqrt(D)) / (2*a)
x2 = (-b - Math.Sqrt(D)) / (2*a)

Now we need a way to display the solutions. If you look at the form, there are two labels that we have not placed text in. Mine are named **lblsolution1** and **lblsolution2**. We now output our solutions to those labels:

lblsolution1.Text = x1
lblsolution2.Text = x2

Feel free to test this program on different values. You will notice that it sometimes does not work. For instance, try 5x^2 – 4x + 3 = 4x^2 + 4x + 2. You will see that the output is **NaN**. Why? The solution to this quadratic equation is a complex number. So how can we get our program to handle complex numbers? In order to understand this, we must first realize when complex numbers are the solution.

We get an imaginary solution to a quadratic equation if the discriminant **D = b^2 – 4ac < 0.** In other words, if what is being square-rooted is a negative number, then the result is imaginary. What does this mean in terms of programming? We know then that if D is positive, then we can use the code we already have. If not, then we need a new way to output the result.

There are two ways that we can handle this situation. We can have the computer test if **D** < 0, and if so, then we can make a **Message Box** appear warning the user that the equation contains imaginary roots. In order to use a **Message Box**, we must first **Import** a specific class that handles the creation of these **Message Boxes**. **Importing** is similar to loading additional packages as we saw in Maple. For instance, if we wanted to work with matrices, we would have loaded the Linear Algebra package using **with(LinearAlgebra)**. In VB, we will import a package called **System.Windows.Forms.Form**. The call to do this is:

```
Imports System.Windows.Forms.Form
```

This line of code should appear before the Public Class line. Once we have this line of code, we can now generate **Message Boxes**. The code needs to test **D**, and generate the message box if **D** < 0, or calculate the two values if **D**>=0. The following code should be inserted after the computation of the variable D.

```
If D < 0 then
  MessageBox.Show("This equation will result in complex solutions!", _
```

```
    "Complex numbers", MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
Else
    x1 = (-b + Math.Sqrt(D)) / (2*a)
    x2 = (-b - Math.Sqrt(D)) / (2*a)
    lblsolution1.Text = x1
    lblsolution2.Text = x2
End if
```

The procedure **MessageBox.Show** takes as its first argument the text that you want displayed in the message itself. Its second argument is the caption that will be displayed at the top of the message box. The third argument allows you to specify what buttons will be on the message box, and the fourth determines what icon will appear.

This method allows us to bypass the problem of having to solve the complex problems by generating a Message Box if the program determines that the equation will be complex. In order to actually solve the complex equations, a new method of outputting the result must be made.

Why must we have a new way to output the result? Our output will need to contain both the real and imaginary components, and the imaginary component must be written with the letter **i**. The proper form of a complex number is **a + bi**, where **a** and **b** are real numbers, and **i** is the square root of -1. Since we will be outputting the letter **i**, which is not a number, we will need to change out output type to **String**. A **String** is a variable that contains text.

We're going to output the values **x1** and **x2** as **String**, so make the appropriate change to the first line of the procedure. Now we need a conditional statement that evaluates if **D** > 0, and if it is, then we need to evaluate **x1** and **x2** in a way that handles these imaginary numbers. If we break apart the two portions of the quadratic formula, we get:

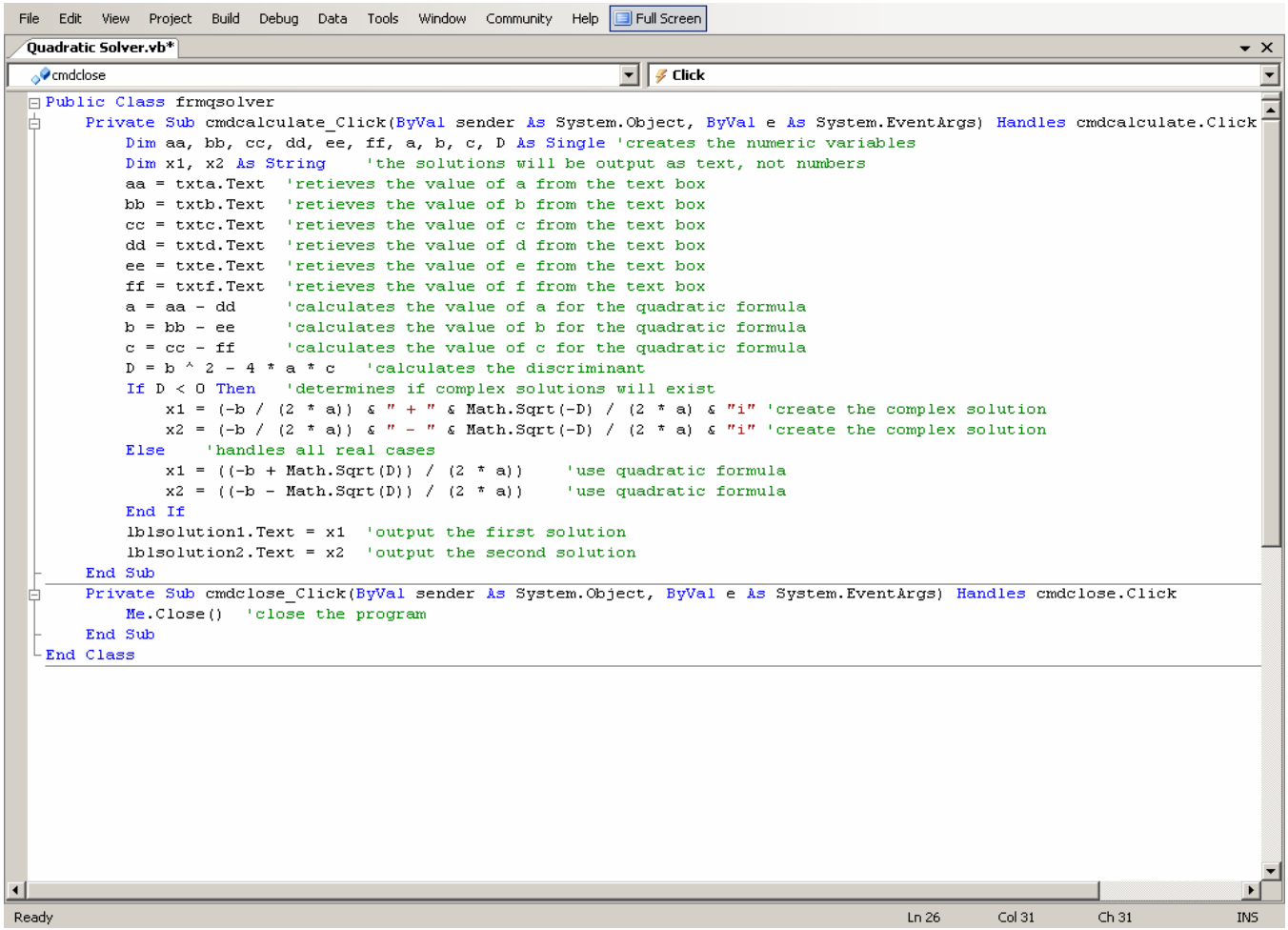
$$x = -b/(2a) \pm (\sqrt{(b^2 - 4ac)}) / (2a)$$

Obviously, the first term, -b / 2a, can never be imaginary. This is called the **real** part of x. The second term is the part that can become negative in the square root, giving us the imaginary number. But remember this: if b^2 – 4ac is negative, then 4ac – b^2 is positive, and has the same numerical value. For instance, 4 – 10 = -6, but 10 – 4 = 6. So if we know that the discriminant **D** is negative, then we can evaluate the square root of –**D**, to get a real number. Then we multiply that real number by **i** to get the imaginary part of the solution. We will need to use a command similar to the way we showed both text and variables using the **print** command in Maple. The correct way to put all of this together is:

```
x1 = (-b / (2 * a)) & " + " & Math.Sqrt(-D) / (2 * a) & "i"
x2 = (-b / (2 * a)) & " - " & Math.Sqrt(-D) / (2 * a) & "i"
```

Notice that the & is used between the terms that will be output. Any text that we want to display is given in the quotation marks. This means that the computer will set the variable **x1** to the line of text given by the computation (-b / (2 * a)), followed by a plus sign, followed by the computation Math.Sqrt(-D) / (2 * a), followed by the letter i.

The final body of code should look like what I have below. Also note that I have commented the body of this code to tell people viewing the code exactly what each line of code is supposed to do. Comments can be made in VB by using the apostrophe (‘) key. Comments are useful to other programmers who must view your code in order to find potential errors. A well commented program will be easy to follow, and will help others to follow the logic of your program.



Before building this program, don't forget to erase the words Label8 and Label9 from the two labels.

Homework1: Write a program that computes the factorial of a number that the user gives. The user should be able to type the number into a text box, press a Calculate button, and the Factorial of that number will appear. We wrote a similar program in Maple which you may use as a reference, but note that loops are done differently in VB. Here's the basic structure of a For loop in VB:

```
For i =  start To end Step amount
Statement
Next
```

In Maple, we used the keyword by to denote how much the counter would be incremented. In VB, we use the word Step. Also, notice that the loop ends with Next, not end do. Also, you will set the counter equal to its initial value with an equal sign, not the word from. For example, a program in Maple that summed up the number from 1 to n would read:

```
SUM:=0;
for i from 1 to n do
  SUM:=SUM + i;
end do;
```

In VB, this would be written as:

```
Dim i, SUM as Integer
SUM = 0
For i = 1 To n Step 1
SUM = SUM + i
Next
```

Another way to write this would be:

```
Dim i, SUM as Integer
SUM = 0
For i = 1 To n Step 1
SUM += i
Next
```

Note that SUM += i means SUM = SUM + i.

Homework 2: Create a program that could be used to order a pizza when a customer walks into a pizza restaurant. They should be able to select Large, Medium, or Small, with additional charges placed if it is to be a stuffed crust, pan, hand-tossed, etc. Allow them to select up to five toppings for the pizza (cheese, pepperoni, sausage, etc). You may decide what the cost of each item will be (nothing is free, however). Have a pop-up window appear if the user selects the same topping twice (except for cheese…you can never have too much cheese on pizza). Make sure that your program has all of its basic functionality before you start adding extra items. If you wish, you may create options to purchase drinks, extra pizzas, appetizers (breadsticks, wings, etc), but this is not required. Please comment you project.