Arrays Part 2

We’ve covered enough material so far that we can write very sophisticated programs. Let’s cover a few more examples that use arrays.

First, once in a while it may be useful to be able to access controls on your forms through an array. Consider a form with five textboxes, and a button. We might want to calculate the average of all the numbers in the textboxes:

![Image of textboxes](image)

If the textboxes are named txtNum1, txtNum2, txtNum3, etc. then we could write code in the Average button to the tune of:

```vbnet
Dim intAve As Integer
intAve = CInt(txtNum1.Text) + CInt(txtNum2.Text) + _
        CInt(txtNum3.Text) + _
        CInt(txtNum4.Text) + CInt(txtNum5.Text)
intAve = intAve \ 5
MessageBox.Show("The average is " & intAve)
```

This works fine, but it is somewhat tedious. Besides, what if we had 50 textboxes? It would be much nicer if there was some way to make an array of textboxes and then we could loop over the array:

```vbnet
sum = 0
For i = 0 to aryTextBoxes.Length – 1
    sum += CInt(aryTextBoxes(i).Text)
Next
intAve = sum \ aryTextBoxes.Length
```

If we could set up this array, then the loop will sum up all the entries in the textboxes and divide by the number of entries to get the average.

There are a couple of ways to perform the above; the simplest way is a bit tedious to set up, but once set up we can use the arrays for all our references.
In this simplest method we do the following:

1) Design our form with all the textboxes, labels, etc. For example, we might make a form with 5 textboxes.
2) Make an array of with the same data type as the control added to the form. The array should be the same size as the number of controls on the form. For example, we might make an array of TextBox that can hold five textboxes (0-4):

\[
\text{Dim aryTextBoxes(4) as TextBox}
\]

3) Assign each entry in the array to one of the textboxes on the form. You have to do this in code, so normally it would go in someplace like the form load event, so it is executed once when the program first starts up:

\[
\begin{align*}
\text{aryTextBoxes(0)} &= \text{txtNum1} \\
\text{aryTextBoxes(1)} &= \text{txtNum2} \\
\text{aryTextBoxes(2)} &= \text{txtNum3} \\
\text{aryTextBoxes(3)} &= \text{txtNum4} \\
\text{aryTextBoxes(4)} &= \text{txtNum5}
\end{align*}
\]

This is the part that can be tedious, but it only has to be done once. This sets a reference or pointer to the textbox on the form:

4) You can now access the array and it will be referencing one of the textboxes; e.g. aryTextBoxes(3).Text will access the same thing as txtNum4.Text.

We could now programmatically compute the average of all textbox entries as:

\[
\begin{align*}
\text{Dim intAve As Integer} \\
\text{Dim sum As Integer} &= 0 \\
\text{Dim i As Integer} \\
\text{For } i = 0 \text{ To aryTextBoxes.Length - 1} \\
\text{sum} &=\text{sum} + \text{CInt(aryTextBoxes(i).Text)} \\
\text{Next} \\
\text{intAve} &= \text{sum} \div \text{aryTextBoxes.Length} \\
\text{MessageBox.Show("The average is ", intAve)}
\end{align*}
\]
Grade Calculator

Here is an example that may actually be of practical use to you! Blackboard’s gradebook stores grades for you, but doesn’t let you run any “what-if” scenarios to see what your final grade will be depending upon what you get on assignments that have not yet been graded. For example, you might want to know what grade you need to get on the Final in order to get at least 90% in the class. Let’s write a program that calculates your grade (as a percentage) based on values you can type into a form.

In our case, we have three categories:

- Exams 50%
- Assignments 30%
- Labs 20%

Each item per category is worth the same amount. To compute the percentage of your final grade based on the exams we would use the formula:

\[
ExamPercent = (0.5) \times \left( \frac{\sum \frac{GradeExam_i}{TotalPo int sExam_i}}{n} \right)
\]

Similarly, we can compute the percentage from assignments and labs:

\[
AssignmentPercent = (0.3) \times \left( \frac{\sum \frac{GradeAssign_i}{TotalPo int sAssign_i}}{n} \right)
\]

\[
LabPercent = (0.2) \times \left( \frac{\sum \frac{GradeLab_i}{TotalPo int sLab_i}}{n} \right)
\]

Our total percentage is then \( ExamPercent + AssignmentPercent + LabPercent \)

In our specific class, we have two exams, six labs, and five assignments. Let’s design our form as follows:
The exams textboxes are named txtExam1 and txtExam2, with their max scores in txtExam1Max and txtExam2Max. I set default values for all the textboxes.

The same naming scheme applies to the labs and the assignments. The labs are named txtLab1, txtLab2, etc. and their max scores in txtLab1Max, txtLab2Max, etc. The assignments are txtAssignment1, txtAssignment2, … txtAssignment1Max, txtAssignment2Max, … finally, the overall grade will be displayed in lblGrade.

First, we can allocate space for all of the arrays. These are created as class variables:

```vbnet
Dim aryLabGrades(5) As TextBox ' Six lab grades
Dim aryLabGradesMax(5) As TextBox ' Max score for each lab
Dim aryExamGrades(1) As TextBox ' Two exam grades
Dim aryExamGradesMax(1) As TextBox
Dim aryAssignmentsGrades(4) As TextBox ' Five homework grades
Dim aryAssignmentsGradesMax(4) As TextBox
```

Next, in the Form_Load event, we can manually set each array entry to the corresponding textbox on the form:

```vbnet
' Assign textboxes on the form to slots in the arrays
aryLabGrades(0) = Me.txtLab1
aryLabGrades(1) = Me.txtLab2
aryLabGrades(2) = Me.txtLab3
aryLabGrades(3) = Me.txtLab4
aryLabGrades(4) = Me.txtLab5
aryLabGrades(5) = Me.txtLab6
aryLabGradesMax(0) = Me.txtLab1Max
aryLabGradesMax(1) = Me.txtLab2Max
aryLabGradesMax(2) = Me.txtLab3Max
aryLabGradesMax(3) = Me.txtLab4Max
```
Finally, we will need to add the code in the Button Click event to compute the final grade based upon what the user enters into the textboxes. Here is how we would compute just the Exam percentage, which was algebraically specified as:

\[
\text{ExamPercent} = (0.5) \times \left( \frac{GradeExam_1}{TotalPo int sExam_1} + \frac{GradeExam_2}{TotalPo int sExam_2} + \ldots + \frac{GradeExam_n}{TotalPo int sExam_n} \right)
\]

In this case we need a loop to calculate the numerator of the equation, divide by the number of entries in the array, then multiply by 0.5:

```vbnet
Dim sngExamComponent As Single = 0
Dim i As Integer

For i = 0 To aryExamGrades.Length - 1
    sngExamComponent += CSng(aryExamGrades(i).Text) / _
                        CSng(aryExamGradesMax(i).Text)
Next
sngExamComponent = sngExamComponent / aryExamGrades.Length
sngExamComponent = sngExamComponent * CSng(Me.txtExamPercent.Text)
```

The loop goes through each exam grades and adds together the grade divided by the max. The whole thing is then divided by the number of grades and finally multiplied by the weight for exams (in this case 0.50).

We can repeat this process for the lab and assignment component, then add them together to get the final grade. Here is the whole piece of code:
```vbnet
Dim sngExamComponent As Single = 0
Dim sngAssignmentComponent As Single = 0
Dim sngLabComponent As Single = 0
Dim sngTotal As Single = 0
Dim i As Integer

' Calculate the exam component
For i = 0 To aryExamGrades.Length - 1
    sngExamComponent += CSng(aryExamGrades(i).Text) / CSng(aryExamGradesMax(i).Text)
Next
sngExamComponent = sngExamComponent / aryExamGrades.Length
sngExamComponent = sngExamComponent * CSng(Me.txtExamPercent.Text)

' Calculate the lab component
For i = 0 To aryLabGrades.Length - 1
    sngLabComponent += CSng(aryLabGrades(i).Text) / CSng(aryLabGradesMax(i).Text)
Next
sngLabComponent = sngLabComponent / aryLabGrades.Length
sngLabComponent = sngLabComponent * CSng(Me.txtLabPercent.Text)

' Calculate the assignments component
For i = 0 To aryAssignmentsGrades.Length - 1
    sngAssignmentComponent += CSng(aryAssignmentsGrades(i).Text) / CSng(aryAssignmentsGradesMax(i).Text)
Next
sngAssignmentComponent = sngAssignmentComponent / aryAssignmentsGrades.Length
sngAssignmentComponent = sngAssignmentComponent * CSng(Me.txtAssignmentsPercent.Text)

' Calculate overall by adding together each component
sngTotal = sngAssignmentComponent + sngLabComponent + sngExamComponent
Me.lblGrade.Text = CStr(sngTotal)
```

Here is a screenshot of the application in action. If we had the grades entered and blew off the final and last assignment (grade of 0) we’d still end up with 60%, a D overall.
The Sentence Verification Experiment

As another example of using arrays, let’s implement a program that runs the sentence verification experiment.

In 1969, cognitive psychologists Collins and Quillian proposed that human memory is organized into a "semantic network." Concepts are represented in the network as nodes. There are links or associations that connect a particular node with other concept nodes. A sample network is shown below:

![Semantic Network Diagram]

To answer questions in the semantic network, Collins and Quillian proposed the notion of spreading activation.

Depending on the input (say, reading or words that you hear), the corresponding nodes in the network are activated. This starts a chain reaction where neighboring nodes to the activated nodes are activated with a tag indicating the source. This process repeats, until a collision or intersection occurs from two different sources at the same node. At this point, information is evaluated to validate any activated tags.

Example: “Is a canary an animal?” The CANARY node and the ANIMAL node are activated. Next, nodes that are neighbors to ANIMAL and CANARY are also activated, and a collision occurs at BIRD. We then evaluate that a canary is an animal and produce a YES answer. Notice that ANIMAL would have also propagated to FISH and CANARY to IS YELLOW but no collision occurs there.

Example: “Is a canary a fruit?” Activation spreads from both CANARY and FRUIT (not shown in the diagram), but there is no intersection except at a high level concept of “object” or “living entity”. Consequently a 'NO' answer is returned.
Collins and Quillian proposed that human memory is actually organized according to the structure described above. To test their hypothesis, we can devise an experiment: The Sentence Verification Experiment. In this experiment we ask a human subject to answer simple questions like, "Is a canary a bird?" as fast as they can and time how long it takes the subject to answer. If human memory does conform to the semantic network model then we should see the following results:

It should take longer for people to answer "Is a canary a bird?" than to answer "Is a canary an organism?". The reason it should take longer is because spreading activation has longer to travel for the semantically distant concepts (canary and organism) than the semantically close concepts (canary and bird).

It should take longer for people to answer false questions like "Is a canary a fruit?" than true questions like "Is a canary a bird?" Once again, the false questions require a farther distance for spreading activation, so they should take longer to answer.

Here is the form design for our program. It will ask questions one at a time. To give the subject some time to position his or her fingers over the “Y” or “N” keys, a button will be clicked and then the subject will have 5 seconds before the question is displayed. Once the subject hits a key then the time it took to answer the question is displayed in a listbox.
I’ve also added a timer to the form which will be used for the 5 second countdown.

Next we need some variables at the class level:

```vba
Dim countDown As Integer ' Use to make a countdown from 5 to 0
Dim startTime As Date ' Used to compute time it takes for
' the user to answer
Dim questions(5) As String ' Questions to ask the user
Dim current As Integer = 0 ' Which question out of the array
' that we are asking
```

We can hard-code the questions in the Form_Load event:

```vba
questions(0) = "Is a petunia a mammal?"
questions(1) = "Is a horse a vegetable?"
questions(2) = "Is a canary a reptile?"
questions(3) = "Is a robin an organism?"
questions(4) = "Is a bee an insect?"
questions(5) = "Is a robin a bird?"
```

Next, let’s implement the piece where the user can click the button, a countdown timer
starts, and when then timer hits 0 the current question is displayed.

```vba
Private Sub btnStart_Click(. . .) Handles btnStart.Click
    timerCountdown.Interval = 5000 ' 5 seconds
    timerCountdown.Enabled = True
    lblCountdown.Visible = True
    lblCountdown.Text = "Get your fingers ready! Question will appear in five seconds."
End Sub
```

After five seconds have elapsed the timer’s Tick() method is invoked, so let’s fill it in so we display the current question:

```vba
Private Sub timerCountdown_Tick(. . .) Handles timerCountdown.Tick
    ' Turn off the timer
    timerCountdown.Enabled = False
    ' Hide the label that shows the countdown message
    lblCountdown.Visible = False
    ' Show the question
    lblQuestion.Visible = True
    lblQuestion.Text = questions(current)
    ' Start timing how long it takes to answer
    startTime = Now
End Sub
```

At this point one question will be displayed after the button is clicked and five seconds
elapse. We are using the startTime variable to store the time (down to milliseconds) that the question was displayed. “Now” refers to the current time on the computer’s clock. This will be used to calculate how long it takes the subject to press a key.
Next, we need to be able to process a keypress. Since focus is on the button (the user had to click it to get this started) we check for the keypress event on the button:

```vbnet
Private Sub btnStart_KeyPress(...) Handles btnStart.KeyPress
    ' Compute time it took to answer
    Dim endTime As Date = Now
    Dim elapsedTime As TimeSpan = endTime.Subtract(startTime)
    Dim DifferenceInMilliSeconds As Integer = CInt(elapsedTime.TotalMilliseconds)
    lstResults.Items.Add("Question: " & questions(current) & 
        " You answered: " & 
        CStr(e.KeyChar) & " In " & DifferenceInMilliSeconds & " ms.")
    btnStart.Text = "Show Next Question"
    ' Hide the question
    lblQuestion.Visible = False
    current += 1
    If current = questions.Length Then
        MessageBox.Show("The end! The questions will now repeat")
        current = 0
    End If
End Sub
```

The first few lines read the current time on the computer’s clock and calculates the difference in milliseconds from startTime to endTime. This is displayed in the listbox and then current is incremented so the next question is shown on the form.
**Sorting**

Sorting an array is the process of re-ordering the items so they are in alphabetical or numerical order. It turns out that VB.NET has a built-in sort routine:

```
Array.Sort(arrayname)
```

For example, the following would sort an array of 10 integers initially scrambled:

```vbnet
dim a() as integer = {5, 4, 8, 7, 10, 9, 1, 3, 2, 6}
dim i as integer
array.sort(a)
for i = 0 to 9
    console.WriteLine(a(i))
next
```

The output of the program would be:

```
1
2
3
4
5
6
7
8
9
10
```

This is pretty handy, except sometimes we might need to make our own sorting routine. This can happen if we have multiple arrays that are linked by index. For example, say that we have an array of foods and another array of how many calories are in that food. We might want to sort the foods alphabetically.

Before sorting each array independently:

```
aryFoods(0) = “cheeseburger”   aryCalories(0)=610
aryFoods(1) = “onion rings”   aryCalories(1)=125
aryFoods(2) = “fries”    aryCalories(2)=360
```

After sorting each array independently:

```
aryFoods(0) = “cheeseburger”   aryCalories(0)=125
aryFoods(1) = “fries”    aryCalories(1)=360
aryFoods(2) = “onion rings”   aryCalories(2)=610
```
We now have the wrong calories associated with each food. What we would really like to do is sort on the foods only, but carry along the calories with the food whenever we rearrange the order.

There are a couple of ways to do this; one is to write our own sort routine. There are hundreds of sort routines that have been invented. Let’s look at a simple one called selection sort.

The idea behind selection sort is somewhat how you might sort a hand of cards. First, scan through all of the cards until the lowest card is found. Move it all the way to the left. Then scan through all of the remaining cards until the lowest card is found. Move it immediately to the right of the leftmost card. This means the first two cards are now sorted. Repeat the process for all cards:

Example:

```
5 2 4 6 1 3
1 5 2 4 6 3
1 2 5 4 6 3
1 2 3 5 4 6
1 2 3 4 5 6
1 2 3 4 5 6
```

In the computer it is a bit of work to actually insert a card in the middle of the hand, because it means in the array we must slide all the other values over by one position. So instead we will just swap the card to insert with the target position.

Here is pseudocode for selection sort:

```
SelectionSort(array A)
    For i = 0 to A.Length - 1
        minIndex=FindIndexOfMin(i, A)
        Swap A(minIndex) with  A(i)
```

Now we can flesh out some VB.NET code. First here is sample code that would invoke our SelectionSort subroutine:

```
Dim a() As Integer = {5, 4, 8, 7, 10, 9, 1, 3, 2, 6}
Dim i As Integer
```
Here is the code for selection sort:

```vbnet
Public Sub SelectionSort(ByVal ary() As Integer)
    Dim i As Integer
    Dim minIndex As Integer

    For i = 0 To ary.Length - 1
        minIndex = FindIndexOfMin(i, ary)
        ' Swap A(minIndex) with A(i)
        Dim temp As Integer = ary(minIndex)
        ary(minIndex) = ary(i)
        ary(i) = temp
    Next
End Sub

Public Function FindIndexOfMin(ByVal i As Integer, ByVal ary() As Integer) As Integer
    Dim minIndex As Integer = i
    Dim minValue As Integer = ary(i)

    i += 1
    Do While (i < ary.Length) And (ary(i) < minValue)
        minValue = ary(i)
        minIndex = i
    End If
    i += 1
    Loop
    Return minIndex
End Function
```

The output is the same as before.

A couple of issues to note regarding this sorting routine: first, there are two nested loops, each running approximately N times, where N is the length of the array. As a result, this algorithm runs in time proportional to N^2. Consequently, if we had to sort an array of 1000 values, we would be processing 1,000,000 numbers. This is inefficient for large arrays. There are other techniques that can sort in N*\log_2 N time, which is considerably more efficient. One bonus for this sorting technique though is the amount of space that it uses. Aside from a few temporary variables, we don’t need to allocate any additional memory – this is called an in-place sorting algorithm.

Finally, here is a modification of the sorting algorithm that we can use for our Calorie program. All we really need to do is change the routine so it is sorting the aryFoods array, which is of type String instead of Integer. Then whenever we make a swap, we should swap both the aryFoods values and the aryCalories values:
Private Sub Button1_Click(...) Handles Button1.Click
    Dim aryFoods(2) As String
    Dim aryCalories(2) As Integer

    aryFoods(0) = "cheeseburger"
    aryCalories(0) = 610
    aryFoods(1) = "onion rings"
    aryCalories(1) = 125
    aryFoods(2) = "fries"
    aryCalories(2) = 360

    Dim i As Integer
    SelectionSort(aryFoods, aryCalories)
    For i = 0 To aryFoods.Length - 1
        Console.WriteLine(aryFoods(i) & " " & aryCalories(i))
    Next
End Sub

Public Sub SelectionSort(ByVal aryFoods() As String, ByVal aryCalories() As Integer)
    Dim i As Integer
    Dim minIndex As Integer

    For i = 0 To aryFoods.Length - 1
        minIndex = FindIndexOfMin(i, aryFoods)
        ' Swap A(minIndex) with A(i)
        Dim temp As Integer = aryCalories(minIndex)
        aryCalories(minIndex) = aryCalories(i)
        aryCalories(i) = temp
        Dim tempStr As String = aryFoods(minIndex)
        aryFoods(minIndex) = aryFoods(i)
        aryFoods(i) = tempStr
    Next
End Sub

Public Function FindIndexOfMin(ByVal i As Integer, ByVal aryFoods() As String) As Integer
    Dim minIndex As Integer = i
    Dim minValue As String = aryFoods(i)

    i += 1
    Do While (i < aryFoods.Length)
        If (aryFoods(i) < minValue) Then
            minValue = aryFoods(i)
            minIndex = i
        End If
        i += 1
    Loop
    Return minIndex
End Function