# Using Arrays in Visual Basic

The variables used so far can store only one value at a time. These types of variables are called **simple** or **scalar** variables. Often programs need to store many values all of the same data type. For example, a program may need to store the hours worked for each of the employees in a department, like the ones listed below.

Employee Name Hours Worked

Joe 40

Rick 20

Lana 40

Yi 35

Ellen 45

If *hoursWorked* is defined as a variable it can hold only one value at a time. Of course, we could define five variables (e.g., *hoursWorked1*, *hoursWorked2*, etc.) to store all of values in the list but this becomes cumbersome to handle, especially when the list increases in size.

## Array declarations

An **array** variable is a simpler way to hold a list of values. All of the values in an array have the same data type and can be referred to by the same array name. The declaration of an array is similar to the declaration of a simple variable except that the array variable name must be followed by a number in parentheses. This number indicates how many values are to be stored in the array (i.e. how many memory locations should be reserved). The declaration below declares an array variable called *hoursAry* that can store the list of hours shown above.

Dim hoursAry(5) As Integer

*hoursAry* is the name of an array that can be used to store five integer values. The values are stored in sequence in memory. Each value in an array is called an element. The elements in an array are numbered beginning with 1 for the first element, 2 for the second element and so on. To identify a particular element of an array you use the array variable name followed by the number of an element in parentheses. For example, *hoursAry (1)* refers to the first element in the *hoursAry*. The value in parentheses is called a subscript. We can visualize the array *hoursAry*, where each box represents a memory location that can hold one integer value, as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

hoursAry(1) hoursAry(2) hoursAry(3) hoursAry(4) hoursAry(5)

To assign a value to the first two elements in the *hoursAry*, use the statements:

hoursAry(1) = 40

hoursAry(2) = 20

(Actually, Visual Basic allocates an additional element with an element number of zero each time you declare an array. For the most part in this class we will ignore the zero element and assume that numbering begins with one.)

## Using subscripts to access array elements

The code below declares a simple variable *J* that can store one integer value and an array variable called *Prices* that can store five values of type *Double*.

Dim J As Integer

Dim Prices (5) As Double

Below are some examples showing ways to assign values to array elements and display the values. Notice that the subscript (in parentheses) is always an integer value which can be an integer constant (like 2), an integer variable (like J), or a numeric expression of type integer (like J + 1).

Prices (2) = 11.99 ' Assigns 11.99 to array element 2

J = 4

Prices (J) = 3.50 ' Assigns 3.50 to array element 4 since the value of the ' variable *J* is 4

Prices (3) = Prices (2) - 5 ' Assigns value of 6.99 (11.99 - 5) to element 3

Prices (J + 1) = 4.59 ' Assigns the value 4.59 to array element 5

Prices (1) = InputBox("Enter a price ") ' Assigns value entered by user to array element 1

MessageBox.Show(Prices (2)) ' Prints 11.99, the value of array element 2

The result of the above assignment statements and *InputBox* statement would be as follows assuming that the user entered 8.99 in the *InputBox* statement:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 |  | 8.99 | 11.99 | 6.99 | 3.50 | 4.59 |

J Prices (1) Prices (2) Prices (3) Prices (4) Prices (5)

## Processing arrays

In the previous example the values were assigned to the array elements at random. However, many times a program needs to access all of the elements of the array in order. For this reason, a counting loop, such as a *For/Next* loop is essential. The loop counter variable can be used as the subscript for the array elements to access each element in the array sequentially. For example, the following code retrieves each array element and displays its value in a message box on the screen.

For J = 1 To 5

MessageBox.Show(Prices (J))

Next

*J* is used as a counter in the *For* loop and as a subscript. As *J* increases by one each time through the loop, the next element in the array is referenced.

A counting loop is also useful when the array element values need to be added together to find the total of all of the values. A declaration is shown for an additional variable to store the total. It is initialized to zero before the loop begins.

Dim total As Double

total = 0

For J = 1 To 5

total = total + Prices (J)

Next

MessageBox.Show("The total is " & total)

Again *J* is used both as a loop counter and a subscript. The first time through the loop, when *J* is 1, *Prices (1)* is added to total so total has a value of 8.99. The second time through the loop when *J* is 2, *Prices (2)* is added to total so total now has a value of 20.98 and so on. The final value displayed for total, assuming the values for the *Prices* elements assigned previously, would be 36.06.

Assume, that you want to find the smallest value in the array of *Prices*. In the code below it is initially assumed that the first element in the array contains the smallest value so this value is assigned to the variable smallest. Then, looping sequentially through the array, the value of each element, starting with element 2, is compared to smallest. If an element with a smaller value is located, that element's value becomes the new value of smallest. The code below will display 3.50 as the smallest value.

Dim smallest As Double

smallest = Prices (1)

For J = 2 To 5

If Prices (J) < smallest Then

smallest = Prices (J)

End If

Next

MessageBox.Show("The smallest value is " & smallest)

## When to use an array

Arrays should be used when data needs to be stored in memory to be processed again later. Although they are often needed and are very useful when data needs to be processed more than once, large arrays do take up a great deal of memory and add more complexity to a program so they should be used only as necessary. For example, to read in a list of numbers and find the average, you do not need to use arrays. However, to print out the values that are larger than the average, you need to save all of the original values in an array. Then, after finding the average, each element's value can be compared to the average to see if it is larger. The following program demonstrates this process. The program reads in five exam scores, saving them in an array and calculates the average score. It then displays each score which is above this average value. Note that we are not using the *Scores(0)* element in the array.

Sub aboveAverageScores()

Dim totalScores As Integer

Dim tverageScore As Double

Dim N As Integer

Dim Scores(5) As Integer

' Read in five test scores

For N = 1 To 5

Scores(N) = InputBox("Enter score for student " & N)

Next

' Calculate and display the average score

totalScores = 0

For N = 1 To 5

totalScores = totalScores + Scores(N)

Next

averageScore = totalScores / 5

MessageBox.Show("The average score is " & averageScore)

' Display all scores higher than the average

For N = 1 To 5

If (Scores(N) > averageScore) Then

MessageBox.Show("Score for Student " & N & " of " & \_

Scores(N) & " is above the average.")

End If

Next

End Sub

The values stored in the Scores array elements are shown below, assuming the input values are: 78, 85, 92, 65, 77

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 78 | 85 | 92 | 65 | 77 |

Scores(1) Scores(2) Scores(3) Scores(4) Scores(5)

The output displayed in message boxes when this program is executed will be:

The average score is 79.4

Score for Student 2 of 85 is above the average

Score for Student 3 of 92 is above the average

## Using Arrays as Arguments to Procedures

An array is a list of values of a particular data type (e.g. *Integer* or *Double* or *String*). You can write a Sub procedure that takes an entire array as an argument. To pass an entire array to a Sub procedure:

• In the Sub procedure call statement use the name of the array without () or a subscript. For example: *IncreasePrices(Prices)*

• In the Sub procedure header the corresponding parameter is specified like other parameters except that its name is followed by parenthesis. For example:

*Sub IncreasePrices(ByRef Prices() As Double)*

Arrays are typically passed *ByRef* as passing *ByVal* means that all elements of the array must be copied to the corresponding array parameter in the procedure, a time-consuming process for large arrays.

This example uses a Sub procedure to add 1 to each element of any integer array.

Const NUM\_ELE As Integer = 10

Sub MainSub()

Dim Ary(numEle) As Integer

. . . ' Code to fill array elements with values

IncrementArray(Ary, NUM\_ELE) ' Sub procedure call

End Sub

Sub IncrementArray(ByRef Ary() As Integer, ByVal NUM\_ELE As Integer)

Dim J As Integer

For J = 1 To NUM\_ELE

Ary(J) = Ary(J) + 1

Next

End Sub

Sometimes an array is not filled to capacity and, in that case, the number of elements of the array actually in use can be passed as an argument. The example to increment each element in the array by 1 can be rewritten to add only the first *eleFilled* elements of the array:

Const NUM\_ELE As Integer = 10

Sub MainSub()

Dim Ary(NUM\_ELE) As Integer, eleFilled As Integer

. . . ' Code to fill array with values and set

' number of elements filled (eleFilled)

IncrementArray(Ary, NUM\_ELE, eleFilled) ' Sub procedure call

End Sub

Sub IncrementArray(ByRef Ary() As Integer, ByVal eleFilled As Integer)

Dim J As Integer

For J = 1 To eleFilled

Ary(J) = Ary(J) + 1

Next

End Sub

**Parallel Arrays**

The sample program below reads in monthly sales figures and prints out the name of the month in which the largest sales are found along with the largest sales amount. The array containing month names is initialized with assignment statements. The monthly sales amounts are entered from the keyboard (via *InputBox*) and stored in elements of the array *MonthTotal*. The arrays *MonthName* and *MonthTotal* are called **parallel arrays** since it is assumed that data in the corresponding elements of these two arrays is related. For example, "January" is stored in element 1 of the *MonthName* array and the sales for January will be stored in element 1 of the *MonthTotal* array, "February" is stored in element 2 of the *MonthName* and the sales for February will be stored in element 2 of the MonthTotal array, etc. For purposes of illustration, separate loops are used for input and for finding the largest value.

Sub FindLargestSales()

Dim MonthTotal(12) As Double, MonthName(12) As String

Dim J As Integer

Dim largeSales as double, largeMonthSub as Integer

' Begin by initializing the month names

MonthName(1) = "January"

MonthName(2) = "February"

MonthName(3) = "March"

MonthName(4) = "April"

MonthName(5) = "May"

MonthName(6) = "June"

MonthName(7) = "July"

MonthName(8) = "August"

MonthName(9) = "September"

MonthName(10) = "October"

MonthName(11) = "November"

MonthName(12) = "December"

**' Read in the monthly sales.**

For J = 1 to 12

MonthTotal (J) = InputBox("Enter sales for " \_

& " the month of " & MonthName(J))

Next

**' Find the month with the largest sales.**

largeSales = 0

largeMonthSub = 0

For J = 1 to 12

If (MonthTotal (J) > largeSales) Then

largeSales = MonthTotal (J)

largeMonthSub = J

End If

Next

**' The largeMonth variable now contains the subscript**

**' corresponding to the largest monthly sales value.**

**' Use this subscript to select the name of the month**

**' to print.**

MessageBox.Show("The largest sales were " & largeSales \_

& Chr(13) & " in the month of " & \_

MonthName(largeMonthSub)

End Sub

## Filling An Array From A File

The following program reads data values from a file and stores them in the elements of an array. First the main procedure ***btnFillArray*** calls the procedure ***FillArrayFromFile*** to fill the array. It then prints out information to demonstrate that the data read from the file was correctly stored in the array.

Private Sub btnFillArray\_Click(…)Handles btnFillArray.Click

Const MAX\_ITEMS As Integer = 25

Dim NumArray(maxItems) As Integer

Dim numItems As Integer

FillArrayFromFile(NumArray, MAX\_ITEMS, numItems)

MessageBox.Show("NumItems = " & numItems & Chr(13) & \_

"First Item = " & NumArray(1) & Chr(13) & \_

"Last Item = " & NumArray(numItems))

End Sub

Sub FillArrayFromFile(ByRef NumArray() As Integer, ByVal MAX\_ITEMS As

Integer, ByRef numItems As Integer)

Dim numFile As StreamReader = IO.File.OpenText("NumData.txt")

numItems = 0

Do Until EndOfStream And numItems < MAX\_ITEMS

numItems = numItems + 1

NumArray(numItems) = numFile.ReadLine

Loop

numFile.Close()

End Sub

The array *NumArray* is passed to the ***FillArrayFromFile*** procedure. The procedure includes statements toopen the data file and read in each data item, saving each value read into the next available element in the array. Then it passes back the filled array, along with a count of the number of data values inserted into the array (*numItems***)**. The *MessageBox.Show* in the main procedure prints the number of array elements filled, the value stored in the first array element and the value stored in the array element that contains the last value read in from the data file.

Note that the constant MAX\_ITEMS is used to dimension *NumArray*. No more than 25 values (the number of elements in the array) may be stored in the array. The condition *numItems* < MAX\_ITEMS in the *Do While* loop ensures that this limit is not exceeded.

## Searching Arrays

The following code is the main procedure for a program to search an array. Three parallel arrays are defined to hold information about products sold by a supermarket including the product's bar code, price, and name. Assume that these arrays have already been filled by reading values from an input file. Also assume that *productCount* indicates how many of the items in the table have been filled in. (Thus, *productCount* will never be larger than the array size of 500.) The main procedure asks the user for a bar code, calls a function procedure to search the *ProductCode* array for a matching bar code and then displays the price and the name of the product if it has been found.

Private Sub btnSearchArray\_Click (…) Handles btnSearchArray.Click

Const SIZE As Integer = 500

Dim ProductBarCode(SIZE) As Integer

Dim ProductPrice(SIZE) As Integer

Dim ProductName(SIZE) As String

Dim productCount As Integer

Dim position As Integer

Dim barCode As Integer

' Assume a Sub procedure called here to fill the arrays with data

barCode = InputBox(“Enter a product code number”)

position = FindProduct(ProductBarCode, ProductCount, barCode)

If position > 0 Then

MessageBox.Show("Product number: " & barCode & Chr(13) & \_

"Product name: " & ProductName(position) & Chr(13) & \_

"Product price: " & ProductPrice(position))

Else

MessageBox.Show("Item ” & barCode & “ not found in table.")

End If

The *FindProduct* function looks through the *ProductBarCode* array to find the one that matches the desired item. It then returns the position of that item in the table (i.e. the subscript). If the bar code is not in the table, it returns zero.

### **Unordered Data: Linear Search**

If the product bar codes are in no specific order, it’s necessary to start searching from the beginning of the array. We keep searching until the item is found. If we reach the end of the table and still haven’t found a matching bar code, we return a value of -1 to indicate that the bar code wasn’t found. This type of search is called a linear search and is implemented by the following function:

Function FindProduct (ByRef ProductBarCode() As Integer,

ByVal productCount As Integer, ByVal barCode As Integer) As Integer

Dim k As Integer

Dim position As Integer

position = -1

For k = 1 To productCount

If ProductCode(k) = barCode Then

position = k

End If

Next

Return position

End Function

|  |  |
| --- | --- |
| ProductBarCode[1] | 123 |
| ProductBarCode[2] | 144 |
| ProductBarCode[3] | 189 |
| ProductBarCode[4] | 213 |
| ProductBarCode[5] | 258 |
| ProductBarCode[6] | 297 |
| ProductBarCode[7] | 302 |
| ProductBarCode[8] | 324 |
| ProductBarCode[9] | 368 |
| ProductBarCode[10] | 416 |
| ProductBarCode[11] | 435 |
| ProductBarCode[12] | 441 |
| ProductBarCode[13] | 487 |
| ProductBarCode[14] | 506 |
| ProductBarCode[15] | 524 |
| ProductBarCode[16] | 547 |
| ProductBarCode[17] | 599 |
| ProductBarCode[18] | 611 |
| ProductBarCode[19] | 625 |
| ProductBarCode[20] | 678 |
| ProductBarCode[21] | 687 |

## 

## Sorting Arrays

Many computer programs require data to be sorted in a specific order. A grading program might sort grades from highest to lowest. An inventory program might sort the product ID numbers from lowest ID to highest ID. Names in a list are usually sorted in alphabetical order. Once data is saved in an array, the array can be sorted using a variety of techniques.

The following program reads in a list of integer scores in arbitrary order and stores them in an array. The values in the array are then sorted in ascending order with the smallest value in the first element of the array, the next smallest value in the second element of the array and so on.

The main program shown below stores values in the array by calling the *readScores* procedure presented in class. It calls a procedure *sortScores*, shown on the next page, to sort the array values into ascending order.

Private Sub btnScores\_Click(…) Handles btnScores.Click

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

' Purpose: This program reads in scores from a data file and stores

' them in an array. It displays the values in the array in a listbox

' before and after sorting the values in ascending order.

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Const MAX\_SIZE As Integer = 10

Dim scores(MAX\_SIZE) As Integer

' Read in scores and store in array

readScores(scores, MAX\_SIZE)

' Print scores in array to list box

lstScores.Items.Add("")

lstScores.Items.Add("Unsorted Scores array")

PrintScores(scores, MAX\_SIZE)

' Sort scores in ascending order (lowest to highest)

sortScores(Scores, N)

' Print scores again after they have been sorted

lstScores.Items.Add("")

lstScores.Items.Add("Sorted Scores array")

printScores(scores, MAX\_SIZE)

End Sub

The *sortScores* procedure shown below uses the bubblesort algorithm, one of the many different algorithms available to sort an array. Although the bubblesort algorithm is not the most efficient sorting algorithm it is one of the simplest.

Sub sortScores(ByRef scores() As Integer, ByVal MAX\_SIZE As Integer)

Dim k As Integer

Dim sorted As String

Dim temp As Integer

Dim numPairs as integer

sorted = "N"

numPairs = MAX\_SIZE - 1

Do Until sorted = "Y" ' Outer loop: continue until all values

sorted = "Y" ' are in sorted order

For k = 1 To numPairs ' Inner loop: compare pairs,

If scores(k) > scores(k + 1) Then ' exchanging if out of order

temp = scores(k)

scores(k) = scores(k + 1)

scores(k + 1) = temp

sorted = "N"

End If

Next

numPairs = numPairs - 1

Loop

End Sub

This procedure sorts the array by making several passes over the data. On the first pass through the array, every pair of adjacent array elements (like *scores(1)* and *scores(2)*) is compared. If these two values are in the correct order (i.e., the If condition is False), they are left alone. If they are out of order (i.e., the If condition is True), the two values are swapped. After going through the entire array once and swapping values if a pair is out of order (the inner loop), the largest value will be in the last position of the array. It is no longer necessary to look at that item. After making a second pass through the array, the largest two items will be in the last two positions of the array. Thus, each time we go through the array, we need to examine one less pair of array elements. The value of *numPairs* tells how many pairs of array elements still need to be examined.

If we ever go through the array without switching any values at all, then the array is sorted and the loop can terminate. That’s the purpose of the variable *sorted* in this program. If a pair of values is out of order, the values are switched and **sorted** is set to "N". Then we need to go through the array once again. If *sorted* is "Y" because no switches had to be made, the array is completely sorted so the outer loop ends.

Once the array is in sorted order, searching the array can be done more efficiently by using a binary search rather than a linear search.