Arrays

(Arrays – Part 1)

Array: An ordered group of related data items having a common variable name, all of the same data type.

QBasic permits us to deal with many related data items as a group by means of a structure known as an array. Today we'll look at ways arrays can be used in a situation such as a television poll problem, in which groups of data items must be stored and manipulated efficiently.

Subscripts/Indices

Element: An individual data item within an array.

Subscript/Index: A value used to identify the position of a particular array element; it should be enclosed in parentheses after the array name.

Subscripted/Indexed variable: A variable that refers to a specific element within an array.

Unsubscripted/Non-Indexed variable: A variable that refers to a single, independent storage location.

Individual data items within an array are called elements. An array consists of a group of consecutive/contiguous storage locations, each location contains a single element. The entire array is given one name, and the programmer indicates an individual element within the array by referring to its position in the array. The same rules that apply to naming simple variables also apply to naming arrays.

A subscript/index is a value enclosed in parentheses that identifies the position of a given element in the array. This subscript/index does not have to be an integer constant; it can consist of any valid numeric expression.

Dimensioning an Array

The `DIM` statement allows the programmer to declare an array and reserve the appropriate amount of memory for the desired size. The `DIM` statement must appear in your program before the first reference to the array it dimensions; a good practice is to place all `DIM` statements at the beginning of your program. When a subscripted/indexed variable is found in a program, QBasic recognized it as part of an array and automatically accesses its storage location for manipulation.

```
DIM nums (1 to 10) AS INTEGER
```

This statement line declares an **`array`** named `nums` as an `INTEGER` **`array`** with ten elements (1 thru 10).

One-Dimensional Arrays

One-Dimensional array: An array that contains a single list of elements (one row of elements).

A major advantage of using arrays is the ability to use a variable rather than a constant as a subscript. Because a single expression such as `tests(count)` can refer to any element in the array named `tests`, depending on

the value of the variable named `count`, this subscripted variable name can be used in a loop that varies the value of the subscript `count`.

A `FOR…NEXT` loop can provide an efficient method of placing data into an array if the exact number of elements to be entered is known in advance.

```
DIM tests (1 to 5) AS SINGLE
DIM count AS INTEGER

FOR count = 1 TO 5
      INPUT "Enter test score: ", tests(count)
NEXT count
```

Printing the Contents of an Array.

The `FOR…NEXT` loop is also a good way to print the contents of an array. The following snippet could be added to the previous program to output the contents of all three arrays:

```
CLS

FOR ctr1 = 1 TO numStud
    PRINT sName(ctr1); age(ctr1); SSN(ctr1)
NEXT ctr1
```

Performing Calculations on Array Elements

Following is a program that calculates average television viewing time, based on the viewing habits of 4 individual viewers. Each viewer's difference from the average is calculated. The results are then displayed.

```
CLS

DIM viewer(1 TO 4) AS STRING
DIM hours(1 TO 4)  AS INTEGER
DIM difference(1 TO 4)  AS INTEGER
DIM totalHours AS INTEGER
DIM average AS SINGLE
DIM ctr1, ctr2, ctr3 AS INTEGER

totalHours = 0
PRINT

FOR ctr1 = 1 TO 4
    INPUT "Enter a person's name: ", viewer(ctr1)
    PRINT "Enter "; viewer(ctr1); "'s total viewing hours: ";
    INPUT "", hours(ctr1)
    totalHours = totalHours + hours(ctr1)
NEXT ctr1

average = totalHours / 4

FOR ctr2 = 1 TO 4
    difference(ctr2) = hours(ctr2) - average
NEXT ctr2
```

```
CLS
PRINT
PRINT " Name"; TAB(17); "Hours"; TAB(24); "Difference from Average"
FOR ctr3 = 1 TO 4
    PRINT viewer(ctr3); TAB(18); hours(ctr3);
    PRINT TAB(35); difference(ctr3)
NEXT ctr3

PRINT
PRINT TAB(5); "Average Viewing Time = "; average


END
```

Two-Dimensional Arrays

Two-Dimensional array: An array in which elements are arranged in both rows and columns.

The arrays shown so far have all been one-dimensional arrays; that is, arrays that store values in the form of a single list. Two-dimensional arrays enable a programmer to represent more complex groupings of data. For example, suppose that a souvenir shop is running a four-day promotion on T-shirts at its three locations. It might keep the following table of data concerning the number of shirts sold by each of the three locations:

		Store		
		1	**2**	**3**
Day	**1**	12	14	15
	2	10	16	12
	3	11	18	13
	4	9	9	10

Each row of the data refers to a specific day of the sale, and each column contains the sales data for one store. Thus, the number of shirts sold by the second store on the third day of the sale (18) can be found in the third row, second column. Data items that can be grouped into rows and columns such as this can be stored easily in a two-dimensional array. A two-dimensional array named `tShirts` that contains the preceding data could be pictured like this:

Array tShirts

12	14	15
10	16	12
11	18	13
9	9	10

Array `tShirts` consists of 12 elements arranged as four rows and three columns. To reference a single element of a two-dimensional array such as this, two subscripts are needed: one to indicate the row and a second to indicate the column. For instance, the subscripted variable `tShirts (4,1)` contains the number of

shirts sold on the fourth day by the first store (9). The first subscript gives the row number and the second subscript gives the column number.

The rules regarding one-dimensional arrays also apply to two-dimensional arrays. Two-dimensional arrays are named in the same way as other variables and cannot use the same name as another array (of any dimensions) in the same program. A two-dimensional array can contain only one type of data; numeric and character string values cannot be mixed. As with one-dimensional arrays, subscripts of two-dimensional arrays can be indicated by any legal numeric expression:

```
tShirts (3, 3)
tShirts (i, 2)
tShirts (i, j)
tShirts(1, ctr + 6)
```

Let's assume that `ctr1 = 4` and `ctr2 = 2`, and that the array `numbers` contains the following 16 elements:

Array numbers

10	15	20	25
50	55	60	65
90	85	100	105
130	135	140	145

The following examples show how the various forms of subscripts are used:

Example	Refers To
numbers(4, ctr1)	`numbers (4, 4)` – the element in the fourth row, fourth column of `numbers`, which is 145.
numbers(ctr2, ctr1)	`numbers (2, 4)` – the element is in the second row, fourth column of `numbers`, which is 65.
numbers(3, ctr2 + 1)	`numbers (3, 3)` – the element is in the third row, third column of `numbers`, which is 100.
numbers(ctr1 – 1, ctr2 – 1)	`numbers (3, 1)` – the element is in the third row, first column of `numbers`, which is 90.

As with one-dimensional arrays, the dimensions of two-dimensional arrays should be set with a `DIM` statement. For example, the array with up to 15 rows and five columns:

```
DIM studentName (1 to 15, 1 to 5) AS STRING
```

This array can hold a maximum of 75 (15 X 5) elements.

Printing Two-Dimensional Arrays

As you may recall from earlier, a `FOR…NEXT` loop is a convenient means of accessing all the elements of a one-dimensional array. `FOR…NEXT` loops can also be used to place data into a two-dimensional array as a group of one-dimensional arrays, with each row making up a single one-dimensional array. A single `FOR…NEXT` loop can

be used to store values in one row. This process is repeated for as many rows as the array contains; therefore, the FOR…NEXT loop that stores data in a single row is nested within a second FOR…NEXT loop controlling the number of rows being accessed.

The array tShirts of the previous example can be filled from the sales data table one row at a time, moving from left to right across the columns. The following program snipped shows the nested FOR---NEXT loops that do this:

```
FOR row = 1 TO 4
    FOR column = 1 TO 3
        PRINT "Enter sales for store"; column; "on day"; row; ": ";
        INPUT "", tShirts(row, column)
    NEXT column
NEXT row
```

Each time the INPUT statement is executed, one value is placed in a single element of the array; the element is determined by the current values of the variables row and column. This statement is executed 12 (4 X 3) times, which is the number of elements in the array. The outer loop (with the loop control variable row) controls the rows, and the loop control variable column controls the columns. Each time the outer loop is executed once, the inner loop is executed three times. While row = 1, column becomes 1, 2, and finally 3 as the inner loop is executed. Therefore, if the user enters, 12, 14, and 15 for the first three sales, these values are stored in tShirts(1, 1), tShirts(1, 2), and tShirts(1, 3), and the first row is filled:

	column =		
	1	**2**	**3**
row = 1	12	14	15

While row equals 2, column again varies from 1 to 3, and values entered are placed into tShirts(2, 1), tShirts(2, 2), and tShirts(2, 3) to fill the second row. Assuming the next three values entered are 10, 16, and 12, the array would now look like this:

	column =		
	1	**2**	**3**
	12	14	15
row = 2	10	16	12

row is incremented to 3 and then to 4, and the third and fourth rows are filled in the same manner.

To print the contents of the entire array, the programmer can substitute a PRINT statement for the INPUT statement in the nested FOR…NEXT loops. The following snippet prints the contents of the array tShirts, one row at a time:

```
DIM blank AS INTEGER
blank = 10

FOR row = 1 TO 4
    FOR column = 1 TO 3
        PRINT TAB(blank * column); tShirts(row, column);
    NEXT column
    PRINT
NEXT row
```

The semicolon at the end of the `PRINT` statement tells QBasic to print three values on the same line. After the inner loop is executed, a blank `PRINT` statement causes a carriage return so that the next row is printed on the next line.

The following program shows how sales data can be entered into this array, totaled, and then printed in table form with appropriate headings. Notice that as the sales data is entered, the following statement keeps track of the total sales:

```
totalShirts = totalShirts + tShirts(row, column)
```

the value of `totalShirts` is then printed at the bottom of the report.

array **tShirts**

12	14	15
10	16	12
11	18	13
9	9	10

```
CLS

DIM tShirts(1 TO 4, 1 TO 3) AS INTEGER
DIM totalShirts, row, column AS INTEGER

totalShirts = 0

FOR row = 1 TO 4 STEP 1
    PRINT
    FOR column = 1 TO 3 STEP 1
        PRINT "Enter sales for store"; column; "on day"; row;
        INPUT "", tShirts(row, column)
        totalShirts = totalShirts + tShirts(row, column)
    NEXT column
NEXT row

CLS

PRINT TAB(8); "T - Shirt Sales Report"
PRINT
PRINT "Day #"; TAB(10); "Store 1"; TAB(20); "Store 2";
PRINT TAB(30); "Store 3"

FOR row = 1 TO 4 STEP 1
```

```
        PRINT row;
        FOR column = 1 TO 3 STEP 1
            PRINT TAB(column * 10); tShirts(row, column);
        NEXT column
        PRINT
    NEXT row

    PRINT "Total T-Shirt sales for all stores: "; totalShirts

    END
```

Adding values stored in Rows

Once data has been stored in an array, it is often necessary to manipulate certain array elements. For instance, the sales manager in charge of the T-shirt promotional sale might want to know how many shirts were sold on the last day of the sale.

Because the data for each day is contained in a row of the array, it is necessary to total the elements in one row of the array (the fourth row) to find the number of shirts sold on the fourth day. The fourth row can be thought of by itself as a one-dimensional array. One loop is therefore required to access all the elements of this row:

```
day4Sales = 0
      FOR column  = 1 TO 3 STEP 1
            day4Sales = day4Sales + tShirts(4, column)
      NEXT column
```

Notice that the first subscript of `tShirts(4, column)` restricts the computations to the elements in row `4`, while column varies from `1 TO 3`.

Adding values stored in Columns

To find the total number of T-shirts sold by the third store, it is necessary to total the elements in the third column of the array. This time we can think of the column by itself as a one-dimensional array of for elements. This operation calls for a `FOR…NEXT` loop, as shown here:

```
store3 = 0
FOR row = 1 TO 4 STEP 1
      store3 = store3 + tShirts(row, 3)
NEXT row
```

Parallel Arrays

Parallel arrays: Arrays in which the values in corresponding element numbers are related to one another. An example might be and individual's name, age, and SSN.

It's often possible to enter data into several arrays within a single loop. In the following program the user enters three values, each of which is assigned to a different array:

```
CLS

DIM numStud AS INTEGER

INPUT "How many students would you like to enter"; numStud

DIM sName(1 TO numStud) AS STRING
DIM SSN(1 TO numStud) AS STRING
DIM age(1 TO numStud) AS INTEGER
DIM ctr1 AS INTEGER

FOR ctr1 = 1 TO numStud
    INPUT "Enter the student's name: ", sName(ctr1)
    INPUT "Enter the student's age: ", age(ctr1)
    INPUT "Enter the student's SSN: ", SSN(ctr1)
NEXT ctr1
END
```