Unit 1 - Activity 5 - Array Reading

4.1 Arrays

Arrays are a convenient way of representing groups of data. Consider the months of the year. We could store them in a program as follows:

```
|  MONTH1$ = "JAN"   |
|  MONTH2$ = "FEB"   |
|  MONTH3$ = "MAR"   |
|  MONTH4$ = "APR"   |
|  ...               |
```

Then when we want to print month X we could code:

```
|   IF X = 1 THEN PRINT MONTH1$  |
|   IF X = 2 THEN PRINT MONTH2$  |
|   IF X = 3 THEN PRINT MONTH3$  |
|   IF X = 4 THEN PRINT MONTH4$  |
|   ...                          |
```

This isn't terribly efficient. And what if, instead of 12 items, we had hundreds or THOUSANDS? We might want to get a total of all incomes in a particular group. If there were just a few people to total we could code

```
| TOTAL.INCOME = INCOME1 + INCOME2 + INCOME3...  |
```

But do you really want to write the equation for 10,000 people? Luckily we have arrays. An array is a grouping of data that still lets us access individual elements. An array is defined by a dimension statement:

```
| DIM MONTH$(12)  |
```

This defines a group of 13 strings starting with MONTH$(0) and ending with MONTH$(12). What's in each one is still up to the programmer and the program, but now we can access a particular item much easier. Using the months example from above, we still need to define each item or element of the array :

```
|   MONTH$(1) = "JAN"  |
|   MONTH$(2) = "FEB"  |
|   MONTH$(3) = "MAR"  |
|   MONTH$(4) = "APR"  |
|   ....               |
```

So far no big change, but look at how easy it is to find a particular value. Now we can get month X directly:

```
|   PRINT month$(X)    |
```

X is called an index. If we wanted to print all twelve months, we could

use a loop:

```
|   FOR X = 1 to 12    |
|   PRINT MONTH$(X)    |
|   NEXT               |
```

Compare this to the hassle of trying to print all months the first way. When you have larger arrays, the savings become spectacular. These arrays are called singly dimensioned arrays since there is only one index. But we can also think of times we'd like to use several dimensions. Maybe we want to track the production of several product lines over several months. We could set

```
|   DIM PRODUCT(10, 12) |
```

This defines an array that will hold sales information on 10 products for each of 12 months. Thus PRODUCT(5, 11) would hold the sales for the 5th product for the 11th month. Note, we could just as easily defined this as

```
|   DIM PRODUCT(12, 10)  |
```

where we have data for each month for each product. Now the data for the 5th product for the 11th month would be PRODUCT(11, 5). The first method is the one usually preferred. Think of the array as starting with the larger category (here, product type) on the left, moving to subcategories on the right (months).

We can extend the number of dimensions to 3 if we want to show the sales for each day of the month:

```
|   DIM PRODUCT(10, 12, 31)  |
```

In theory you can have 255 dimensions. In reality you'd run out of memory well before using all 255. Each added dimension will raise the required storage by at least a factor of 2. The total memory space that is available to Basic is only 64k. Thus even in our products example, we move from the 11 elements of PRODUCT(10) to the 143 of PRODUCT(10,12) to 4576 for PRODUCT(10,12,31) (11*13*32). Another problem is the fact that most people have trouble conceptualizing more than 3 or 4 dimensions. Usually it's easier to restate the problem. In years of programming I can recall only a few instances where more than 3 dimensions made any practical sense.

Arrays form the basis of most data processing applications, especially in areas like spreadsheets and statistics. In Basic, an array is considered to start from item 0. Many programmers forget about this element and though it will take up a little more space, you often can ignore it too. But there are some cases where it comes in handy. Suppose we have a 5 by 5 array and want to get totals in each direction. Using our product by month example, we'd want to get totals for each product for all months and totals for each month for all products. Without arrays we'd have to construct separate assignment statements for each total. (No, this won't be assigned as an exercise. But

just think how long it would take to do this, and how many places you could mistype and cause an error!) The short program total.bas shows this:

```
| 10  ' totals.bas                                |
| 20  ' do cross totals on an array               |
| 30  DIM X(5,5)                                  |
| 40  X(0,0) = 0                ' grand totals    |
| 50  FOR I = 1 TO 5                              |
| 60  X(I,0) = 0                                  |
| 70     FOR J = 1 TO 5                           |
| 80     X(I,J) = RND(1)*10                       |
| 90     X(I,0) = X(I,0) + X(I,J)  ' line totals  |
| 100    X(0,J) = X(0,J) + X(I,J)  ' column totals|
| 110    X(0,0) = X(0,0) + X(I,J)  ' grand total  |
| 120    NEXT                                     |
| 130 NEXT                                        |
| 140 FOR I = 5 TO 0 STEP -1                      |
| 150    IF I = 0 THEN PRINT "  "   ' extra space |
| 160    FOR J = 5 TO 0 STEP - 1                  |
| 170    IF J = 0 THEN PRINT "  ";  ' extra row   |
| 180    PRINT USING " ###.## ";X(I,J);           |
| 190    NEXT                                     |
| 200    PRINT                                    |
| 210 NEXT                                        |
```

First we define an array X to be 5 by 5. (Later you can expand this to more rows and columns, but you may run out of space to display it on the screen.) Since we're only going to use the elements that have indices greater than 1, we have 3 classes of elements that would otherwise be wasted. These are all X(i, 0), X(0, j) and the single element X(0,0). We'll use these as follows:

```
|   X(i, 0) will store the total for row i                  |
|   X(0, j) will store the total for column j               |
|   X(0, 0) will store the grand total of all rows and columns |
```

Line 40 sets the grand total to 0. Now we have a double-do loop (not to be confused with the DOO-WAH loop that's often used for timing.) Since this is just a test, we don't want to burden the user by having them enter all the data, so in line 80 we just fill in a random number from 0 to 9. Then we accumulate the line and column totals in lines 90-110. That's all there is to it. To display the results we'll use a pair of reversed loops. This will present the 0 indices in the last rows and columns, so the table will look more like the spreadsheet format you may be used to. In the lower right corner is X(0,0) the grand total.

The PRINT USING statement " ###.##" says to print the value of X using 8 spaces. The value will be shown as up to 3 digits with 2 additional places shown after the decimal place.

Sum and product of an array

```
10 REM Create an array with some test DATA in it
20 DIM A(5)
30 FOR I = 1 TO 5: READ A(I): NEXT I
40 DATA 1, 2, 3, 4, 5
50 REM Find the sum of elements in the array
60 S = 0
65 P = 1
70 FOR I = 1 TO 5
72 S = SUM + A(I)
75 P = P * A(I)
77 NEXT I
80 PRINT "The sum is "; S;
90 PRINT " and the product is "; P
```