CHAPTER 8

ADVANCED DATA HANDLING

- READ and DATA
- Averages
- Subscripted Variables
  - One-Dimensional Arrays
  - Averages Revisited
- DIMENSION
- Simulated Dice Roll With Arrays
- Two-Dimensional Arrays

READ AND DATA

You've seen how to assign values to variables directly within the program (A = 2), and how to assign different values while the program is running—through the INPUT statement.

There are many times, though, when neither one of these ways will quite fit the job you're trying to do, especially if it involves a lot of information.

Try this short program:

```
10 READ X
20 PRINT  "X IS NOW : "; X
30 GOTO 10
40 DATA 1, 34, 10.5, 16, 234.56

RUN

X IS NOW : 1
X IS NOW : 34
X IS NOW : 10.5
X IS NOW : 16
X IS NOW : 234.56

?OUT OF DATA ERROR IN 10
READY
```

In line 10, the computer READs one value from the DATA statement and assigns that value to X. Each time through the loop the next value in the DATA statement is read and that value assigned to X, and PRINTed. A pointer in the computer itself keeps track of which value is to be used next:

```
          Pointer
            ↓
40 DATA 1, 34, 10.5, 16, 234.56
```

When all the values have been used, and the computer executed the loop again, looking for another value, the OUT OF DATA error was displayed because there were no more values to READ.

It is important to follow the format of the DATA statement precisely:

```
4Ø DATA 1, 34, 1Ø.5, 16, 234.56
        ↑                  ↑
   Comma separates      No Comma
   each item
```

Data statements can contain integer numbers, real numbers (234.65), or numbers expressed in scientific notation. But you can't READ other variables, or have arithmetic operations in DATA lines. This would be incorrect:

```
4Ø DATA A, 23/56, 2*5
```

You can, however, use a string variable in a READ statement and then place string information in the DATA line. The following is acceptable:

```
NEW

10 FOR X = 1 to 3
15 READ A$
20 PRINT "A$ IS NOW : "; A$
30 NEXT
40 DATA THIS, IS, FUN

RUN

A$ IS NOW : THIS
A$ IS NOW : IS
A$ IS NOW : FUN
READY
```

Notice that this time, the READ statement was placed inside a FOR . . . NEXT loop. This loop was then executed to match the number of values in the data statement.

In many cases you will change the number of values in the DATA statement each time the program is run. A way to avoid counting the number of values and still avoid an OUT OF DATA ERROR is to place a "FLAG" as the last value in the DATA line. This would be a value that your data would never equal, such as a negative number or a very large or small number. When that value is READ the program will branch to the next part.

There is a way to reuse the same DATA later in the program by RE-

STOREing the data pointer to the beginning of the data list. Add line 50 to the previous program:

```
5Ø GOTO 1Ø
```

You will still get the OUT OF DATA error because as the program branches back to line 10 to reread the data, the data pointer indicates all the data has been used. Now, add:

```
45 RESTORE
```

and RUN the program again. The data pointer has been RESTOREd and the data can be READ continuously.

AVERAGES

The following program illustrates a practical use of READ and DATA, by reading in a set of numbers and calculating their average.

```
NEW

5 T = 0 : CT = 0
10 READ X
20 IF X = -1 THEN 50: REM CHECK FOR FLAG
25 CT = CT + 1
30 T = T + X : REM UPDATE TOTAL
40 GOTO 10
50 PRINT "THERE WERE "; CT;"VALUES READ"
60 PRINT "TOTAL = ";T
70 PRINT "AVERAGE ="; T/CT
80 DATA 75, 80, 62, 91, 87, 93, 78, -1

RUN
THERE WERE 7 VALUES READ
TOTAL = 566
AVERAGE = 80.8571429
```

Line 5 sets CT, the CounTer, and T, Total, equal to zero. Line 10 READs a value and assigns the value to X. Line 20 checks to see if the value is our flag (here a −1). If the value READ is part of the valid DATA, CT is incremented by 1 and X is added to the total.

When the flag is READ, the program branches to line 50 which PRINTs

the number of values read. Line 60 PRINTs the total, and line 70 divides the total by the number of values to get the average.

By using a flag at the end of the DATA, you can place any number of values in DATA statements—which may stretch over several lines—without worrying about counting the number of values entered.

Another variation of the READ statement involves assigning information from the same DATA line to different variables. This information can even be a mixture of string data and numeric values. You can do all this in the following program that will READ a name, some scores—say bowling—and print the name, scores, and the average score:

```
NEW

10 READ N$,A,B,C
20 PRINT N$;"'S SCORES WERE: ";A;" ";B;" ";C
30 PRINT "AND THE AVERAGE IS: ";(A+B+C)/3
40 PRINT: GOTO 10
50 DATA MIKE, 190, 185, 165, DICK, 225, 245, 190
60 DATA JOHN, 155, 185, 205, PAUL, 160, 179, 187

RUN

MIKE'S SCORES WERE: 190  185  165
AND THE AVERAGE IS : 180

DICK'S SCORES WERE: 225  245  190
AND THE AVERAGE IS : 220
```

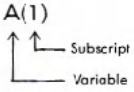
In running the program, the DATA statements were set up in the same order that the READ statement expected the information: a name (a string), then three values. In other words N$ the first time through gets the DATA "MIKE", A in the READ corresponds to 190 in the data statement, "B" to 185 and "C" to 165. The process is then repeated in that order for the remainder of the information. (Dick and his scores, John and his scores, and Paul and his scores.)

SUBSCRIPTED VARIABLES

In the past we've used only simple BASIC variables, such as A, A$, and NU to represent values. These were a single letter followed by a

letter or single digit. In any of the programs that you would write, it is doubtful that we would have a need for more variable names than possible with all the combinations of letters or numbers available. But you are limited in the way variables are used with programs.

Now let's introduce the concept of subscripted variables.



This would be said: A sub 1. A subscripted variable consists of a letter followed by a subscript enclosed within parentheses. Please note the difference between A, A1, and A(1). Each is unique. Only A(1) is a subscripted variable.

Subscripted variables, like simple variables, name a memory location within the computer. Think of subscripted variables as boxes to store information, just like simple variables:

A(0)	
A(1)	
A(2)	
A(3)	
A(4)	

If you wrote:

```
10 A(0) = 25: A(3) = 55 : A(4) = -45.3
```

Then memory would look like this:

A(0)	25
A(1)	
A(2)	
A(3)	55
A(4)	−45.3

This group of subscripted variables is also called an array. In this case, a one-dimensional array. Later on, we'll introduce multidimensional arrays.

Subscripts can also be more complex to include other variables, or computations. The following are valid subscripted variables:

A(X) A(X+1) A(2+1) A(1*3)

The expressions within the parentheses are evaluated according to the same rules for arithmetic operations outlined in Chapter 2.

Now that the ground rules are in place, how can subscripted variables be put to use? One way is to store a list of numbers entered with INPUT or READ statements.

Let's use subscripted variables to do the averages a different way.

```
5 PRINT CHR$(147)
10 INPUT "HOW MANY NUMBERS :";X
20 FOR A = 1 TO X
30 PRINT "ENTER VALUE # ";A;:INPUT B(A)
40 NEXT
50 SU = 0
60 FOR A = 1 TO X
70 SU = SU + B(A)
80 NEXT
90 PRINT : PRINT "AVERAGE = "; SU/X


RUN

HOW MANY NUMBERS :? 5
ENTER VALUE # 1  ? 125
ENTER VALUE # 2  ? 167
ENTER VALUE # 3  ? 189
ENTER VALUE # 4  ? 167
ENTER VALUE # 5  ? 158

AVERAGE =  161.2
```

There might have been an easier way to accomplish what we did in this program, but it illustrates how subscripted variables work. Line 10 asks for how many numbers will be entered. This variable, X, acts as the counter for the loop within which values are entered and assigned to the subscripted variable, B.

Each time through the INPUT loop, A is increased by 1 and so the next value entered is assigned to the next element in the array A. For example, the first time through the loop A = 1, so the first value entered is assigned to B(1). The next time through, A = 2; the next value is assigned to B(2), and so on until all the values have been entered.

But now a big difference comes into play. Once all the values have been entered, they are stored in the array, ready to be put to work in a variety of ways. Before, you kept a running total each time through the

INPUT or READ loop, but never could get back the individual pieces of data without re-reading the information.

In lines 50 through 80, another loop has been designed to add up the various elements of the array and then display the average. This separate part of the program shows that all of the values are stored and can be accessed as needed.

To prove that all of the individual values are actually stored separately in an array, type the following immediately after running the previous program:

```
FOR A = 1 TO 5 : ?B(A),: NEXT

125         167         189         167
158
```

The display will show your actual values as the contents of the array are PRINTed.

DIMENSION

If you tried to enter more than 10 numbers in the previous example, you got a DIMENSION ERROR. Arrays of up to eleven elements (subscripts 0 to 10 for a one-dimensional array) may be used where needed, just as simple variables can be used anywhere within a program. Arrays of more than eleven elements need to be "declared" in a dimension statement.

Add this line to the program:

```
5 DIM B(1ØØ)
```

This lets the computer know that you will have a maximum of 100 elements in the array.

The dimension statement may also be used with a variable, so the following line could replace line 5 (don't forget to eliminate line 5):

```
15 DIM B(X)
```

This would dimension the array with the exact number of values that will be entered.

Be careful, though. Once dimensioned, an array cannot be redimensioned in another part of the program. You can, however, have multiple arrays within the program and dimension them all on the same line, like this:

```
1Ø DIM C(2Ø), D(5Ø), E(4Ø)
```

SIMULATED DICE ROLL WITH ARRAYS

As programs become more complex, using subscripted variables will cut down on the number of statements needed, and make the program simpler to write.

A single subscripted variable can be used, for example, to keep track of the number of times a particular face turns up:

```
1 REM DICE SIMULATION : PRINT CHR$(147)
10 INPUT "HOW MANY ROLLS:";X
20 FOR L = 1 TO X
30 R = INT(6*RND(1))+1
40 F(R) = F(R) + 1
50 NEXT L
60 PRINT "FACE", "NUMBER OF TIMES"
70 FOR C = 1 TO 6 : PRINT C, F(C): NEXT
```

The array F, for FACE, will be used to keep track of how many times a particular face turns up. For example, every time a 2 is thrown, F(2) is increased by one. By using the same element of the array to hold the actual number on the face that is thrown, we've eliminated the need for five other variables (one for each face) and numerous statements to check and see what number is thrown.

Line 10 asks for how many rolls you want to simulate.

Line 20 establishes the loop to perform the random roll and increment the proper element of the array by one each for each toss.

After all of the required tosses are completed, line 60 PRINTs the heading and line 70 PRINTs the number of times each face shows up.

A sample run might look like this:

```
HOW MANY ROLLS: ? 1000
FACE                NUMBER OF TIMES
1                   148
2                   176
3                   178
4                   166
5                   163
6                   169
```

Well, at least it wasn't loaded!

Just as a comparison, the following is one way of re-writing the same program, but without using subscripted variables. Don't bother to type it in, but do notice the additional statements necessary.

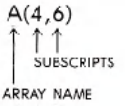
```
10 INPUT "HOW MANY ROLLS:";X
20 FOR L = 1 TO X
30 R = INT(6*RND(1))+1
40 IF R = 1 THEN F1 = F1 + 1 : NEXT
41 IF R = 2 THEN F2 = F2 + 1 : NEXT
42 IF R = 3 THEN F3 = F3 + 1 : NEXT
43 IF R = 4 THEN F4 = F4 + 1 : NEXT
44 IF R = 5 THEN F5 = F5 + 1 : NEXT
45 IF R = 6 THEN F6 = F6 + 1 : NEXT
60 PRINT "FACE", "NUMBER OF TIMES"
70 PRINT 1, F1
71 PRINT 2, F2
72 PRINT 3, F3
73 PRINT 4, F4
74 PRINT 5, F5
75 PRINT 6, F6
```

The program has doubled in size from 8 to 16 lines. In larger programs the space savings from using subscripted variables will be even more dramatic.

TWO-DIMENSIONAL ARRAYS

Earlier in this chapter you experimented with one-dimensional arrays. This type of array was visualized as a group of consecutive boxes within memory each holding an element of the array. What would you expect a two-dimensional array to look like?

First, a two-dimensional array would be written like this:



and could be represented as a two-dimensional grid within memory:

	Ø	1	2	3	4	5	6
Ø							
1							
2							
3							
4							

The subscripts could be thought of as representing the row and column within the table where the particular element of the array is stored.

A(3,4) = 255

↑ ↑ COLUMN

ROW

	0	1	2	3	4	5	6
0							
1							
2							
3					255		
4							

If we assigned the value 255 to A(3,4), then 255 could be thought of as being placed in the 4th column of the 3rd row within the table.

Two-dimensional arrays behave according to the same rules that were established for one-dimensional arrays:

They must be dimensioned:	DIM A(20,20)
Assignment of data:	A(1,1) = 255
Assign values to other variables:	AB = A(1,1)
PRINT values:	PRINT A(1,1)

If two-dimensional arrays work like their smaller counterparts, what additional capabilities will the expanded arrays handle?

Try this: can you think of a way using a two-dimensional array to tabulate the results of a questionnaire for your club that involved four questions and had up to three responses for each question? The problem could be represented like this:

CLUB QUESTIONNAIRE

Q1: ARE YOU IN FAVOR OF RESOLUTION #1?

☐1-YES ☐2-NO ☐3-UNDECIDED

. . . and so on.

The array table for this problem could be represented like this:

	RESPONSES		
	YES	NO	UNDECIDED
QUESTION 1			
QUESTION 2			
QUESTION 3			
QUESTION 4			

The program to do the actual tabulation for the questionnaire might look like that shown on page 103.

This program makes use of many of the programming techniques that have been presented so far. Even if you don't have any need for the actual program right now, see if you can follow how the program works.

The heart of this program is a 4 by 3 two-dimensional array, A(4,3). The total responses for each possible answer to each question are held in the appropriate element of the array. For the sake of simplicity, we don't use the first rows and column (A(0,0) to A(0,4)). Remember, though, that those elements are always present in any array you design.

In practice, if question one is answered YES, then A(1,1) is incremented by one—row 1 for question 1 and column 1 for a YES response. The rest of the questions and answers follow the same pattern. A NO response for question three would add one to element A(3,2), and so on.

```
                SHIFT
20 PRINT "{CLR/HOME}"
30 FOR R = 1 TO 4
40 PRINT "QUESTION # : "; R
50 PRINT " 1-YES  2-NO  3-UNDECIDED"
60 PRINT "WHAT WAS THE RESPONSE : ";
61 GET C : IF C <1 or C>3 THEN 61
65 PRINT C: PRINT
70 A(R,C) = A(R,C) + 1: REM UPDATE ELEMENT
80 NEXT R
85 PRINT
90 PRINT "DO YOU WANT TO ENTER ANOTHER": PRINT
   "RESPONSE (Y/N)";
100 GET A$ : IF A$ = "" THEN 100
110 IF A$ = "Y" THEN 20
120 IF A$ <> "N" THEN 100
130 PRINT "{CLR/HOME}";"THE TOTAL RESPONSES
    WERE:":PRINT
140 PRINT  SPC(18);"RESPONSE"
141 PRINT "QUESTION","YES","NO","UNDECIDED"
142 PRINT "--------  ------------------------------"
150 FOR R = 1 TO 4
160 PRINT R, A(R,1), A(R,2), A(R,3)
170 NEXT R
RUN

QUESTION # : 1
1-YES  2-NO  3-UNDECIDED
WHAT WAS THE RESPONSE : 1

QUESTION # : 2
1-YES  2-NO  3-UNDECIDED
WHAT WAS THE RESPONSE : 1

And so on...

THE TOTAL RESPONSES WERE:
                          RESPONSE
QUESTION          YES          NO        UNDECIDED

   1              6            1            0
   2              5            2            0
   3              7            0            0
   4              2            4            1
```