

## CHAPTER 6

# USING A REPETITIVE PROCESS

The power of a computer lies in its ability to perform tedious and repetitive processes quickly and to relieve people of such tasks. In fact, this capability can even result in changes to the nature of a solution. For example, when working by hand, you might approximate a solution to a problem; however, working with a computer, you might find you can do the entire set of computations in the same amount of time.

MINC will do repetitive processes if it is programmed to execute certain statements over and over again in a "loop". A *loop* is a sequence of statements that MINC repeats continuously until an end condition is met.

The sines and cosines problem executes some statements over and over and can calculate the sine and cosine of any number of angles. Again, the program is:

```
20 PRINT 'SINE','COSINE'
30 PRINT 'DEGREES';
40 INPUT X
60 Z=X*PI/180
70 PRINT ,SIN(Z),COS(Z)
80 GO TO 30
```

MINC repeats statements 30, 40, 60, 70, and 80 until you type CTRL/C. What is actually happening is:

<i>Statement</i>	<i>What Happens</i>
20	print labels
30	prints prompt

<i>Statement</i>	<i>What Happens</i>
40	inputs angle in degrees
60	converts angle to radians
70	prints sine and cosine
80	returns control to line 30
30	prints prompt
40	inputs angle in degrees
60	.
70	.
80	.
30	

In this program, statements 30 through 80 form a loop.

## LOOPS USING IF STATEMENTS AND GO TOs

A simple program that computes the squares and cubes of positive whole numbers follows:

```
10 PRINT 'I','I^2','I^3'
20 I = 1
30 PRINT I,I^2,I^3
40 I = I + 1
50 GO TO 30
RUNNH
```

I	I <sup>2</sup>	I <sup>3</sup>
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
.	.	.
.	.	.
.	.	.

Statements 30, 40, and 50 are the statements in the loop. This program does not terminate unless you press CTRL/C.

This program is in an *infinite loop*. Infinite loops never terminate by themselves. You must intervene and terminate the loop by pressing CTRL/C twice.

If you want a loop to terminate by itself, you can place a condition in the loop that terminates the loop when the condition is met. This condition is called the *end condition*. For example, this program can be rewritten to print the squares and cubes from 1 to 100 and stop after 100.

```

10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 IF I=100 GO TO 70
50 I=I+1
60 GO TO 30
70 END
RUNNH

```

I	I <sup>2</sup>	I <sup>3</sup>
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
.	.	.
.	.	.
.	.	.
99	9801	970299
100	10000	1.00000E+06

READY

This time the program terminates by itself after 100, 100<sup>2</sup>, and 100<sup>3</sup> are printed.

Take care when programming the end condition of a loop. If you interchange statement 40 (the end test) and statement 50 (which increments the statement variable), then the test would no longer work as before. For example:

```

10 PRINT 'I','I^2','I^3'
20 I=1
30 PRINT I,I^2,I^3
40 I=I+1
50 IF I=100 GO TO 70
60 GO TO 30
70 END

```

The results of this loop are not as defined in the problem and are shown below. When I reaches 98, this is what happens:

<i>Statement</i>	<i>What Happens</i>
30	prints 98,98 <sup>2</sup> ,98 <sup>3</sup>
40	increments I to 99
50	I (99) is not equal to 100 so go to 60
60	go to 30
30	prints 99,99 <sup>2</sup> ,99 <sup>3</sup>
40	increments I to 100

*Statement*

*What Happens*

50

I (100) is equal to 100 so go to 70

70

end the program

The values  $100$ ,  $100^2$ , and  $100^3$  are never printed because the end condition is not correct. In this case, statement 50 must be changed to:

```
50 IF I>100 GO TO 70
```

You should convince yourself (possibly by trying it) that this new line 50 allows  $100$ ,  $100^2$ , and  $100^3$  to be printed.

To present another example of the many ways in which a loop can be written and still perform the same function, the following program is equivalent to the two previous versions of the squares and cubes problem.

```
10 PRINT 'I','I^2','I^3'  
20 I = 1  
30 PRINT I,I^2,I^3  
40 I = I + 1  
50 IF I <= 100 GO TO 30  
60 END
```

Although both the sines and cosines program and the squares and cubes program are loops, the logic of the programs is very different. The squares and cubes problem executes its loop exactly 100 times. The program counts the number of times the loop is executed. The number of times the sines and cosines loop is executed varies with each run and depends on when the user types CTRL/C.

Loops like that in the sines and cosines problem, which are terminated by a specific end condition, can most easily be programmed using IF and GO TO statements. However, loops like that in the squares and cubes problem, which are dependent on a count, can be done more easily with a FOR statement, which automatically counts the number of times the loop is executed.

## FOR LOOPS AND THE FOR/NEXT STATEMENTS

Again, the squares and cubes program is listed below:

```
10 PRINT 'I','I^2','I^3'  
20 I = 1  
30 PRINT I,I^2,I^3  
40 I = I + 1  
50 IF I <= 100 GO TO 30  
60 END
```

Using the FOR statement you can rewrite the program like this:

```
10 PRINT 'I','I'2','I'3'
20 FOR I=1 TO 100
30 PRINT I, I'2, I'3
40 NEXT I
50 END
```

The results of the two versions are the same, but the second is one statement shorter.

The FOR and NEXT statements do the counting automatically. In the second example, the FOR statement:

```
20 FOR I=1 TO 100
```

sets up I as the variable that will count the loop, and I counts from 1 to 100. I is the *control variable* of the FOR loop. The FOR statement starts the top of the loop. The NEXT statement:

```
40 NEXT I
```

determines the bottom of the loop. The NEXT statement increments I by one and returns control to the top of the loop to start the loop over. When I gets incremented to greater than 100, control is passed to the statement following the NEXT (in this case, statement 50, the END statement).

There is only one difference between the way the two examples function, but the difference would not show in the output unless each program were changed slightly as follows.

```
10 PRINT 'I','I'2','I'3'
20 I=1
30 PRINT I,I'2,I'3
40 I=I+1
50 IF I<=100 GO TO 30
55 PRINT I
60 END
```

```
10 PRINT 'I','I'2','I'3'
20 FOR I=1 TO 100
30 PRINT I, I'2, I'3
40 NEXT I
45 PRINT I
50 END
```

In statement 55 of the first example above, MINC prints 101 as the value of I. In statement 45 of the second example above, MINC prints 100 as the value of I. The only difference between

the two loops is the final value of I when MINC exits the loop. The NEXT statement subtracts one from I upon leaving the loop, which leaves the maximum count value (in this case 100) in I.

The general form of the FOR statement is:

FOR control-var = start-value TO end-value STEP step

where:

control-var	is a numeric variable called the control variable.
start-value	is a numeric expression that represents the starting value of the control variable.
end-value	is a numeric expression that represents the ending value of the control variable.
STEP	is optional (and was not used in the previous example), is the amount by which to increment the control variable. If STEP is not present, the increment is 1.
step	is a numeric expression that represents the actual increment.

The general form of the NEXT statement is:

NEXT control-var

where control-var matches the variable in the FOR statement.

The statements between the FOR and NEXT statements are called the body of the loop.

For a more advanced explanation of FOR loops, see the FOR statement in Book 3.

Examples of FOR loops follow.

The following loop shows that you can use a negative step value, which causes the control variable to decrease each time through the loop.

```
20 FOR I = 5 TO 1 STEP -1
30 PRINT I
40 NEXT I
50 END
RUNNH
```

5  
4  
3  
2  
1

READY

The following example shows that any expression within the FOR statement can be fractional.

```
10 FOR J=0 TO 1.5 STEP .3
20 PRINT J
30 NEXT J
40 END
RUNNH
```

0  
.3  
.6  
.9  
1.2  
1.5

READY

The next example shows that you can alter the control variable within the loop; however, you should take care to be sure that this is what you really want to do. (In fact, changing the control variable unintentionally is a major cause of program errors.)

```
10 FOR I = 1 TO 10
20 I = I + 1
30 PRINT I
40 NEXT I
50 END
RUNNH
```

2  
4  
6  
8  
10

READY

## *PROGRAMMING FUNDAMENTALS*

MINC never executes this last loop because I (10) is greater than the end value (1) right from the beginning.

```
10 FOR I = 10 TO 1
20 PRINT I
30 NEXT I
40 END
RUNNH
```

READY



## CHAPTER 7

# ARRAYS AND NESTED LOOPS

An array is a group of conceptually similar variables, all with the same name. For example, suppose a professor has given a 20-question questionnaire to some students and asks you to process the results of this questionnaire using MINC. Your first idea for representing the questions might be to refer to each question on the questionnaire as Q1, Q2, Q3, and so forth. However, this method fails at Q10 because Q10 is not a valid BASIC variable name.

Instead, you can put the questions in an array named Q. You can then reference question 1 by the name Q(1). You can reference question 2 by the name Q(2). And, you can reference question 20 by the name Q(20).

In mathematical notation, these *array elements* are written as follows:

$$Q_0 \quad Q_1 \quad Q_2 \quad Q_3 \quad \dots \quad Q_{19} \quad Q_{20}$$

The small number below the Q is called a *subscript*, and  $Q_2$  is read "Q sub 2."

In MINC notation, the equivalent array elements are written as follows:

$$Q(0) \quad Q(1) \quad Q(2) \quad Q(3) \quad \dots \quad Q(19) \quad Q(20)$$

In MINC, the number in parentheses is also called a subscript, and Q(0) is read "Q sub 0."

The questionnaire problem corresponds very well with the use of arrays. By using an array to represent the questionnaire in MINC, you give all the questions the same variable name, Q. Each individual question is distinguished by its unique subscript.

## CREATING AN ARRAY

In order that MINC can reserve a workspace of sufficient size for your array of variables, you must define how many you will use. In BASIC you can use a dimension statement (DIM) to do this as follows.

```
10 DIM Q(20)
```

This statement tells MINC that you are defining an array named Q with a *dimension* of 20. The dimension determines the number of distinct elements in the array.

If you give an array a dimension of 20, MINC creates 21 distinct elements (elements 0 through 20). All arrays begin with element 0.

The general form of the DIM statement is:

DIM array-list

The argument array-list is the list of arrays to be dimensioned in this statement. An array in the list is written as follows:

array-name(dimension)

The array-name must follow the rules for numeric variable names if the elements in the array are numeric values. The array-name must follow the rules for string variable names if the elements in the array are going to be strings.

The dimension represents the maximum subscript that you can use in the array. The dimension is the number of elements in the array minus one (because of element 0). In the questionnaire example, the dimension is 20 because there are 20 questions in the questionnaire.

The maximum number of elements that you can define in an array depends on the size of the program.

Because the array and the program must both fit in the workspace, a short program leaves more room for a large array, and a large program requires a shorter array.

Remember that with a dimension of 20, there are really 21 elements. For convenience, element 0 will be ignored for now. In later sections, this chapter discusses how to adjust the program to use element 0.

You should use an array for the questionnaire problem because it is very inefficient to name the questions Q1, Q2, and so forth. Even if MINC allowed variable names like Q20 (which it does not), the programming to input all 20 questions would be a problem. You would have to type the following INPUT statement.

```
10 INPUT Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10
11 INPUT Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,Q19,Q20
```

If you wanted to make your program more practical by prompting each question, you would have to type the following 20 lines just to input the answers to one questionnaire:

```
10 PRINT '1'\ INPUT Q1
20 PRINT '2'\ INPUT Q2
30 PRINT '3'\ INPUT Q3
.
.
.
```

By using an array, you can take advantage of FOR loops. Even prompting each question becomes easy. The part of the program to input the answers to one questionnaire is:

```
10 DIM Q(20)
40 FOR I=1 TO 20
50 PRINT I;\ INPUT Q(I)
60 NEXT I
```

Instead of using 20 lines of code, this method uses four. The way this loop works is:

<i>Statement</i>	<i>What Happens</i>
40	starts I at 1
50	prints 1, inputs Q(1) (because I = 1)
60	increments I to 2
50	prints 2, inputs Q(2) (because I = 2)
60	increments I to 3
.	.
.	.
.	.
60	increments I to 20

## WHY USE ARRAYS?

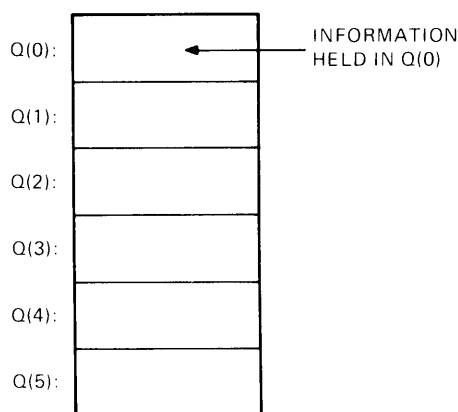
<i>Statement</i>	<i>What Happens</i>
50	prints 20, inputs Q(20) (because I = 20)
60	increments I to 21, exits loop, decrements I to 20

Because you can use arrays so easily in FOR loops, arrays make programming much easier for problems like this one. The computer does the repetition instead of the programmer.

MINC allows up to two subscripts for an array. An array with one subscript is called a *one-dimensional array*. An array with two subscripts is called a *two-dimensional array*. These two kinds of arrays are discussed in the following sections.

## ONE-DIMENSIONAL ARRAYS

So far the method used for storing the results of the questionnaire has been to store the results in a *one-dimensional array*. A one-dimensional array can be pictured as follows. This particular one-dimensional array is named Q.



MR 1587

Figure 7. One-Dimensional Array

This one-dimensional array has six elements (0 through 5).

From the examples in the previous sections, you now know how to get the data from one questionnaire into a one-dimensional array in MINC. Now you have to devise a scheme to process the data.

Suppose the questionnaire has 20 questions, each with three possible responses: agree, disagree, and don't care.

For the time being, all the professor wants is to keep a tally for each question of the number of agrees, disagrees, and don't cares

received. One method is to type in the following input for each question.

Enter a 1 for agree.

Enter a 2 for disagree.

Enter a 3 for don't care.

A program to compute the results of these responses can use an ON/GO TO statement. Besides needing an array for the input of each questionnaire, you need 3 more arrays — one to tally agrees, one to tally disagrees, and one to tally don't cares. Statement 10 becomes:

```
10 DIM Q(20), A(20), D(20), C(20)
```

For example, A(1) tallies the number of people who agreed with question 1. D(15) tallies the number of people who disagreed with question 15. C(12) tallies the number of people who don't care about question 12, and so forth. (Remember that element 0 of each array is being ignored for the time being.)

The program to read and process the questionnaire now becomes:

```
10 DIM Q(20),A(20),D(20),C(20)
20 REM - Q represents the original questionnaire
30 REM - A represents the number of people who agreed
35 REM - with each question
40 REM - D represents the number of people who disagreed
45 REM - with each question
50 REM - C represents the number of people who don't
55 REM - care about each question
60 REM - Input the questionnaire
70 FOR I = 1 TO 20
80 PRINT I; \ INPUT Q(I)
90 NEXT I
100 REM - Process questionnaire
110 FOR I = 1 TO 20
120 ON Q(I) GOTO 150, 160, 170
150 A(I) = A(I) + 1 \ GOTO 200
160 D(I) = D(I) + 1 \ GOTO 200
170 C(I) = C(I) + 1
200 NEXT I
```

In line 120, if Q(I), the response to question number I, is 1, the agree counter is incremented. If Q(I) is 2, the disagree counter is incremented. If Q(I) is 3, the don't care counter is incremented. The ON/GO TO statement ensures that only one counter is incremented for each question.

Because MINC sets all variables to 0 when you type the RUN command, the counters are all started at 0. You do not have to give the counter arrays an initial value of zero.

You can shorten this program segment to input and process a questionnaire. Q is an unnecessary array. You do not need to read in all 20 questions at once when you can process one question at a time. By eliminating the Q array, you can save 19 variables in the workspace. You need only one variable, rather than a whole array of 20 variables.

Without changing the basic algorithm or the approach, you can change the program to reduce the number of variables as follows.

```
10 DIM A(20),D(20),C(20)
30 REM - A represents the number of people who agreed
35 REM - with each question
40 REM - D represents the number of people who disagreed
45 REM - with each question
50 REM - C represents the number of people who don't
55 REM - care about each question
60 REM - Input and process one questionnaire
70 FOR I=1 TO 20
75 REM - R represents one response to one question
80 PRINT I; \ INPUT R
100 REM - Process question
120 ON R GOTO 150, 160, 170
150 A(I)=A(I)+1 \ GOTO 200
160 D(I)=D(I)+1 \ GOTO 200
170 C(I)=C(I)+1
200 NEXT I
```

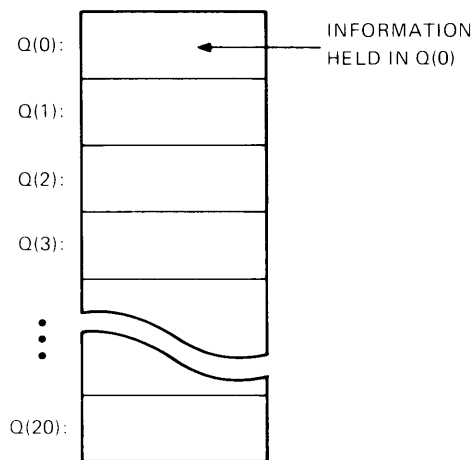
There is another method that uses two-dimensional arrays for inputting and processing a questionnaire. This method is discussed in the next section.

## TWO-DIMENSIONAL ARRAYS

The previous algorithm for processing questionnaires uses three one-dimensional arrays. Again, a one-dimensional array uses one subscript and can be pictured as shown in Figure 8.

A *two-dimensional array* can be thought of as a matrix and requires two subscripts. Two-dimensional arrays can be pictured as shown in Figure 9.

You can use one two-dimensional array to replace the three one-dimensional arrays in the questionnaire problem. Array R,



MR 1588

Figure 8. One-Dimensional Array

INFORMATION HELD IN A(0,0)

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

MR 1589

Figure 9. Two-Dimensional Array

which stands for response, will be dimensioned as follows:

```
10 DIM R(20,3)
```

The first subscript (ranging from 0 through 20) represents the question being answered. The second subscript (1 through 3) represents the response. Therefore, the value of R(2,1) represents the number of people who agreed with question 2. (Remember that all elements with a subscript of 0 are being ignored.)

This method has no fewer variables because three one-dimensional arrays that are 20 elements long have as many variables as one two-dimensional array with 20 x 3 variables. However, the program becomes much shorter. (In fact, this

method is wasting a few more variables by ignoring Q(0,0) through Q(20,0). However, using these elements is discussed in later sections.)

Use of a two-dimensional array simplifies the loop to input and process questionnaires to the following four statements:

```
70 FOR I=1 TO 20
80 PRINT I; \ INPUT Q
90 R(I,Q)=R(I,Q)+1
100 NEXT I
```

The first subscript represents the number of the question; the second subscript represents the code for the answer. In line 90, if I is equal to 12 and Q is equal to 2, then R(12,2) is incremented.

### NESTED LOOPS

A very simple program is now developed for processing one questionnaire. However, the program must be expanded to process many questionnaires. The program needs a loop to process more than one questionnaire; but, you cannot use a FOR loop because you do not know how many questionnaires there are to be processed. Instead, you must create a loop that will work for any number of questionnaires. The following program segment presents a loop for processing many questionnaires.

```
20 PRINT 'Input another questionnaire'; \ INPUT A$
30 IF A$='YES' GO TO 70
40 IF A$='NO' GO TO 350
50 GO TO 20
60 REM - Input and process one questionnaire
70 FOR I=1 TO 20
.
.
.
110 GO TO 20
350 END
```

Many people do not test specifically for a "NO" or invalid response (that is, something other than yes or no) as in lines 40 and 50. Their test would be:

```
50 IF A$<>'YES' GOTO 350
```

However, if you are inputting questionnaire number 199 and you mistype yes for questionnaire 200, you do not want to have to start typing all over again. Thus, it is a good idea to test for the proper input.



The program segment for inputting and processing many questionnaires is shown as follows:

```

10 DIM R(20,3)
20 PRINT 'Input another questionnaire'; \ INPUT A$
30 IF A$ = 'YES' GO TO 70
40 IF A$ = 'NO' GO TO 350
50 GO TO 20
60 REM - Input and process one questionnaire
70 FOR I = 1 TO 20
80 PRINT I; \ INPUT Q
90 R(I,Q) = R(I,Q) + 1
100 NEXT I
110 GO TO 20
350 END

```

This program has one loop (the FOR loop) within another loop (the IF/GO TO loop). This process of putting one loop inside another is called *nesting* loops. You can also nest FOR loops if you need to.

These nested loops work exactly the way this problem was solved. The *inner loop*, which includes statements 70 to 100, is processed first. That is, the program processes all 20 questions of one questionnaire — by repeating statements 70 through 100 twenty times. Once the entire questionnaire is processed, control passes to statement 110, and the outer loop is processed. If there is another questionnaire, control passes back to statement 70 which will again start the inner loop, and the whole process described in this paragraph is repeated. If there is not another questionnaire, the program will stop.

This program only reads and processes the questionnaire. To make the program of any value to the professor, you must produce a report of the results.

The following program produces a report that lists the results.

```

10 DIM R(20,3)
15 REM - Input next questionnaire
20 PRINT 'Input another questionnaire'; \ INPUT A$
30 IF A$ = 'YES' GO TO 70
40 IF A$ = 'NO' GO TO 310
50 GO TO 20
60 REM - Input and process one questionnaire
70 FOR I = 1 TO 20
75 REM - Q represents the response to one question
80 PRINT I; \ INPUT Q
85 REM - Process question
90 R(I,Q) = R(I,Q) + 1
100 NEXT I

```

## PROGRAMMING FUNDAMENTALS

```
110 GO TO 20
300 REM - Print results of all questionnaires
310 PRINT „'AGREE', 'DISAGREE', 'DON'T CARE'
320 FOR I= 1 TO 20
330 PRINT 'Question';I,,R(I,1), R(I,2), R(I,3)
340 NEXT I
350 END
```

Below is a partial sample run of this program. This sample does not show the entire input process or the entire output.

RUN

QUEST                      28-FEB-80                      10:27:00

Input another questionnaire? YES

1? 1

2? 3

3? 1

4? 2

.

.

.

20? 3

Input another questionnaire? NO

	AGREE	DISAGREE	DON'T CARE
Question 1	23	10	7
Question 2	5	29	6
Question 3	18	4	18
Question 4	2	38	0
.	.	.	.
.	.	.	.
.	.	.	.

The above example of the QUEST program now does the entire job of taking questionnaires as input, processing the results, and printing the results. However, this program does not process the questionnaires well. You should be aware of the following problems:

- The program expects an upper case YES as the answer to "Input another questionnaire" and does not recognize a lower case yes.
- The program expects the person typing in the questionnaire responses to be a perfect typist. The program does not validate that the current response is between 1 and 3. In the above example, if the response is other

than 1, 2, or 3, MINC will have a problem as it tries to update  $R(I,Q)$  in line 90. The value of  $Q$  will not be within the dimensions of the array  $R$ . MINC prints the following message:

?MINC-F-Array subscript is negative or too large at line 90

Techniques for resolving these and similar problems are discussed in Chapters 8 and 11.

Suppose for some reason you decided that you want to reset your two-dimensional array  $R$  to zero. That is, you want to set each element to zero later in the program without having to run the program again. You can accomplish this by several means, some easier than others.

### Nested FOR Loops

The easiest way to access each element of a two-dimensional array is to use nested FOR loops (as shown in the following example).

```

300 FOR I = 1 TO 20
301   FOR J = 1 TO 3
302     R(I,J) = 0
303   NEXT J
304 NEXT I
  
```

Nested FOR loops must be completely nested. That is, the inner FOR loop must be entirely within the outer FOR loop as shown by the lines drawn in on the previous example.

Here is an example of an invalid pair of nested FOR loops.

```

200 FOR I = 1 TO 20
201   FOR J = 1 TO 3
202     R(I,J) = 0
203   NEXT I
204 NEXT J
  
```

At statement 204, what is the value of  $I$ ? MINC gets confused at this line, tries to execute the loop, cannot accept this loop, and prints the following message.

?MINC-F-No corresponding FOR statement for NEXT at line 203

Here are some examples of acceptably and unacceptably nested FOR loops.

*Acceptable FOR loops*

```

10 FOR I=1 TO 10
20 FOR I1=1 TO 4
30 FOR I2=2 TO 7
40 NEXT I2
50 NEXT I1
60 NEXT I
    
```

```

10 FOR J=1 TO 10
20 FOR K=2 TO 20
30 NEXT K
40 FOR I=1 TO 12
50 FOR K2=1 TO 5
60 NEXT K2
70 NEXT I
80 NEXT J
    
```

*Unacceptable FOR loops*

```

10 FOR I=2 TO 15
20 FOR K=1 TO 5
30 NEXT I
40 FOR J=1 TO 5
50 NEXT J
60 NEXT K
    
```

```

10 FOR I=1 TO 5
20 FOR K=1 TO 5
30 NEXT K
40 FOR J=1 TO 5
50 NEXT I
60 NEXT J
    
```

## USING ARRAY ELEMENT 0

### One-Dimensional Arrays

The previous examples in this chapter ignored element 0 of each array. The next two sections discuss how you can modify your algorithms to use element 0 for one- and two-dimensional arrays.

Remember that when you dimension a one-dimensional array, it actually has one more element than the value of the dimension implies. For example, the following array has six elements, even though its dimension is 5.

```
10 DIM A(5)
```

In the professor's problem, Q(0), A(0), D(0), and C(0) were ignored, and the FOR loops started at 1. Because the questionnaires started with question 1, not question 0, the programming was made easier by ignoring element 0 in each of the arrays.

A FOR loop to access every element of the six-element array follows:

```
10 FOR I=0 TO 5
20 A(I)=1
30 NEXT I
```

In this FOR loop, every element is given an initial value of 1.

Both dimensions of a two-dimensional array start with zero. For example, the following array:

```
10 DIM A(4,4)
```

actually looks like Figure 9 (Page 99).

The two-dimensional solution to the professor's problem ignored row zero and column zero of the arrays and started the FOR loops at 1. Because the array was dimensioned 20 by 3, the program wasted  $20 + 3 + 1$  array elements, or 24 variables. You really don't want to waste space (that is, declare unused variables) if possible. You can save the space by changing the program:

```
10 DIM R(19,2)
.
.
.
70 FOR I=0 TO 19
80 PRINT I+1; \ INPUT Q
90 R(I,Q-1)=R(I,Q-1)+1
100 NEXT I
.
.
.
```

The array R now has 60 elements. The FOR loop still executes 20 times. Line 80 still prints the right question number;  $0 + 1$  is 1 for the first question.  $Q-1$  converts the agree, disagree, and don't care codes to 0, 1, and 2. You could change the input to 0, 1, and 2, but the programmed conversion is just as simple. Your choice of algorithm depends on how you want to structure your problem. In this case, questionnaires rarely start with question 0, so the program makes the conversion from question 1 to element 0.

This program makes better use of the workspace by not wasting variables. Better yet, the program change is invisible to anyone

## Two-Dimensional Arrays

using the program. That is, the person inputting the questionnaires and printing the results for the professor does not have to learn a new code for the input or have to do anything different, even though the program has changed.

## SUBSCRIPTED VARIABLES

Arrays are a group of variables all given the same name. The individual elements are distinguished by the one or two unique subscripts. Each element then is called a *subscripted variable*.

### Subscripts

The subscripts of a variable determine the unique array element being referenced. Picture a one-dimensional array that looks like Figure 10.

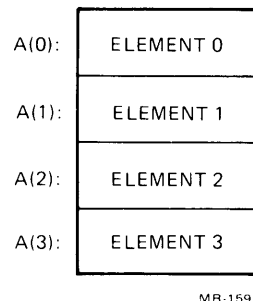


Figure 10. One-Dimensional Array

Array A has four unique elements. The first element is specifically referenced by the name A(0). The fourth element is specifically referenced by the name A(3). As you can see in Figure 10, each array element is a discrete unit, and referencing A(3.7) does not really make any sense. Therefore, before MINC references an array element, it truncates any fractional part of all subscripts to make them whole numbers. Thus, in the following statements, reference A(3), the value of I remains 3.7, but in statement 20, A(3), not A(3.7), gets the value 7.

```
10 I=3.7
20 A(I)=7
```

A subscript can be any numeric expression. For example, A(I+3\*B^2) is a perfectly valid reference to a subscripted variable (as long as I+3\*B^2 is within the dimension of the array). B(Q+7,I\*2.65+9) is a valid reference to a subscripted variable for the two-dimensional array B. If either or both subscripts are not whole numbers, MINC truncates each subscript to a whole number before accessing the array element.

Subscripts can never be string literals or string variables. Subscripts must always be numeric.

Subscripts must remain within the *bounds* or *dimensions* of the array. For example, if you dimension the following array:

```
10 DIM C(8)
```

MINC will not let you reference C(-5) or C(9), nor will it let you use any subscript that is not between 0 and 8. Should you try to reference elements that are out of bounds, MINC will issue the following error message and terminate your program.

```
?MINC-F-Array subscript is negative or too large
```

When using a two-dimensional array, you must take care that both dimensions are within the bounds set in the DIM statement. For example, if you dimension the following array:

```
10 DIM C1(3,5)
```

the first subscript must be between 0 and 3, and the second subscript must be between 0 and 5. C1(4,3) is an invalid reference because the first subscript is too large. C1(1,6) is an invalid reference because the second subscript is too large. C1(-1,8) is an invalid reference because both subscripts are out of bounds.

If the subscripts of a two-dimensional array are out of bounds, MINC will terminate the program with the following message.

```
?MINC-F-Array subscript is negative or too large
```

## Example 1

## Examples

This example counts the number of times each number from 0 to 10 is entered by the person running the program (you, if you try this example).

```
NEW COUNT
10 DIM A(10)
20 FOR I=1 TO 5
30 PRINT 'Input a number from 0 to 10';\ INPUT N
40 A(N)=A(N)+1
50 NEXT I
60 PRINT
100 FOR I=0 TO 10
110 PRINT 'The number';I;'appeared';A(I);'time(s).'
```

```
120 NEXT I
```

```
130 END
```

```
RUN
```

COUNT

01-MAR-80

15:59:55

Input a number from 0 to 10? 5  
 Input a number from 0 to 10? 3  
 Input a number from 0 to 10? 1  
 Input a number from 0 to 10? 0  
 Input a number from 0 to 10? 5

The number 0 appeared 1 time(s).  
 The number 1 appeared 1 time(s).  
 The number 2 appeared 0 time(s).  
 The number 3 appeared 1 time(s).  
 The number 4 appeared 0 time(s).  
 The number 5 appeared 2 time(s).  
 The number 6 appeared 0 time(s).  
 The number 7 appeared 0 time(s).  
 The number 8 appeared 0 time(s).  
 The number 9 appeared 0 time(s).  
 The number 10 appeared 0 time(s).

READY

There are no problems with this example. However, when MINC displays READY, if you type RUN and enter the numbers 1 and 11, MINC will halt the program. In line 40, MINC tries to increment A(11), cannot find A(11), and terminates the program.

### *Example 2*

The following short program written by a history professor tallies votes for the Moderate and Reform parties in all elections between 1900 and 1915. As input, it takes the year and the party — M for Moderate and R for Reform.

```

10 DIM E(15,1)
20 PRINT 'Enter year,party';\ INPUT Y,P$
30 REM - validate party
35 IF P$<>'r' THEN IF P$<>'m' GO TO 20
40 IF P$='m' THEN P=0
50 IF P$='r' THEN P=1
60 REM - validate year
70 IF Y<1900 GO TO 20
80 IF Y>1915 GO TO 20
85 REM - update tally
90 E(Y-1900,P)=E(Y-1900,P)+1
100 REM - next input
110 PRINT 'more data';\ INPUT R$
120 IF R$='yes' GO TO 20
130 FOR I=0 TO 15
140 PRINT 'year';I+1900;'moderate votes';E(I,0);'reform votes';E(I,1)
150 NEXT I
160 END
    
```



This program checks the input data to ensure that each piece of data is correct before the program tries to access an array element. If the data are not correct, the program reissues the prompt and waits for valid data.

### Example 3

In the following example, the subscript goes outside the bounds of the array, but the error is not obvious.

```
10 DIM A(10)
20 FOR I=0 TO 10 STEP 2
30 PRINT A(I),A(I+1)
40 NEXT I
50 END
```

In line 30, when  $I=10$ ,  $I+1=11$ .  $A(11)$  is out of the bounds of the array  $A$ .

Subscripted variables are array elements, or variables, that are referred to by the use of subscripts. Although the subscripts themselves must always be numeric, the variables can hold either numeric or string data (depending on the type of array).

### Subscripted Variables

Subscripted string variables can be used anywhere that string variables can be used. Subscripted numeric variables can be used anywhere that numeric variables can be used except as the control variable for a FOR loop.

The following example uses a string array. This program accepts as input students' names and their three test scores. The program prints out the students' names, test scores, and final grade based on the test scores.

```
10 DIM S$(25),G1(25),G2(25),G3(25),G$(25)
20 PRINT 'Input student name and three test scores';
25 INPUT S$(I),G1(I),G2(I),G3(I)
30 F = G1(I) + G2(I) + G3(I)
40 REM - compute student's letter grade based on average
45 REM - of the three test scores
50 F = F/3
55 IF F<60 THEN G$(I) = 'F'
60 IF F>= 60 THEN IF F<70 THEN G$(I) = 'D'
70 IF F>= 70 THEN IF F<80 THEN G$(I) = 'C'
80 IF F>= 80 THEN IF F<90 THEN G$(I) = 'B'
90 IF F>= 90 THEN G$(I) = 'A'
100 PRINT 'Another student';\ INPUT R$
110 IF R$ = 'no' GO TO 150
120 IF R$<>'yes' GO TO 100
```

```

125 REM - count the next student
130 I=I+1
140 GO TO 20
150 PRINT 'STUDENT NAME','GRADE 1','GRADE 2','GRADE 3','FINAL GRADE'
160 FOR J=0 TO I
170 PRINT S$(J),G1(J),G2(J),G3(J),G$(J)
180 NEXT J
190 END
RUNNH

```

Input student name and three test scores? Andrea,90,93,95  
 Another student? yes  
 Input student name and three test scores? Jason,85,75,80  
 Another student? no

STUDENT NAME	GRADE 1	GRADE 2	GRADE 3	FINAL GRADE
Andrea	90	93	95	A
Jason	85	75	80	B

READY

## HOW ARRAYS ARE STORED IN THE WORKSPACE

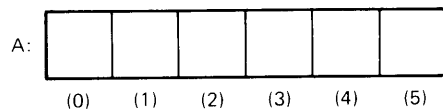
It is not necessary for you to read the following discussion of arrays unless you intend to use some of the specialized MINC program features (see Books 4 and 6).

When you dimension an array with a DIM statement, MINC sets up an area in the workspace for the array.

One-dimensional arrays are stored as they have been previously pictured. Thus, the array dimensioned in the following statement:

```
10 DIM A(5)
```

is stored in the workspace as shown in Figure 11.



MR 1591

Figure 11. One-Dimensional Array

Although you can think of two-dimensional arrays as matrices, they are stored linearly in the workspace. Thus, the array dimensioned in the following statement:

```
10 DIM B(2,3)
```

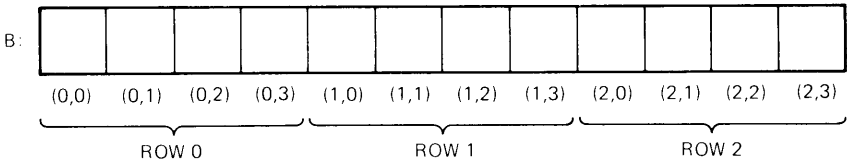
can be pictured as in Figure 12.

ROW 0	B(0,0)	B(0,1)	B(0,2)	B(0,3)	ROW 0
ROW 1	B(1,0)	B(1,1)	B(1,2)	B(1,3)	ROW 1
ROW 2	B(2,0)	B(2,1)	B(2,2)	B(2,3)	ROW 2

MR-1592

Figure 12. Two-Dimensional Array

However, this array is stored in the workspace as shown in Figure 13.



MR-1593

Figure 13. Two-Dimensional Array Stored in the Workspace

MINC stores a two-dimensional array in *row-major order*. That is, MINC stores all of row zero, then all of row one, and so forth. In this order, the rightmost subscript varies the fastest as you look down the linear list of the two-dimensional array.

When working with a two-dimensional array in a BASIC program, you must always use two subscripts when referencing one of the array elements. Thus, the way the array is stored in the workspace is usually of little importance to you.

However, some aspects of the lab module routines discussed in Book 6 are easier to use if you know how two-dimensional arrays are stored in the workspace.

In MINC you perform this calculation by typing in the following BASIC command:

```
PRINT (27 + 32)*(15-8)/327
```

MINC displays the answer beneath the PRINT statement, like this:

```
PRINT (27 + 32)*(15-8)/327
1.263
```

When you use MINC as a calculator, you are using it in what is called the *immediate mode*. In the immediate mode, MINC performs your instructions as soon as you press RETURN.

The rules for using MINC as a calculator are quite simple and are explained in this chapter.

## THE PRINT STATEMENT

To get MINC to display information on the terminal, you must use the PRINT statement. As you saw above, the form of the PRINT statement is:

```
PRINT expression (RET)
```

where expression represents the number or calculation that is to be printed, and (RET) represents pressing the RETURN key. The expression is printed in blue ink, because it is optional in a PRINT statement. If you omit the expression, MINC prints a blank line.

The following three examples are valid PRINT statements where 7, 23.8, and the result of  $5324.7 + 78625.9$  are to be printed by each PRINT statement respectively.

```
PRINT 7
```

```
PRINT 23.8
```

```
PRINT 5324.7 + 78625.9
```

If you enter each of the above statements, MINC prints the results on the screen as follows:

```
PRINT 7
7
```

## CHAPTER 8

# DATA TYPES AND FUNCTIONS

Chapter 2 discussed numeric variables and numeric functions (the trigonometric functions and some arithmetic functions). This chapter treats data types and numeric variables in more detail and discusses numeric and string functions.

So far, this manual has dealt with two data types: numeric and string. String variables and literals have been discussed completely; however, numeric variables and literals have been simplified until now.

There are two kinds of numeric variables and literals: real and integer. These numeric data types are discussed in the following sections.

Those variables and literals that have previously been described as numeric are *real variables* and *literals*. Real numbers can have a fractional part or they can be whole numbers. For example, 7 and 7.3 are both real literals. The limits on real literals are  $10^{-38}$  to  $10^{38}$ .

Real variables are named by a letter, or a letter followed by a digit. Thus, A and A1 are two real variable names. A program can have at most 286 real variables (A-Z, and A0-A9 through Z0-Z9).

*Integer variables* and *literals* can have only whole number values. If you try to give an integer variable a fractional value, MINC truncates the value to the whole number portion.

### DATA TYPES

#### Real Variables and Literals

#### Integer Variables and Literals

To distinguish an integer literal from a real literal, you must type a percent sign (%) after the literal. For example, the following literal is a real literal.

7

The integer literal 7 is denoted as follows:

7%

The limits on integer literals are -32,768 to 32,767.

*Integer variables* can take on only whole number values. Like integer literals, integer variables never have a fractional part. Integer variables have the same range of values as integer literals, -32,768 to +32,767.

MINC integer variable names are represented with a letter followed by a percent sign, or with a letter followed by a digit followed by a percent sign. For example, A% and A2% are two distinct integer variable names.

MINC stores real numbers and integers differently. A real number takes twice as much room in the workspace as an integer, but a real number has more precision (the fractional part) than an integer and has a different range.

You must be careful when you perform arithmetic operations using integer variables because fractional results of any operation are truncated.

Operations involving only integer operands are called *integer arithmetic*. Operations involving a combination of integer and real operands are called *mixed mode arithmetic*.

#### NOTE

The term *whole number* is used to denote a number with no fractional part. A whole number can be stored in a real variable or can be a real literal. The term *whole number* does not denote a data type. In contrast, the term *integer* refers to the *data type*. An integer literal must be followed by a percent sign. Integers must be whole numbers, but whole numbers are not necessarily integers.

**Integer Arithmetic**

MINC handles operations that use only integer operands quite differently than it does those involving real numbers. For example, because there are no decimal fractions allowed with integers, the following division results in a value of 0.

```
PRINT 1%/7%
0
```

The above example, and the following examples in this section, are not trying to show good use of MINC. That is, you would probably never print the value 1%/7%. However, while using MINC you might want to do a division using integer variables, or perform an operation using a mixture of integer variables and real literals or variables. The following examples show you how to do these operations. To make the examples clear, this section will use literals rather than variables, even though you would never do these specific types of operations with literals.

Again, the limits on integer values are -32768 to +32767. If you exceed these limits, MINC will print an error message. For example:

```
PRINT 3% * 30000%
?MINC-W-Value of integer expression not in range -32768 to +32767
0
```

MINC warns you that it cannot handle an integer as large as 90,000 and substitutes 0 for the result of the calculation.

If you assign a real value to an integer variable, MINC truncates the value; that is, it cuts off the fractional portion. For example:

```
A% = 7.999
PRINT A%
7
```

You can perform calculations with a mixture of integer and real literals and variables. However, unless you understand how MINC performs the *mixed mode arithmetic*, you can obtain very unexpected results.

**Mixed Mode Arithmetic**

Real numbers take precedence over integers. That is, a combination of real and integer operands results in a real result. For example, in the first division below, the real operand (15) produces a real result (.666667). In the second division, both operands are integer, producing an integer result.

```
PRINT 10%/15
.666667
```

However:

```
PRINT 10%/15%
0
```

The order in which MINC does the calculations can affect the result. In the following example, MINC performs the calculation from left to right because division and multiplication are of equal priority. The integer division is done before the real multiplication, resulting in a value of 0.

```
PRINT 10%/15%*20
0
```

This same calculation follows, but in a different order. In this second example, the multiplication is done first, resulting in a real interim result, which causes the division to produce a real result.

```
PRINT 20*10%/15%
13.3333
```

Parentheses override operator precedence in mixed mode arithmetic. For example:

```
PRINT 20*(10%/15%)
0
```

When MINC assigns a real number to an integer variable, it truncates the value before making the assignment. For example:

```
A% = 15/10
```

```
PRINT A%
1
```

The following chart sums up the results of mixed mode operations:

<i>operand 1</i>	<i>operand 2</i>	<i>result</i>
real	real	real
real	integer	real
integer	real	real
integer	integer	integer



In Chapter 2, the following arithmetic and trigonometric functions were discussed.

## ARITHMETIC AND TRIGONOMETRIC FUNCTIONS

### *Arithmetic functions*

SQR	the square root function
EXP	the exponential function
LOG, LOG10	the logarithmic functions

### *Trigonometric functions*

PI	the function that computes the value of $\pi$
SIN	the sine function
COS	the cosine function
ATN	the arc tangent function

You can use the arithmetic and trigonometric functions in a program as easily as in the immediate mode because the argument of a function can include variables as well as constants.

For example, the following program computes the nonimaginary roots of the following quadratic equation.

$$0 = ax^2 + bx + c$$

This program uses the quadratic formula given below.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
10 PRINT 'Input a, b, and c'; INPUT A,B,C
20 D=B^2-4*A*C
25 IF D>=0 GO TO 40
30 PRINT 'Roots are imaginary'
35 GO TO 70
40 R1=(-B+SQR(D))/2*A
50 R2=(-B-SQR(D))/2*A
60 PRINT 'The roots are';R1;'and';R2
70 END
```

The general form of an arithmetic or trigonometric function is:

function name (argument)

The argument can be any arithmetic expression. The remaining arithmetic functions available with MINC are:

INT     calculates the integer portion of its argument

ABS     calculates the absolute value of its argument

RND     produces a random number

SGN     computes the sign of its argument

These arithmetic functions are described in the following four sections.

### Integer Function

The integer function (INT) takes on the value of the greatest integer less than the value of the argument. If the argument is not a whole number, the function truncates the fractional part.

The form of the integer function is:

INT(expression)

For example:

```
PRINT INT(7)
7
```

```
PRINT INT (-7.35)
-8
```

```
PRINT INT (7.3E-9)
0
```

### Absolute Value Function

The absolute value function (ABS) takes on the absolute value of the argument of the function. The form of the absolute value function is:

ABS(expression)

For example:

```
PRINT ABS (-3)
3
```

```
PRINT ABS (-16.25)
16.25
```

### Random Number Function and RANDOMIZE Statement

The random number function (RND) generates a pseudo-random number between 0 and 1 each time it is invoked. The term *pseudo-random number* is used because MINC computes the random numbers according to a formula that ultimately repeats its sequence of numbers. However, this sequence of numbers is

so large that the numbers can be considered random.

For example:

```
PRINT RND  
  .0407319
```

Following every RUN or SCR command, the RND function will begin at the same point in the set of random numbers. This feature helps you find potential errors in your program by producing the same conditions for each run.

To cause MINC to produce a set of random numbers with a new starting point for each run, use the RANDOMIZE statement in the immediate mode or as a statement in your program. For example:

```
READY  
SCR
```

```
READY  
PRINT RND  
  .0407319
```

```
READY  
SCR
```

```
READY  
PRINT RND  
  .0407319
```

```
READY  
SCR
```

```
READY  
RANDOMIZE
```

```
READY  
PRINT RND  
  .86727
```

```
READY  
SCR
```

```
READY  
RANDOMIZE
```

```
READY  
PRINT RND  
  .169211
```

```
READY
```

### Computing the Sign of an Expression

The RANDOMIZE statement is a BASIC statement and not a function. It does not calculate a value. Its only purpose is to cause MINC to start the random numbers at various points in the sequence.

The sign function (SGN) takes on the value +1 if the argument is positive, a value of -1 if the argument is negative, or a value of 0 if the argument is 0.

The form of the SGN function is:

SGN (arithmetic expression)

For example, in the following assignment statement, X is assigned the value -1.

```
X=SGN(-3)
```

```
PRINT X  
-1
```

Note that the following mathematical relationship is true in MINC.

```
X=SGN(X)*ABS(X)
```

## STRING FUNCTIONS

When you use a string variable or string literal, MINC stores the characters in the string in the workspace. MINC stores characters in a code called the ASCII code. This code represents each character with a numeric code.

For example, the internal representation of the character '2' is very different from the internal representation of the number 2. For example, the decimal value of the internal representation of the number 2 is 2. The decimal value of the ASCII code for the character '2' is 50. The ASCII code for 'A' is 65 and the ASCII code for 'a' is 97. The ASCII codes are all in the range 0 to 255. For a full list of the characters and their ASCII codes, see the table in Appendix A.

MINC provides the string functions that allow you to examine and modify strings of ASCII characters and perform certain string-to-numeric conversions.

String functions that produce a string value have a dollar sign (\$) at the end of their name. String functions that produce real or integer numbers do not have a dollar sign.

When you start your MINC, you have to type the date and time. MINC has functions that let you determine the current values MINC has for the date and time.

### Clock and Calendar Functions

The CLK\$ function produces the time, measured on a 24-hour clock. The DAT\$ function produces the date. For example:

```
Please enter
Today's date: 26-APR-80
Current time: 11:28
```

```
READY
PRINT CLK$
11:28:37
```

```
READY
PRINT CLK$
11:28:43
```

```
READY
PRINT DAT$
26-APR-80
```

```
READY
```

The CLK\$ and DAT\$ functions have no arguments.

If, when using MINC, you want to change the time or date that MINC has stored, you can use the TIME and DATE commands. The TIME command sets the MINC system clock, and the DATE command sets the system date. The form of the TIME command is:

### The TIME and DATE Commands

```
TIME hh:mm:ss
```

where hh stands for hours, mm stands for minutes, and ss stands for seconds. The minutes and seconds as well as the trailing colons are optional. The MINC system clock is a 24-hour clock.

The form of the DATE command is:

```
DATE dd-mmm-yy
```

where dd stands for the day of the month, mmm stands for the first three letters of the month, and yy stands for the last two digits of the year.

For example:

TIME 17:15

READY  
DATE 1-FEB-80

READY  
PRINT CLK\$  
17:15:35

READY  
PRINT DAT\$  
1-FEB-80

READY

### String Manipulation Functions

The MINC string manipulation functions allow you to do the following:

- Determine the length of a string (LEN).
- Trim off trailing blanks from a string (TRM\$).
- Search for the position of a set of characters within a string (POS).
- Copy a segment from a string (SEG\$).

Remember that the string operation is concatenation (& or +). The string relational operators test for equality of strings (=, <>) and alphabetic precedence (<, >, <=, >=).

### Finding the Length of a String

You can use the LEN function to find the length, or number, of characters in a string. The LEN function produces a whole number value equal to the length of the string you specify. The form of the LEN function is:

LEN(string)

The following example prints the length of a string containing all the letters in the alphabet.

```
10 A$ = 'abcdefghijklmnopqrstuvwxyz'  
20 PRINT LEN(A$)  
RUNNH
```

26

READY

The TRM\$ function produces the argument string with any trailing blanks removed. The form of the TRM\$ function is:

### Trimming Trailing Blanks Off a String

TRM\$(string)

The following example concatenates and prints two strings both before and after it trims trailing blanks.

```
10 A$ = 'abcd'
20 B$ = 'efg'
30 PRINT 'before trimming: '; A$ & B$
40 PRINT 'after trimming: '; TRM$(A$) & B$
RUNNH
```

before trimming: abcd efg  
after trimming: abcdefg

READY

You can use the POS function to see if a group of characters (*substring*) occurs within a larger string. The form of the POS function is:

### Finding the Position of a Substring

POS(search-string, substring, numeric-expression)

where:

search-string	is the string being searched.
substring	is the substring the POS function is searching for.
numeric-expression	is the position in the search-string at which MINC starts the search.

The POS function searches for and produces a whole number equal to the first occurrence of the substring in the search-string. POS begins the search with the character position specified by numeric-expression. If POS finds the specified substring, it produces the character position of the first character of the substring relative to the beginning of the entire string. If POS does not find the specified substring, it produces a 0.

The following example translates each name of a month to its numeric equivalent (for example, APR to 4). In line 140 the POS function returns the position of the input string M\$ in the string containing the first three letters of each month, T\$.

If the program finds the month you specify, it prints the number of the month. If it does not find the month, it requests you to try again.

```
10 T$ = 'JANFEBMARAPR MAYJUNJULAUGSEPOCTNOVDEC'
100 PRINT 'type the first 3 letters of a month'; \ INPUT M$
120 IF M$ = 'END' GO TO 32767
130 IF LEN(M$) <> 3 GO TO 200
140 M = (POS(T$, M$, 1) + 2) / 3 \ REM - find the number of the month
150 IF M <> INT(M) GO TO 200
160 PRINT M$; 'is month number'; M
170 GO TO 100
200 PRINT 'invalid entry --- try again' \ GO TO 100
32767 END
RUNNH
```

```
type the first 3 letters of a month? NOV
NOV is month number 11
type the first 3 letters of a month? MAY
MAY is month number 5
type the first 3 letters of a month? JUD
invalid entry --- try again
type the first 3 letters of a month? END
```

READY

There are certain boundary conditions that are dependent on the values of the strings and the expression. For the details of these boundary conditions, see Book 3.

## Copying Segments from a String

You can use the SEG\$ function to produce a segment (*substring*) of a string. The SEG\$ function produces a substring consisting of the characters in the string you specify between specified character positions. The original string is unchanged. For example, the following call to SEG\$ prints characters 3 through 5 of string 'ABCDEF'.

```
10 PRINT SEG$('ABCDEF', 3, 5)
RUNNH
```

CDE

READY

The form of the SEG\$ function is:

SEG\$(string, start-position, end-position)

where:

string

is the string from which the segment is returned.



**start-position** is a numeric expression specifying the starting character position of the segment.

**end-position** is a numeric expression specifying the last character position of the segment.

There are several boundary conditions based on the values of the positions and the string. For details on these boundary conditions, see Book 3.

By using the SEG\$ function and the string concatenation operator (& or +), you can replace a segment of a string. For example, line 20 of the following program replaces the characters CDE in the string A\$ with XYZ.

```
10 A$='ABCDEFGH'
20 C$=SEG$(A$,1,2)&'XYZ'&SEG$(A$,6,7)
30 PRINT C$
RUNNH
```

ABXYZFG

READY

You can use similar string expressions to replace any given characters in a string. A general formula to replace the characters in positions n through m of string A\$ with B\$ is:

```
C$=SEG$(A$,1,n-1)&B$&SEG$(A$,m+1,LEN(A$))
```

MINC provides several string functions that convert characters to numbers and numbers to characters. You can use MINC functions to make the following conversions:

### Conversion Functions

1. Character to its corresponding ASCII code (ASC).
2. ASCII code to its corresponding character (CHR\$).
3. Number to its character representation (STR\$).
4. Character representation of a number to its numeric value (VAL).
5. String representing a binary number to a decimal number (BIN). (See Book 3 for the description of this function.)
6. String representing an octal number to a decimal number (OCT). (See Book 3 for the description of this function.)

These functions provide you with flexibility in manipulating both strings and numbers.

## Character and ASCII Code Conversions

MINC uses the ASCII code to represent characters internally. The ASC function returns the decimal ASCII code of a one-character string that you specify.

See Appendix A for a table of the ASCII codes.

The CHR\$ function returns the one-character string that has the ASCII value you specify.

For example:

```
LISTNH
10 PRINT 'The ASCII code for 'A' is ';ASC('A')
20 PRINT 'The ASCII code for 'B' is ';ASC('B')
30 PRINT 'The ASCII code for 'Z' is ';ASC('Z')
40 PRINT 'The ASCII code for 'a' is ';ASC('a')
45 PRINT 'The ASCII code for 'z' is ';ASC('z')
50 PRINT 'The ASCII code';ASC('1');'represents the character ';CHR$(49)
60 PRINT 'The ASCII code';ASC('9');'represents the character ';CHR$(57)
```

```
READY
RUNNH
```

```
The ASCII code for 'A' is 65
The ASCII code for 'B' is 66
The ASCII code for 'Z' is 90
The ASCII code for 'a' is 97
The ASCII code for 'z' is 122
The ASCII code 49 represents the character 1
The ASCII code 57 represents the character 9
```

```
READY
```

The ASC and CHR\$ functions can be used with the SEG\$ function to analyze the characters in a string.

The form of the ASC function is:

ASC(string)

The string must be a one-character string. If string is a null string or contains more than one character, MINC prints the following error message.

?MINC-F-Arguments in definition do not match function called

The ASC function produces a whole number value.

The form of the CHR\$ function is:

CHR\$(numeric expression)

The CHR\$ function generates only one character at a time. The numeric expression must be greater than or equal to 0. MINC treats arguments greater than 255 as being modulo 256 (that is, MINC treats 256 as 0, 257 as 1, and so forth).

The numeric expression represents the decimal ASCII value of some characters. Its value is treated modulo 256 because the ASCII codes are within the range 0 to 255.

Two functions, VAL and STR\$, convert numbers to their ASCII character representation and vice versa. You can use these functions to input a numeric value in a string variable or to print a number without the spaces around it.

### Numbers and Their String Representation Conversions

The VAL function returns the number represented by the specified string. The form of the VAL function is:

VAL(string)

The string may contain the digits 0 through 9, the letter E (for E notation), and the symbols + (plus), - (minus), and . (decimal point), and must be a string representation of a number.

For example:

```
READY
PRINT VAL('23')
23
```

```
READY
PRINT VAL('2.345E6')
2.34500E+06
```

```
READY
A$ = '12345'
```

```
READY
PRINT VAL(A$)
12345
```

```
READY
PRINT VAL('A')
```

?MINC-F-Arguments in definition do not match function called

```
READY
```

The STR\$ function converts a number to its string representation. The form of the STR\$ function is:

STR\$(numeric-expression)

The STR\$ function returns the value of numeric-expression as it would be printed by a PRINT statement but without a leading or trailing space. Use the STR\$ function when you want to print a number without spaces before and after it or when you want to perform string operations or functions on a number.

### Example — Converting Lower Case to Upper Case

This example converts lower case input to upper case and leaves upper case input alone. The algorithm for this program can be described as follows:

1. Input a string.
2. Look at the first character in the string.
3. If the character is lower case, then convert it to upper case.
4. Append the converted character on to the new string of upper case.
5. Look at the next character.
6. Go to step 3.

Obviously lines 2 through 6 can be replaced with a FOR loop similar to the unfinished loop shown below.

```

7 FOR I=1 TO length of string
8 T$=next character
9 IF T$ is lower case THEN convert it
10 R$=R$ & T$
11 NEXT I
    
```

Remember that & and + are the string concatenation operators (see Chapter 3, page 39).

In line 7, *length of string* is simply LEN(S\$), where S\$ is the string to be converted from lower to upper case.

In line 8, to get the *next character*, use the SEG\$ function to extract the *ith* character from the input string as follows.

```
8 T$=SEG$(S$,I,I)
```

So far, the program is this:

```

1 INPUT S$
7 FOR I=1 TO LEN(S$)
    
```

```

8 T$ = SEG$(S$,I,I)
9 IF T$ is lower case THEN convert it
10 R$ = R$ & T$
11 NEXT I

```

Now all that remains to be done is to determine if T\$ is a lower case character and convert it.

To convert a lower case character to an upper case character, subtract the constant 32 from the ASCII code of the lower case character. You do not need to know the ASCII code of any character.

Remember that the function ASC returns the value of the ASCII code of the argument. Thus ASC('a') represents the ASCII code for 'a'. ASC('a')-32 represents the ASCII code for 'A'. Therefore, if T\$ is a lower case character, ASC(T\$)-32 represents the upper case equivalent of T\$. (See Appendix A for the table of ASCII codes.)

In fact, you do not have to remember the constant 32. It turns out that the the following mathematical relationship is true:

$$32 = \text{ASC}('a') - \text{ASC}('A')$$

To convert T\$ from lower case to upper case, you can use the following BASIC statement.

```
T$ = CHR$(ASC(T$)-32)
```

The argument ASC(T\$)-32 represents the ASCII code for an upper case character. The function CHR\$ converts this ASCII code back to a character.

Now the IF statement looks like this:

```
IF T$ is lower case THEN T$ = CHR$(ASC(T$)-32)
```

The ASCII code for each of the lower case letters is greater than the ASCII code for the equivalent upper case letters. Therefore, to determine if T\$ is lower case, you can use the following relations:

$$\text{ASC}(T\$) \geq \text{ASC}('a')$$

$$\text{ASC}(T\$) \leq \text{ASC}('z')$$

That is, if  $\text{ASC}(T\$) \geq \text{ASC}('a')$  and if  $T\$ \leq \text{ASC}('z')$ , then T\$ is

a lower case character.

Then, the IF statement becomes:

```
IF ASC(T$)>=ASC('a') THEN IF ASC(T$)<=ASC('z') THEN T$=CHR$(ASC(T$)-32)
```

Finally, the program to convert lower case to upper case input is:

```
10 REM - This program converts lower case string input in S$
20 REM - to upper case, in R$. T$ is a temporary variable.
30 REM - If the string is lower case, it is converted to upper
40 REM - case. If the string is upper case, it is left alone.
50 U=ASC('a')-ASC('A')
60 PRINT 'Input a string'; \ INPUT S$
70 FOR I=1 TO LEN(S$)
80 T$=SEG$(S$,I,I)
90 IF ASC(T$)>=ASC('a') THEN IF ASC(T$)<=ASC('z') THEN T$=CHR$(ASC(T$)-U)
100 R$=R$ & T$
110 NEXT I
120 PRINT R$
RUNNH
```

Input a string? yes  
YES

READY

## USER-DEFINED FUNCTIONS

You can define your own functions and then use them as you would the SQR, SIN, or SEG\$ functions. For example, the following program defines a function to cube a variable. The function is named FNC, and it is defined in line 10.

```
10 DEF FNC(X)=X^3\ REM — define the cube function
20 PRINT 'I','I^3'
30 FOR I=1 TO 5
40 PRINT I,FNC(I)\ REM — print the number and its cube
50 NEXT I
60 END
RUNNH
```

I	I <sup>3</sup>
1	1
2	8
3	27
4	64
5	125

READY

The name of a user-defined function consists of the letters FN followed by a third letter, and optionally followed by a percent

sign (%) or a dollar sign (\$). If you end the function name with a percent sign, the function returns an integer; if you end the function name with a dollar sign, it produces a string; otherwise, the function produces a real number. Table 4 shows some valid and invalid user-defined function names.

Table 4. User-Defined Function Names

<i>Valid User-Defined Function Names</i>	<i>Invalid User-Defined Function Names</i>
FNA FNC% FNR\$	FN1 FNC1 FNC%\$

You must define each user-defined function once in a program with a DEF statement. You can define it anywhere in the program. The general form of the DEF statement for user-defined functions is:

DEF name (dummy-argument-list) = expression

where:

name	is the function name as described previously.
dummy-argument-list	contains between one and five dummy argument names. A dummy argument in a function definition is a place holder for the actual argument. The arguments may be integer, real, or string variables.
expression	is the string or arithmetic expression that defines what the function is supposed to do. The expression contains any of the dummy arguments or any other variables in the program as well as any MINC-defined function or user-defined function.

You must ensure that the expression is the same type, string or numeric, as the function name. The rules of mixed mode arithmetic apply to user-defined functions.

You can define a function anywhere in the program. You can use any defined function in a program. The format for using the function is:

function-name (actual-argument-list)

For example,

$X = \text{FNC}(3)$

For example, in the following DEF statement, which computes the value of X modulo M, X and M are the dummy arguments.

```
10 DEF FNM(X,M) = X-M*INT(X/M)
```

The variables X and M are place holders for the actual arguments of the function. In the following assignment statement, 53 and 10 are the actual arguments of the function FNM.

```
I = FNM(53,10)
```

You can even use one of the dummy argument names as an actual variable name within the program without affecting the function definition. That is, in this example, you can use variables X and M within the program without affecting the definition of FNM.

For example, in the following program, the value of FNM(53,10) remains unchanged, even though X and M are assigned values in line 30. Note too that the execution of the function does not alter the values of the actual variables X and M.

```
10 DEF FNM(X,M) = X-M*INT(X/M)
20 PRINT FNM(53,10)
30 X = 5 \ M = 23
40 PRINT FNM(53,10),X,M
RUNNH
```

```
3
3          5          23
```

The defining expression in a DEF statement can contain any constants, variables, MINC-supplied functions or user-defined functions. For example:

```
10 DEF FNQ(A,B,C) = SQR(B^2-4*A*C)
20 DEF FNR(A,B,C) = (-B-FNQ(A,B,C))/2*A
30 DEF FNS(A,B,C) = (-B+FNQ(A,B,C))/2*A
```

You can include any variables in the defining expression. However, if the expression contains variables that are not in the



dummy argument list, they are not dummy variables. That is, when MINC evaluates the user-defined function, the variables have the value currently assigned to them in the main program.

The definition must have at least one dummy argument. However, the expression does not have to contain any variables. For example:

```
10 DEF FNA$(X) = 'This is a string constant despite the value of X'
20 R$ = FNA$(10)
30 PRINT R$
RUNNH
```

This is a string constant despite the value of X

READY

If you enter a DEF statement in immediate mode, MINC ignores it. MINC does not define a function until a program is run.

For example, if you type a function definition and then try to use the function in the immediate mode, MINC produces an error message.

```
10 DEF FNA(X) = X^2
PRINT FNA(3)
```

?MINC-F-No DEF statement for the function named

READY

However, once you run the program, the function is defined.

RUNNH

```
READY
PRINT FNA(3)
9
```

READY

Now the function works.

If you try to define the same function more than once in a program, MINC prints the following message.

?MINC-F-An earlier statement already defined the function at line 20

Line 20 is the line of the second definition, that is, the line with the error.

In MINC you perform this calculation by typing in the following BASIC command:

```
PRINT (27 + 32)*(15-8)/327
```

MINC displays the answer beneath the PRINT statement, like this:

```
PRINT (27 + 32)*(15-8)/327
1.263
```

When you use MINC as a calculator, you are using it in what is called the *immediate mode*. In the immediate mode, MINC performs your instructions as soon as you press RETURN.

The rules for using MINC as a calculator are quite simple and are explained in this chapter.

## THE PRINT STATEMENT

To get MINC to display information on the terminal, you must use the PRINT statement. As you saw above, the form of the PRINT statement is:

```
PRINT expression (RET)
```

where expression represents the number or calculation that is to be printed, and (RET) represents pressing the RETURN key. The expression is printed in blue ink, because it is optional in a PRINT statement. If you omit the expression, MINC prints a blank line.

The following three examples are valid PRINT statements where 7, 23.8, and the result of  $5324.7 + 78625.9$  are to be printed by each PRINT statement respectively.

```
PRINT 7
```

```
PRINT 23.8
```

```
PRINT 5324.7 + 78625.9
```

If you enter each of the above statements, MINC prints the results on the screen as follows:

```
PRINT 7
7
```

## CHAPTER 9 SUBROUTINES

Some of the previous examples used many BASIC statements to validate input. For example, just validating a yes or no response takes three statements as follows:

```
200 INPUT R$  
210 IF R$= 'YES' GO TO 100  
220 IF R$= 'NO' GO TO 300  
230 GO TO 200
```

These three statements do not completely test the input. For example, these three lines do not test for a lower case response or a combination of upper and lower case.

In a long program, you may prompt the user to answer yes or no to more than one question. In this case, you can put the three (or more) lines after every input of yes or no, or you can figure out a way to reuse the same three (or more) validation lines at every yes or no input.

BASIC simplifies reusing the same validation statements with a mechanism called a *subroutine*. A subroutine is a group of statements in a program with distinct entry and exit points that performs a task, such as input validation.

Before the details of using subroutines are discussed, an example is given below to show you how subroutines are used.

This example shows a subroutine that validates a yes or no input. The problem is the questionnaire program from page 101.

The subroutine prints a prompt that the program put in variable P\$, checks for a yes or no response, and sets variable Z to 1 for yes and Z to 2 for no.

The essential portion of the previous method for processing questionnaires is:

```
10 DIM R(20,3)
45 PRINT 'Input another questionnaire'; \ INPUT A$
50 IF A$ = 'YES' GO TO 70 \ IF A$ = 'NO' GO TO 310 \ GO TO 45
60 REM - Input and process one questionnaire
70 FOR I = 1 TO 20
.
.
.
350 END
```

Below is an example of the essential portion of this program now changed to use a subroutine to validate the yes or no response. The person using the program sees exactly the same thing as in the previous version.

```
10 DIM R(20,3)
45 P$ = 'Input another questionnaire'
50 GOSUB 800
55 ON Z GO TO 70, 310
60 REM - Input and process one questionnaire
70 FOR I = 1 TO 20
.
.
.
800 REM - Subroutine to validate yes or no
805 REM - P$ is the prompt string
806 REM - R$ is the response (yes or no)
810 PRINT P$ \ INPUT R$
820 IF R$ <> 'YES' THEN IF R$ <> 'NO' GO TO 810
830 IF R$ = 'YES' THEN Z = 1
840 IF R$ = 'NO' THEN Z = 2
850 RETURN
900 END
```

The GOSUB statement in line 50 transfers control to line 800. The RETURN statement in line 850 returns control to line 55, the line after the GOSUB statement.

This particular example using the subroutine is not shorter than the program it replaced. However, the subroutine handles any case that requires a yes or no response and is therefore more general. Thus, if the program must be expanded to handle many cases with yes or no responses, each new response can be handled by the subroutine.

**THE GOSUB AND  
RETURN  
STATEMENTS**

The GOSUB statement is used to transfer control to a subroutine. The form of the GOSUB statement is:

GOSUB statement-number

The statement-number is the statement number of the first statement in the subroutine to be executed.

For example, the following GOSUB statement transfers control to statement 800.

```
GOSUB 800
```

A GOSUB is not like a GO TO, however, even though it unconditionally transfers control to another statement number. The difference is that when you use a GO TO, MINC unconditionally transfers control to the new statement and forgets the statement number of the GO TO; but when you use a GOSUB, MINC remembers the statement number of the GOSUB statement. Thus, when MINC executes the statement 50 GOSUB 800, it goes to statement 800, but it remembers that it came from statement 50.

The RETURN statement is used to transfer control from the subroutine just executed back to the statement after the GOSUB that called the subroutine.

The form of the RETURN statement is:

```
RETURN
```

The following example expands the use of the validating subroutine. When the subroutine is called from line 110, it returns to line 120. When the subroutine is called from line 210, it returns to line 240.

```
10 REM - This program processes questionnaires
20 REM - the questionnaires are in two groups:
30 REM - the special group, and the rest of the questionnaires
```

```

.
.
100 P$ = 'Another questionnaire'
110 GOSUB 800
120 ON Z GO TO 200, 500
200 P$ = 'Is questionnaire in special group'
210 GOSUB 800
240 ON Z GO TO 300, 400
300 REM - Process special questionnaires
```

```
.  
. .  
. .  
400 REM - Process regular questionnaires  
. .  
. .  
500 REM - Print results  
. .  
. .  
799 GO TO 1000  
800 REM - Subroutine to validate yes or no response  
810 PRINT P$ \ INPUT R$  
820 IF R$ <> 'YES' THEN IF R$ <> 'NO' GO TO 810  
830 IF R$ = 'YES' THEN Z = 1  
840 IF R$ = 'NO' THEN Z = 2  
850 RETURN  
. .  
. .  
1000 END
```

Notice that a new statement (number 799) has been put in to go to statement 1000 after printing the results. You must be very careful to make sure that a program does not "fall through" to a subroutine. If statement 799 were not there, control would pass to statement 800 after the results were printed. MINC would finally terminate the program with an error message at statement 850 because MINC would not know where to return.

Because it is difficult to look at a section of a BASIC program and determine whether this section is a subroutine, it is a good idea to begin each subroutine with a REMARK statement that identifies the subroutine.

## EXAMPLE OF SUBROUTINES

This section gives more advanced examples of subroutines as well as some techniques and rules.

In this example, the program to convert lower case input to upper case is changed to a subroutine.

Again, the complete program to convert lower to upper case is as follows. (See also Chapter 8, page 130).

```
10 REM - This program converts lower case string input  
20 REM - to upper case.  
30 REM - If the string is lower case, it is converted to upper  
40 REM - case. If the string is upper case, it is left alone.  
50 U = ASC('a')-ASC('A')  
60 PRINT 'Input a string'; \ INPUT S$  
70 FOR I = 1 TO LEN(S$)
```

```

80 T$ = SEG$(S$,I,I)
90 IF ASC(T$) >= ASC('a') THEN IF ASC(T$) <= ASC('z') THEN T$ = CHR$(ASC(T$)-U)
100 R$ = R$ & T$
110 NEXT I
120 PRINT R$

```

To make this program a general subroutine, alter the program so that the lower-to-upper-case subroutine no longer prints a message or accepts input. To be most useful and general, the lower-to-upper-case subroutine should assume that the calling program "knows" what it wants converted to upper case. All that a lower-to-upper-case subroutine should do is convert a string from lower to upper case. The program (or other subroutine) that uses the lower-to-upper-case subroutine should worry about printing messages and inputting strings. This subroutine assumes the calling program puts the string to be converted in S\$.

Notice in the lower-to-upper-case subroutine shown below that now a new statement (statement 1030) is added to initialize R\$ to the null string. Every time you use the RUN command to run the program to convert lower case to upper case, MINC reinitializes all numeric variables to 0 and all string variables to the null string (''). However, when writing a subroutine, you do not know how many times the subroutine will be used within one run of the program. Thus, during one run, R\$ contains the old string from the previous time it was used, unless you remove the old string.

```

1000 REM - Subroutine to convert lower to upper case
1010 REM - Subroutine converts only lower case
1020 U = ASC('a') - ASC('A')
1030 R$ = ''
1040 FOR I = 1 TO LEN(S$)
1050 T$ = SEG$(S$,I,I)
1060 IF ASC(T$) >= ASC('a') THEN IF ASC(T$) <= ASC('z') THEN T$ = CHR$(ASC(T$)-U)
1070 R$ = R$ & T$
1090 NEXT I
1100 RETURN

```

The new version of the yes-no validation subroutine that uses the lower-to-upper-case subroutine is as follows:

```

800 REM - Subroutine to validate yes or no response
810 PRINT P$ \ INPUT S$
820 GOSUB 1000 \ REM - Convert response to upper case
830 IF R$ <> 'YES' THEN IF R$ <> 'NO' GO TO 810
840 IF R$ = 'YES' THEN Z = 1
850 IF R$ = 'NO' THEN Z = 2
860 RETURN

```

Notice that the yes-no validation subroutine now uses another subroutine. This process is called *nesting subroutines*.

You must remember, that any program that uses the yes-no validation subroutine given here must place a prompt in P\$ and must not use variables S\$, R\$, T\$, or U. That is, unlike user-defined functions that have dummy arguments, variables used in subroutines are actual variables within the program.

## THE ON/GOSUB STATEMENT

The ON/GOSUB statement is used to conditionally transfer control to one of several subroutines. The ON/GOSUB statement has the following form:

ON numeric-expression GOSUB list-of-statement-numbers

The numeric-expression is any valid numeric expression and the statement numbers must be separated by commas.

The ON/GOSUB statement works like the ON/GO TO statement (see page 82). When MINC executes the ON/GOSUB statement, it first evaluates the numeric expression. If the value of the expression is 1, control passes to the first line number specified; if it is 2, control passes to the second line number specified; and so forth. If the expression is less than 1 or greater than the number of line numbers in the list, MINC prints the following error message.

?MINC-F-Value of control expression is out of range at line XX

where XX represents the statement number where the error occurred.

The following statement is an example of an ON/GOSUB statement.

```
20 ON A + B GOSUB 200,300,120
```

If A + B is equal to 1, then MINC passes control to line 200. If A + B is equal to 2, then MINC passes control to line 300. If A + B is equal to 3, then MINC passes control to line 120. If A + B is greater than 3 or less than 1, MINC prints out the error message and stops executing the program.

In this example, no matter which subroutine is executed from the ON/GOSUB statement, as soon as MINC reaches a RETURN statement, it passes control back to the statement



physically after the ON/GOSUB statement (the statement after line 20 in this example).

When you use the RESEQ command to resequence all or part of a program, MINC correctly resequences all references to subroutines. (See pages 82 and 83 for other references to the RESEQ command.) For example, suppose the program below is in need of resequencing.

## RESEQUENCING PROGRAMS WITH GOSUBS

```

3 DIM R(20,3)
6 P$ = 'input another questionnaire'
9 GOSUB 45
12 ON Z GO TO 15,30
15 FOR I = 1 TO 20
18 PRINT I; \ INPUT Q
21 R(I,Q) = R(I,Q) + 1
24 NEXT I
27 GO TO 6
30 PRINT 'AGREE', 'DISAGREE', 'DON'T CARE'
33 FOR I = 1 TO 20
36 PRINT 'question'; I, R(I,1), R(I,2), R(I,3)
39 NEXT I
42 GO TO 63
45 REM - subroutine to validate yes or no
48 PRINT P$ \ INPUT R$
51 IF R$ <> 'yes' THEN IF R$ <> 'no' GO TO 48
54 IF R$ = 'yes' THEN Z = 1
57 IF R$ = 'no' THEN Z = 2
60 RETURN
63 END

```

Then, the RESEQ command changes the statement numbers, and the program is numbered as shown below. Notice that the GO TOs, the ON/GO TOs, and the GOSUB statements are renumbered correctly.

```

10 DIM R(20,3)
20 P$ = 'Input another questionnaire'
30 GOSUB 150
40 ON Z GO TO 50,100
50 FOR I = 1 TO 20
60 PRINT I; \ INPUT Q
70 R(I,Q) = R(I,Q) + 1
80 NEXT I
90 GO TO 20
100 PRINT 'AGREE', 'DISAGREE', 'DON'T CARE'
110 FOR I = 1 TO 20
120 PRINT 'question'; I, R(I,1), R(I,2), R(I,3)
130 NEXT I
140 GO TO 210
150 REM - subroutine to validate yes or no

```

## PROGRAMMING FUNDAMENTALS

```
160 PRINT P$ \ INPUT R$
170 IF R$<>'yes' THEN IF R$<>'no' GO TO 160
180 IF R$='yes' THEN Z=1
190 IF R$='no' THEN Z=2
200 RETURN
210 END
```

MINC can also resequence programs with ON/GOSUB statements. Remember, however, that the RESEQ command does not update any statement numbers that are referenced within REMARK statements.

## CHAPTER 10

# MINC ROUTINES

You can use the graphic, instrument bus, and lab module features of MINC in the immediate mode or in the program mode. The MINC system provides these capabilities in *routines*, a means by which you request MINC to perform a complex task.

There is a set of graphic routines, a set of instrument bus routines, and a set of lab module routines. Book 4 describes how to use the graphic routines, Book 5 describes how to use the instrument bus routines, and Book 6 describes how to use the lab module routines.

This chapter describes the general format for using routines. The examples in this chapter use graphic routines because all MINC systems include the graphic capability, but the concepts described here apply to the lab module and instrument bus routines as well.

For example, the following BASIC program uses the graphic routine HTEXT to display two lines on the screen, a flashing boldface line and a reverse video line. Type this program in and run it to see the graphics that this program generates.

```
10 HTEXT('flash,bold',22,1,'Flashing boldface')
20 HTEXT('reverse',23,1,'Reverse video')
```

HTEXT is the *routine name*, 'flash,bold', 22, 1, and 'Flashing boldface' are the routine arguments.

The task that routine HTEXT performs is printing a line on the terminal screen. The way the printed line looks depends on the arguments that you specify.

As you can see by this example, you use a routine as you would any other BASIC statement. You can use routine statements where you need them in a program along with any other BASIC statements. For example:

```
10 PRINT 'Another line'; \ INPUT R$
20 IF R$<>'yes' THEN STOP
30 PRINT 'Input line to print in flashing boldface'; \ INPUT L$
40 IF L$<>' ' THEN htext('flash,bold',23,1,L$) \ GO TO 10
50 PRINT 'Line entered is null — try again' \ GO TO 30
```

Notice that MINC does not convert routine names to upper case as it does for any other BASIC statement. Thus, if you type in a routine name in lower case, it remains in lower case.

For more information on the routines, see Books 4, 5, and 6.

## CHAPTER 11

### FILE CONTROL

When you type the SAVE command, MINC saves the program in the workspace in a program file on the volume that you specify. In Chapter 4 the definition of program file is simply that place on the volume in which the program is stored.

A *file* in general is a named portion of a volume that holds information. It is conceptually similar to a file kept in a drawer in a filing cabinet. You can think of the cabinet as a volume, and the drawers as files. Each folder within the drawer holds information (probably on paper). The drawer (file) holds a set of similar data. Each folder holds one record or entry. However, the kind of information in a file may vary from file to file. For example, one file may hold correspondence and another file may hold sales brochures.

A file on a volume can hold a program. Other files on the same volume can hold, for example, the ASCII characters of a letter you typed or data collected by an instrument connected to MINC. This chapter discusses the two types of MINC files, sequential files and virtual array files, and suggests possible uses for these files.

When you type a SAVE command, MINC creates a file on the volume you specify with the name that you specify. For example:

SAVE QUEST

This command creates a program file on the default volume,

#### PROGRAM FILES AND OTHER FILES

names the program file QUEST.BAS, and stores the ASCII characters of the program in the workspace into the program file. The program file on the volume looks like this:

```
10 SP DIM SP R(20,3) RET 20 SP P$ = 'Input SP another SP quest...
```

That is, the program file holds every ASCII character in the program, including the spaces ( SP) and RETURNS ( RET). When you use the OLD command, MINC automatically transfers the ASCII characters from the file into the workspace. Then the program is there, ready for you to run.

There are other kinds of files besides program files. For example, you may want to collect more data from an instrument than MINC can store in the workspace. To do this, you must create a file on a volume in which MINC can store the data. As another example, you may need to use an array in a program that does not fit in the workspace. You can create a file on a volume that MINC can use as extra array space.

When you are using program files, MINC manages them for you when you type the SAVE, REPLACE, or OLD commands. When you use nonprogram files, such as a data file, you must handle creating the file, putting information into the file, and storing the file yourself with the appropriate BASIC statements in a BASIC program.

To create a file or use an existing file, you must *open* the file with an OPEN statement. To make sure the file is stored properly on the volume, you must *close* the file with a CLOSE statement after you have used the file. How you use the file depends on the type of file; that is, whether it is a sequential file or a virtual array file.

A *sequential file* is a file that must be accessed serially. That is, to get to the fifth piece of information in the file, MINC must first look at pieces one, two, three, and four. The information in a sequential file is always ASCII characters, and MINC can either output information from the workspace to an open sequential file or input information from a sequential file to the workspace, but MINC cannot do both input and output with the same sequential file. Program files are sequential files.

A *virtual array file* is a file that can be accessed directly. That is, MINC can access the fifth piece of information without having to access the first four. This type of file is called a virtual *array* file, because you use it in your program as you would an array.

The term *virtual* is used because this file is not an array stored in the workspace even though it looks like an array to the program; it is a file stored on a volume. Because you can treat a virtual array file like an array, MINC can both input from and output to the same virtual array file.

The information in a virtual array file can be ASCII characters or it can be numeric information where 2 is stored as the number 2 and not as the ASCII character "2".

The following sections describe using sequential and virtual files.

## SEQUENTIAL FILES

You use sequential files in the same way that you use terminal input and output. However, when you input from a sequential file (using an INPUT or LINPUT statement), MINC inputs the information from the sequential file instead of from the user sitting at the terminal. When you output to a sequential file (using a PRINT statement), MINC writes the information in the file instead of on the terminal screen. Before you can use a sequential file, however, you must open the file.

### Opening a Sequential File

You open a sequential file with an OPEN statement. The OPEN statement opens a *channel* over which MINC transfers the information from the file to the workspace. A channel in MINC is similar to a television channel. Currently, television channels are open for input only. You tune your television to a channel and receive a television program.

The OPEN statement links the file with the channel until the channel is closed by a CLOSE statement. Then the channel becomes available again. There are 12 available channels numbered 1 through 12. Channel 0 is always open to the terminal, and you cannot open it.

The three most common forms of the OPEN statement are:

OPEN filespec-string FOR INPUT AS FILE # numeric-expression

OPEN filespec-string FOR OUTPUT AS FILE # numeric-expression

and

OPEN filespec-string AS FILE # numeric-expression

where:

filespec-string	is a string representing the file specification.  The file name is any name (up to 6 characters) that you choose to call the file. The file type is any 3-character file type you want. If you do not specify the file type, MINC defaults to .DAT for the file type. The .DAT file type stands for data. Thus the file type is by default not a program file.
FOR INPUT	is optional and specifies using an existing file.
FOR OUTPUT	specifies creating a new file.
#	is optional.
numeric-expression	is the channel number of the file. Later in the program when you want to refer to the file, you use the channel number instead of the file name. The channel number can have any whole number value between 1 and 12.

If you specify FOR INPUT, MINC opens an existing file, and you can only input information from it to the workspace.

If you specify FOR OUTPUT, MINC creates a new file. Any existing file with the same file specification is superseded when the new file is closed (see next section). If you specify FOR OUTPUT for a sequential file, you can only write to the file.

Specifying neither FOR INPUT nor FOR OUTPUT for an existing sequential file is equivalent to specifying FOR INPUT. If the sequential file does not exist, specifying neither is equivalent to specifying FOR OUTPUT.

If you use LP: as the filespec with the FOR OUTPUT option, MINC can output to the line printer with the PRINT# statement described later in the chapter. For more details see the section entitled "Line Printer" in Book 3.

Following is an example of creating a new sequential file called QIN.DAT on the system volume.



```
10 OPEN 'QIN' FOR OUTPUT AS FILE #1
```

Remember that the .DAT file type is the default file type. If you want to create a file with another file type, you must specify the file type in the file specification. For example, the following OPEN statement creates a new sequential file with the name QIN.TXT on the SY1: volume.

```
10 OPEN 'SY1:QIN.TXT' FOR OUTPUT AS FILE #2
```

For more advanced features of the OPEN statement, see Book 3.

When you create a sequential file using the OPEN statement with FOR OUTPUT, you must close the file in the program with a CLOSE statement. If you do not close the file and the program terminates with an error, MINC does not create the file.

### **Closing a Sequential File**

The two forms of the CLOSE statement are:

```
CLOSE # expression, # expression, ...
```

```
CLOSE
```

The expression is the channel number of the file to close. You can close more than one file with the CLOSE statement.

If you use the CLOSE statement with no arguments, MINC closes all open files.

If the program terminates normally (that is, by executing the END statement or the statement with the highest line number), MINC closes all files for you automatically. However, you should put in a CLOSE statement.

You use sequential files in the same way that you use terminal input and output. You program MINC to input from a sequential file with an INPUT or LINPUT statement, and MINC inputs the information from the sequential file instead of from the terminal. You put information into the sequential file with a PRINT # statement.

### **Using a Sequential File**

You create a sequential file by opening with the FOR OUTPUT specification. Once the file is open, you put information into it with the PRINT # statement.

### **Storing Data in a Sequential File**

The form of the PRINT # statement is:

```
PRINT # expression,argument-list
```

where:

expression is the channel number. Note that this channel number must match a channel number from an OPEN statement. If the expression is 0, MINC prints the output on the terminal.

argument-list contains the items to be printed just as you would print information on the terminal screen. The argument list can contain any numeric and string expressions and the TAB function. You can separate the items to be printed with commas or semicolons, directing MINC to output the information to the file with or without print zones.

You can use a colon (:) instead of a comma (,) after the expression.

If there are no items in the argument list, MINC prints a blank line in the file. When there are no items in the list, you need not specify the comma or colon after the expression.

The following example creates a sequential file and puts 5 names in it.

```
10 OPEN 'NAMES' FOR OUTPUT AS FILE #1
20 PRINT #1, 'JOHN S. DOE',' ','THOMAS R. SMITH'
30 PRINT #1, 'JANET Q. BROWN',' ','CHERYL F. JONES'
40 PRINT #1, 'ANDREW G. SCOTT'
50 CLOSE #1
60 END
RUNNH
```

READY

Line 10 creates the file. Lines 20, 30, and 40 put the names in the file. Notice that the #1 in the PRINT statements matches the #1 in the OPEN statement.

Notice also that between each name is a comma in a string literal ("," ). You should print a comma in a string literal between each data item on a line in the file. If you do not, later you will not be able to input the data items from the file using an INPUT statement.

The CLOSE statement in line 50 closes the file.

After you run this program, if you look at the directory (DIR), you will see a new file on the volume called NAMES.DAT. If you look at NAMES.DAT with the TYPE command, the file looks like this:

```
JOHN S.DOE,THOMAS R. SMITH
JANET Q. BROWN,CHERYL F. JONES
ANDREW G. SCOTT
```

The following example reads the file NAMES.DAT created in the previous program, and prints the results on the terminal.

### Accessing Data in a Sequential File

```
10 OPEN 'NAMES' FOR INPUT AS FILE #1
20 INPUT #1,A$,B$
30 LINPUT #1,C$
40 INPUT #1,D$
50 PRINT A$,B$
60 PRINT C$
70 PRINT D$
80 END
RUNNH
```

```
JOHN S. DOE          THOMAS R. SMITH
JANET Q. BROWN,CHERYL F. JONES
ANDREW G. SCOTT
```

Notice that the first INPUT # statement at line 20 has the same number of variables (A\$,B\$) as the first line in the NAMES file has values (JOHN S. DOE, THOMAS R. SMITH). An INPUT # statement uses one whole line of input, even if the number of variables in the statement is less than the number of values in the corresponding line of the input file. For example, suppose the input line was A,B,C and the input statements were:

```
10 INPUT #1, A$,B$
20 INPUT #1, C$
```

The value of A\$ then becomes "A" and the value of B\$ becomes "B". However, the value of C\$ becomes "" (the null string). The first INPUT # statement used the whole line of input, despite the fact that there were fewer variables than values.

The LINPUT # statement inputs a string from a file. MINC treats LINPUT # just as it treats LINPUT; all characters on the input line, including commas and quotation marks, are assigned to the string. Like the INPUT # statement, the LINPUT # statement expects input from the terminal if the file number is 0. The LINPUT # statement in line 30 reads the entire line of input from the NAMES file, including the comma.

### Checking for the End of the Input File

Suppose that you have the following input file.

```
JOHN S. DOE  
JANET Q. BROWN  
ANDREW G. SCOTT  
THOMAS R. SMITH  
CHERYL F. JONES
```

You could then write the following program to read the file and write out its contents.

```
10 OPEN 'NAMES' FOR INPUT AS FILE #2  
20 LINPUT #2,N$  
30 PRINT N$  
40 GO TO 20  
50 CLOSE #2  
60 END  
RUNNH  
JOHN S. DOE  
JANET Q. BROWN  
ANDREW G. SCOTT  
THOMAS R. SMITH  
CHERYL F. JONES
```

?MINC-F-Too few values for INPUT or READ variables at line 20

Notice that this program terminates with an error message. MINC continued to read the file until it came to the end. When MINC reached the end of the file, there were no more values to read with the LINPUT # statement, so MINC terminated the program and printed the message.

In this case, the program is terminated before the file is closed in line 50. There are no problems in this example, because the program does not create any sequential files. However, if the program were creating a sequential file, and did not close the file, the file would be lost. You can use the CLOSE statement in the immediate mode if this happens (see Book 3).

You can check for the end of file and have MINC terminate the program without an error by using the IF END # statement. The form of this statement is:

```
IF END # expression THEN statement  
IF END # expression THEN statement-number  
IF END # expression GO TO statement-number
```

The expression is the file number of the file. The value of the expression can not be 0 and the file associated with the expression cannot be a terminal.

If the next attempt to input a value would produce the ?MINC-F-Too few values for INPUT or READ variables at line XX error message, MINC executes the statement after the THEN or transfers control to the specified line number. Otherwise MINC transfers control to the statement after the IF as it does with any other IF statement.

The following example terminates without an error message.

```
10 OPEN 'NAMES' FOR INPUT AS FILE #2
20 IF END #2 GO TO 40
30 GO TO 60
40 PRINT 'End of file'
50 GO TO 90
60 LINPUT #2, N$
70 PRINT N$
80 GO TO 20
90 CLOSE #2
100 END
RUNNH
```

```
JOHN S. DOE
JANET Q. BROWN
ANDREW G. SCOTT
THOMAS R. SMITH
CHERYL F. JONES
End of file
```

READY

Line 20 checks for the end of the file.

Note that the IF END # statement tests if there is one more item in the file. If there is one item left when the IF END checks the file and your INPUT # statement requests two items, MINC prints the ?MINC-F-Too few values for INPUT or READ variables at line XX error message and terminates the program without closing the file.

The RESTORE # statement resets the specified sequential input file from its current position to its beginning. The format of the RESTORE # statement is:

RESTORE # expression

The expression is the channel number of the file to be restored.

This example merges two files of names and addresses in alphabetical order into one file in alphabetical order and removes duplicate names and addresses.

### Restoring a File to the Beginning

### Example of Using Sequential Files

The algorithm description of the basic loop is:

1. Input a name and address (name1) from file 1 and a name and address (name2) from file 2.
2. If name1 < name2 then put name1 in the output file, get next name1 from file 1, and repeat step #2.
3. If name2 < name1 then put name2 in the output file, get next name2 from file 2, and go to step #2.
4. At this point name1 = name2. If addresses match then they are duplicates, put name1 in file, and go back to step #1.
5. If addresses do not match, put both names and addresses in file and go back to step #1.

The following section of a BASIC program implements this algorithm. Compare this program section to the algorithm. N1\$ is the name from file number 1, A1\$ is the street or P.O. Box number from number 1, and S1\$ is the city and state from file number 1. N2\$,A2\$, and S2\$ correspond to the names, streets or P.O. Box numbers, and cities from file number 2. File number 3 is the merged output file.

Lines 95 through 320 correspond to step 1 of the algorithm. Lines 330 through 400 correspond to step 2 of the algorithm. Lines 405 through 460 correspond to step 3 of the algorithm. Lines 465 through 510 correspond to step 4 of the algorithm. Finally, lines 515 through 560 correspond to step 5 of the algorithm.

```

95 REM -----
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$

195 REM -----
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350

305 REM -----
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$

```

```

330 REM -----
340 REM - Compare the files
345 REM
350 IF SEG$(N1$,1,28)>=SEG$(N2$,1,28) GO TO 410
355 REM -----
360 REM - File 1 alphabetically precedes 2
365 REM
370 PRINT #3,N1$
380 PRINT #3,A1$
390 PRINT #3,S1$
400 GO TO 230
405 REM -----
406 REM - Compare the files
407 REM
410 IF N2$=N1$ GO TO 480
415 REM -----

420 REM - Record from file 2 alphabetically precedes 1
425 REM
430 PRINT #3,N2$
440 PRINT #3,A2$
450 PRINT #3,S2$
460 GO TO 120
465 REM -----
470 REM - The names are equal
475 REM
480 PRINT #3,N1$
490 PRINT #3,A1$
500 PRINT #3,S1$
505 REM -----
506 REM - Compare the addresses
507 REM
510 IF A1$=A2$ THEN IF S1$=S2$ THEN GO TO 60
515 REM -----
520 REM - the addresses are different, so print both
525 REM
530 PRINT #3,N2$
540 PRINT #3,A2$
550 PRINT #3,S2$
560 GO TO 60

```

This program segment is not complete. It does not open the files nor test for the end of either of the files. First, in this example, the program must terminate normally or the new, merged file will not be closed, and consequently will not exist. Second, if one of the files is ended and the other is not, the program should put the rest of the unfinished file on the end of the merged file.

A specific example of the way the program should work is shown below. Notice that file 2 is longer than file 1.

*File 1*

DIGITAL EQUIPMENT CORP. A/S  
P.O. Box 3914  
7001 Trondheim, Norway  
DIGITAL EQUIPMENT CORP. S.A.  
Burgunderstrasse 42A  
Basle, Switzerland CH-4051  
DIGITAL EQUIPMENT INT., LTD.  
Ballybrit Industrial Estate  
Galway, Ireland

*File 2*

DEC DE PUERTO RICO  
P.O. Box 106  
San German, Puerto Rico 00753  
DIGITAL EQUIPMENT CORP.  
1400 Terra Bella Avenue  
Mountain View, CA 94043  
DIGITAL EQUIPMENT CORP.  
200 Forest Street  
Marlboro, MA 01752  
DIGITAL EQUIPMENT CORP.  
P.O. Box 80  
Albuquerque, NM 87103  
DIGITAL EQUIPMENT INT., LTD.  
Ballybrit Industrial Estate  
Galway, Ireland

*Merged File*

DEC DE PUERTO RICO  
P.O. Box 106  
San German, Puerto Rico 00753  
DIGITAL EQUIPMENT CORP.  
1400 Terra Bella Avenue  
Mountain View, CA 94043  
DIGITAL EQUIPMENT CORP.  
200 Forest Street  
Marlboro, MA 01752  
DIGITAL EQUIPMENT CORP.  
P.O. Box 80  
Albuquerque, NM 87103  
DIGITAL EQUIPMENT CORP. A/S  
P.O. Box 3914  
7001 Trondheim, Norway  
DIGITAL EQUIPMENT CORP. S.A.  
Burgunderstrasse 42A  
Basle, Switzerland CH-4051  
DIGITAL EQUIPMENT INT., LTD.  
Ballybrit Industrial Estate  
Galway, Ireland

The end conditions for this problem are rather complex. You do not want one of the files to finish causing the LINPUT statement to run out of data. If this happens, MINC terminates the program abnormally with an error message. Not only is the merged file not closed, it also is not finished. The names and addresses in the longer file do not appear in it because the program terminated before it read them. The LINPUT # statements given previously are:

```
110 LINPUT #1,N1,A1$,S1$
210 LINPUT #2,N2$,A2$,S2$
320 LINPUT #1,N1$,A1$,S1$
```

When the LINPUT # statement at line 110 inputs the last value from file 1, the program must still finish processing file 2.

The BASIC statements to do this are:

```
40 REM -----
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
```



```

70 GO TO 110
80 F = 2
90 GO TO 590
95 REM -----
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$

195 REM -----
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350

305 REM -----
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
.
.
.

580 REM - Finish off remaining file
585 REM
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$PRINT #3,A$PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END

```

When control passes to line 210, the program has already input a name and address from file 1 (N1\$,A1\$, and S1\$). However, if file 2 has no more values, then the program should print out N1\$, A1\$, and S1\$ before finishing off file 1 at line 590. Otherwise, program control still passes to line 590, but the program ignores the values left in N1\$, A1\$, and S1\$.

The BASIC statements to do this are:

```

40 REM -----
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
70 GO TO 110
80 F = 2
90 GO TO 590
95 REM -----
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$
115 REM -----
116 REM - Check for end of file 2

```

```

117 REM
120 IF END #2 GO TO 140
130 GO TO 210
140 F = 1
145 REM -----
150 REM - End of file 2, finish file 1
155 REM
160 N$ = N1$
170 A$ = A1$
180 S$ = S1$
190 GO TO 610
195 REM -----
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350

305 REM -----
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
.
.
.
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$\PRINT #3,A$\PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END

```

Finally, when control passes to line 320, the program has already input a name and address from file 2. Therefore, if file 1 has no more values, then the program must print out the current value of N2\$,A2\$, and S2\$ before returning to line 590 and finishing file 2.

The BASIC statements to do this are:

```

40 REM -----
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
70 GO TO 110
80 F = 2
90 GO TO 590
95 REM -----
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$
115 REM -----
116 REM - Check for end of file 2
117 REM

```

```

120 IF END #2 GO TO 140
130 GO TO 210
140 F = 1
145 REM -----
150 REM - end of file 2, finish file 1
155 REM
160 N$ = N1$
170 A$ = A1$
180 S$ = S1$
190 GO TO 610
195 REM -----
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350
225 REM -----
226 REM - Check for end of file 1
227 REM
230 IF END #1 GO TO 250
240 GO TO 320
245 REM -----
250 REM - End of file 1, finish file 2
255 REM
260 F = 2
270 N$ = N2$
280 A$ = A2$
290 S$ = S2$
300 GO TO 610
305 REM -----
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
.
.
.
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$\PRINT #3,A$\PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END

```

The whole BASIC program to merge two files is:

```

10 OPEN 'NAMES1' FOR INPUT AS FILE #1
20 OPEN 'NAMES2' FOR INPUT AS FILE #2
30 OPEN 'MAILST' FOR OUTPUT AS FILE #3
40 REM -----
50 REM - Check for end of file 1
55 REM
60 IF END #1 GO TO 80
70 GO TO 110
80 F = 2
90 GO TO 590

```

## PROGRAMMING FUNDAMENTALS

```
95 REM -----
100 REM - Input from file 1
101 REM
110 LINPUT #1,N1$,A1$,S1$
115 REM -----
116 REM - Check for end of file 2
117 REM
120 IF END #2 GO TO 140
130 GO TO 210
140 F = 1
145 REM -----
150 REM - End of file 2, finish file 1
155 REM
160 N$ = N1$
170 A$ = A1$
180 S$ = S1$
190 GO TO 610
195 REM -----
200 REM - Input from file 2
201 REM
210 LINPUT #2,N2$,A2$,S2$
220 GO TO 350
225 REM -----
226 REM - Check for end of file 1
227 REM
230 IF END #1 GO TO 250
240 GO TO 320
245 REM -----
250 REM - End of file 1, finish file 2
255 REM
260 F = 2
270 N$ = N2$
280 A$ = A2$
290 S$ = S2$
300 GO TO 610
305 REM -----
310 REM - Input from file 1
315 REM
320 LINPUT #1,N1$,A1$,S1$
330 REM -----
340 REM - Compare the files
345 REM
350 IF SEG$(N1$,1,28) >= SEG$(N2$,1,28) GO TO 410
355 REM -----
360 REM - File 1 alphabetically precedes 2
365 REM
370 PRINT #3,N1$
380 PRINT #3,A1$
390 PRINT #3,S1$
400 GO TO 230
405 REM -----
406 REM - Compare the files
407 REM
410 IF N2$ = N1$ GO TO 480
```

```

415 REM -----
420 REM - Record from file 2 alphabetically precedes 1
425 REM
430 PRINT #3,N2$
440 PRINT #3,A2$
450 PRINT #3,S2$
460 GO TO 120
465 REM -----
470 REM - the names are equal
475 REM
480 PRINT #3,N1$
490 PRINT #3,A1$
500 PRINT #3,S1$
505 REM -----
506 REM - Compare the addresses
507 REM
510 IF A1$ = A2$ THEN IF S1$ = S2$ THEN GO TO 60
515 REM -----
520 REM - The addresses are different, so print both
525 REM
530 PRINT #3,N2$
540 PRINT #3,A2$
550 PRINT #3,S2$
560 GO TO 60
570 REM -----
580 REM - Finish off remaining file
585 REM
590 IF END # F GO TO 630
600 LINPUT # F,N$,A$,S$
610 PRINT #3,N$ \ PRINT #3,A$ \ PRINT #3,S$
620 GO TO 590
630 CLOSE
640 END

```

This program is still not foolproof. A name typed in upper and lower case is not equal to the same name typed in upper case only. To make sure you are comparing comparable names, you can use the lower-to-upper-case subroutine.

Also, the LINPUT statements input three lines from a name and address file. If the number of lines in each input file is not a multiple of three, then the program will terminate abnormally and the output file will not be created. You can correct this problem by inserting an IF END statement before inputting each variable.

You should now type this program and save it. You cannot test this program, however, until you create two files of names and addresses named NAMES1 and NAMES2.

The following example shows you how to create these name and

address files as well as shows you the general form of the OPEN statement.

### Another Example

This example allows you to create a sequential file and put information into the file by running the program. (Note: You can also create files using the editor explained in Chapter 15.) However, this program does not let you correct the file if you make a mistake in typing. That is, if you press RETURN before you notice the mistake, you cannot correct the file with this program. You can, of course, use the DELETE key if you notice the mistake before you press RETURN.

```

10 PRINT "File number(1-12)"; \ INPUT N
20 PRINT "File name(max. of 6 characters)"; \ LINPUT F$
30 PRINT \ PRINT
40 PRINT "At the question mark, type the next line of the file."
50 PRINT "To end, press RETURN at the question mark."
60 REM - - Create arrays to hold records
70 DIM A$(100)
80 DIM B$(100)
90 DIM C$(100)
100 REM - - Accept records from terminal
110 T = -1
120 FOR I = 0 TO 100
130 LINPUT L$
140 IF L$ = "" THEN GO TO 330
150 LINPUT M$
160 LINPUT N$
170 REM - - Change lower case in name to upper case
180 U = ASC("a") - ASC("A")
190 R$ = ""
200 FOR K = 1 TO LEN(L$)
210 T$ = SEG$(L$, K, K)
220 IF ASC(T$) >= ASC("a") THEN IF ASC(T$) <= ASC("z") THEN T$ = CHR$(ASC(T$) - U)
230 R$ = R$ & T$
240 NEXT K
250 L$ = R$
260 REM - - Insert records into arrays
270 A$(I) = L$
280 B$(I) = M$
290 C$(I) = N$
300 T = T + 1
310 NEXT I
320 REM - - Open output file
330 OPEN F$ FOR OUTPUT AS FILE #N
340 REM - - Arrange records in alphabetical order
350 FOR C = 0 TO T
360 FOR D = C TO T
370 IF SEG$(A$(C), 1, 28) > SEG$(A$(D), 1, 28) THEN GO TO 460
380 NEXT D
390 REM - - Send record to output file
400 PRINT #N, A$(C)
410 PRINT #N, B$(C)
420 PRINT #N, C$(C)

```

```

430 NEXT C
440 CLOSE #N \ GO TO 500
450 REM - - Switch positions of two records
460 S$ = A$(C) \ A$(C) = A$(D) \ A$(D) = S$
470 S$ = B$(C) \ B$(C) = B$(D) \ B$(D) = S$
480 S$ = C$(C) \ C$(C) = C$(D) \ C$(D) = S$
490 GO TO 380
500 PRINT \ PRINT
510 PRINT "The file ";F$;" is created."
520 END

```

SAVE INFILE

READY

RUNNH

File number (1-12)?1  
 File name (max. of 6 characters)? NAMES1

At the question mark, type the next line of the file.  
 To end, press RETURN at the question mark.

?Digital Equipment Int., Ltd.  
 ?Ballybrit Industrial Estate  
 ?Galway, Ireland  
 ?Digital Equipment Corp. S.A.  
 ?Burgunderstrasse 42A  
 ?Basle, Switzerland CH-4051  
 ?Digital Equipment Corp. A/S  
 ?P.O. Box 3914  
 ?7001 Trondheim, Norway  
 ?(RET)  
 The file NAMES1 is created.

READY

To see the file, use the TYPE command. For example,

TYPE NAMES1.DAT

Remember that you must specify the .DAT file type because the TYPE command defaults to .BAS.

Now you can create the name and address files and run the file merge program.

You can use virtual array files in the same way that you use a large array. Just as you can access the elements of an array in the workspace in any order, you can access the elements of a virtual file in any order.

## VIRTUAL ARRAY FILES

Virtual array files have several advantages over sequential files.

- You can access elements in a direct, nonsequential manner. The last element in a virtual array file can be accessed as quickly as any other element. Remember that when using a sequential file, you must input the entire file before inputting the last element.
- When MINC stores data in virtual array files, it does not convert them to ASCII characters but rather stores them in their numeric representation. Consequently, there is no loss of precision caused by data conversion. There is some loss of precision with sequential files because all data are converted to ASCII. Remember that sequential files are ASCII files only.
- You can update virtual array files without copying the entire file. That is, you can open a virtual array file for both input and output.

Virtual array files also have advantages over arrays stored in the workspace.

- Virtual array files allow you to create much larger arrays than can be stored in the workspace.
- You can permanently store data in virtual array files. That is, when your program ends, the virtual array files are stored on a volume. Remember that when you run a program or use the SCR or CLEAR commands, previous arrays stored in the workspace are cleared, and thus lost.

Virtual array files also have several restrictions that do not apply to arrays stored in the workspace.

- Virtual array files are slower because MINC must read the file on the volume before manipulating data.
- Although strings stored in the workspace can have length (up to 255 characters), you cannot use these variable-length strings in virtual array files. Strings in virtual array files have a fixed maximum length from 1 to 255 that you specify in the DIM # statement (explained in the next section). MINC handles strings shorter than this maximum length similarly to strings



stored in the workspace. MINC truncates strings longer than the maximum length.

- You must dimension only one virtual array file in each DIM # statement.
- Many MINC arrays do not allow virtual array files as arguments although they accept workspace arrays as arguments.

As with sequential files, you open a virtual array file with an OPEN statement. The OPEN statement opens a sequential file unless the file-channel number specified in the OPEN statement is also specified in a DIM # statement (explained later in this section). Again, the three most common forms of the OPEN statement are:

OPEN filespec-string FOR INPUT AS FILE # numeric-expression

OPEN filespec-string FOR OUTPUT AS FILE # numeric-expression

and

OPEN filespec-string AS FILE # numeric-expression

where:

filespec	is a string representing the file specification. If you do not specify the file type, MINC defaults to .DAT for the file type.
FOR INPUT	is optional and specifies opening an existing file.
FOR OUTPUT	specifies creating a new file.
#	is optional.
numeric expression	is the channel number of the file. Later in the program when you want to refer to the file, you use the number instead of the file name. The number can have any whole number value from 0 to 12. The channel number must match the channel number in the corresponding DIM# statement.

### Opening a Virtual Array File

If you specify **FOR INPUT**, MINC opens an existing file, and you can only input information from it to the workspace.

If you specify **FOR OUTPUT**, MINC creates a new file. Any existing file with the same file specification is superseded when the new file is closed (see next section). If you specify **FOR OUTPUT** for a virtual file, you can either write to or input from the file.

Specifying neither **FOR INPUT** nor **FOR OUTPUT** for an existing virtual array file opens the file for both input and output. To update an existing virtual array file, specify neither. If the virtual array file does not exist, specifying neither is equivalent to specifying **FOR OUTPUT**.

Following is an example of creating a new virtual array file on the system volume called **QIN.DAT**.

```
10 OPEN 'QIN' FOR OUTPUT AS FILE #1
```

Remember that the **.DAT** file type is the default file type. If you want to create a file with another file type, you must specify the file type in the file specification. For example, the following **OPEN** statement creates a new virtual array file with the name **QIN.TXT** on the **SY1:** volume.

```
10 OPEN 'SY1:QIN.TXT' FOR OUTPUT AS FILE #2
```

For more advanced features of the **OPEN** statement, see Book 3.

To make **QIN.DAT** a virtual array file, you must also include a **DIM #** statement with the same file number as the **OPEN** statement. The **DIM #** statement is similar to the **DIM** statement in that it dimensions the virtual array file.

The form of the **DIM #** statement is:

**DIM # channel, array**

where:

channel	is a whole number numeric literal (with or without percent sign) that specifies the channel number.
---------	---

array	is a one- or two-dimensional array specification of the form:
-------	---

variable-name (number-of-elements)

where variable-name can be any string, integer, or real variable name, and number-of-elements represents the dimensions (maximum size) of the subscript or subscripts.

string size is an optional whole number literal (with or without a percent sign) that specifies the maximum length for elements in a string virtual array file. Its value must be in the range 1 to 255. If it is omitted for a string array, the maximum length is 16.

To access information in an existing virtual array file, be sure that the DIM # statement specifies the same variable type (string, real, or integer) and number of subscripts that are specified in the program that created the file. The variable name associated with the file can be different from the original as long as it is the same variable type.

Below are some examples of opening virtual array files.

The following example creates a 2001-element integer virtual array file. The virtual array file is named A% in this program. You can assign values to the elements and then use the values.

```
10 OPEN 'ARRAY1' FOR OUTPUT AS FILE #1
20 DIM #1, A%(2000)
```

The next example opens an existing two-dimensional, string virtual array file. The virtual array file is named F\$ in this program. Only input is allowed; that is, if you try to assign a value to an element of the array, MINC prints out the following message.

?MINC-F-OPEN statement for this file channel prohibits transfer at line 50

```
30 OPEN 'ARRAY2' FOR INPUT AS FILE #2
40 DIM #2, F$(100,2)=25
```

In this previous example, each string element can have a maximum of 25 characters. A complete example of a program using a virtual array file is given later in this chapter.

When you open a virtual array file for output you must always close the file in the program with a CLOSE statement. If you do not close the file and the program terminates with an error,

MINC leaves the file in an indeterminate state. That is, some of the changes might be in the file and some might not.

The two general forms of the CLOSE statement are:

**CLOSE**

**CLOSE # expression, # expression, ...**

The expression is the channel number of the file to close. You can close more than one file with the CLOSE statement. The CLOSE statement with no arguments closes all open files.

If the program terminates normally (that is, by executing the END statement or the statement with the highest line number), MINC closes all files for you. However, you should put in a CLOSE statement.

### **Using Virtual Array Files**

You use a virtual array file just as you use an array stored in the workspace. For example:

```
10 OPEN 'ARRAY1' AS FILE #1
20 DIM #1, A$(2000)=23
.
.
.
100 A$(5)= 'This is a string'
.
.
.
200 F$=A$(7)
.
.
1000 CLOSE #1
```

In line 100, the program puts the string value 'This is a string' in A\$(5). This line stores that value in the file. In line 200, F\$ receives the string that is already stored in A\$(7). Line 200 inputs a value from the file. Line 1000 closes the file.

### **Example of Using a Virtual Array File**

The following example uses the questionnaires program again. (See pages 101-102.) This time, the array that holds the responses is stored on a volume as a virtual array file. Because the array is now stored on the volume, you can enter some questionnaires now and some later. You no longer lose those questionnaires you typed in when you scratch the workspace.

This time, however, the program should test the input to make

sure that each response is between 1 and 3 (1 = agree, 2 = disagree, 3 = don't care). If the typist enters a value that is not between 1 and 3, and the program does not test the value, the program terminates with an error in statement 40 (the ON/GOTO). If the program terminates with an error, the virtual array file is not closed and its contents are unknown to you (that is, some of the elements might be updated while others might not be).

The new program is as follows:

LIST

QUEST                    23-MAR-80                    09:30:37

```

5 OPEN 'resp' AS FILE 1
10 DIM #1,R(20,3)
20 P$ = 'Input another questionnaire'
30 GOSUB 150
40 ON Z GO TO 50,100
50 FOR I = 1 TO 20
60 PRINT I; \ INPUT Q
63 IF Q >= 1 THEN IF Q <= 3 GO TO 70
66 PRINT 'input not between 1 and 3' \ GO TO 60
70 R(I,Q) = R(I,Q) + 1
80 NEXT I
90 GO TO 20
100 PRINT ',','AGREE','DISAGREE','DON'T CARE'
110 FOR I = 1 TO 20
120 PRINT 'question';I,,R(I,1),R(I,2),R(I,3)
130 NEXT I
140 GO TO 1000
150 REM - subroutine to validate yes or no
160 PRINT P$ \ INPUT S$
170 GOSUB 300
180 IF R$ <> 'YES' THEN IF R$ <> 'NO' GO TO 160
190 IF R$ = 'YES' THEN Z = 1
200 IF R$ = 'NO' THEN Z = 2
210 RETURN
300 REM - subroutine to convert lower to upper case
310 U = ASC('a')-ASC('A')
320 R$ = ""
330 FOR K = 1 TO LEN(S$)
340 T$ = SEG$(S$,K,K)
350 IF ASC(T$) >= ASC('a') THEN IF ASC(T$) <= ASC('z') THEN T$ = CHR$(ASC(T$)-U)
360 R$ = R$&T$
370 NEXT K
380 RETURN
1000 CLOSE #1
1010 END

```

READY

Now when you run this program, the questionnaire responses that you type are saved in the file called RESP.DAT. Notice that line 5 does not specify FOR INPUT or FOR OUTPUT. By specifying neither, the program creates the file the first time the program is run and updates the file the rest of the times the program is run.

Note that the IF END statement is not applicable to a virtual array file. Each element is equally accessible. Thus, you do not need to test for the end of the file. You just need to close the file.

## DELETING A FILE

Whenever you no longer want a virtual array file or a sequential file on one of your volumes, you can delete the file with the KILL statement.

The form of the KILL statement is:

KILL filespec

The filespec is a string that is a specification of the file. For example:

KILL 'ARRAY1'

If you do not specify the file type, the .DAT file type is the default. If you specify a device (for example, SY1:), you must then specify the entire file specification — including the file type.

The KILL statement is similar to the UNSAVE command. However, the default file type for the KILL statement is .DAT where the default file type for the UNSAVE command is .BAS.

Because you can delete a file from a program with the KILL statement, your programs can open a temporary file to create more room for your program's data, and then can delete the file at the end of the program with a KILL statement.

## RENAMING A FILE

You can change the name of a virtual array file or a sequential file with the NAME statement.

The form of the NAME statement is:

NAME file-to-be-renamed TO new-name

The new name must have the same volume specification as the

old one. That is, the NAME statement changes the file name, not the physical location. See the NAME statement in Book 3 for further details. The names are strings and must be enclosed by quotes.

For example:

```
NAME 'PROG1.BAS' TO 'INFILE.BAS'
```

If you do not specify the file type in the NAME command, the default is .DAT.

In MINC you perform this calculation by typing in the following BASIC command:

```
PRINT (27 + 32)*(15-8)/327
```

MINC displays the answer beneath the PRINT statement, like this:

```
PRINT (27 + 32)*(15-8)/327
1.263
```

When you use MINC as a calculator, you are using it in what is called the *immediate mode*. In the immediate mode, MINC performs your instructions as soon as you press RETURN.

The rules for using MINC as a calculator are quite simple and are explained in this chapter.

## THE PRINT STATEMENT

To get MINC to display information on the terminal, you must use the PRINT statement. As you saw above, the form of the PRINT statement is:

```
PRINT expression (RET)
```

where expression represents the number or calculation that is to be printed, and (RET) represents pressing the RETURN key. The expression is printed in blue ink, because it is optional in a PRINT statement. If you omit the expression, MINC prints a blank line.

The following three examples are valid PRINT statements where 7, 23.8, and the result of  $5324.7 + 78625.9$  are to be printed by each PRINT statement respectively.

```
PRINT 7
```

```
PRINT 23.8
```

```
PRINT 5324.7 + 78625.9
```

If you enter each of the above statements, MINC prints the results on the screen as follows:

```
PRINT 7
7
```