Chapter 2

PROGRAMMING STATEMENTS AND STRUCTURES

2.1 STANDARD BASIC COMMANDS

2.1.1 Variables and assignment statements

The BBC microcomputer allows the user a wide choice of names for the numerical variables that are used to identify the various parameters in a calculation. They may consist of upper and lower case letters, numbers, the pound sign and the underline character.

The important constraints are

1. They must begin with an upper or lower case letter. Starting a variable name with a number is not permitted. Every character in the name is significant, which means that TEMPERATURE1 is different from TEMPERATURE2.
2. It is not permissible to have spaces in variable names but the underline character can be used instead. Thus ‘first number’ is not acceptable but ‘first_number’ is in order.
3. Variable names must not contain punctuation marks such as , . ? ! or :.
4. Although BASIC keywords may be embedded in variable names (for example, CONCISE contains the BASIC keyword ON), they must not start with BASIC keywords, unless the keyword is in lower case letters, or partially in lower case letters. So ‘today’ and ‘Today’ are permitted but ‘TODAY’ is not because it starts with ‘TO’.

It can also be convenient to use integer variables in a program, for counting purposes perhaps, and the addition of the percent sign, %, at the end of a variable name indicates an integer variable which will not suffer from the round-off errors associated with the binary arithmetic used by computers. The BBC microcomputer has a set of *resident integer variables,* A% to Z%, which are permanently allocated space in memory. Typing RUN or NEW does not destroy them or change their values, so they can be used to pass values from one program to the next. Their values will be lost if the computer is switched off, but they will not be lost after CLEAR, <BREAK> or even <CTRL-

BREAK>.

Apart from the resident integer variables, the first appearance of any numerical variable should be in the left hand side of an assignment statement. Thus to set the value of TYRE_PRESSURE in a program it must be used in a statement such as

```
LET TYRE_PRESSURE = 32.5
```

Thereafter TYRE_PRESSURE will have the value 32.5 and the statement

```
PRINT TYRE_PRESSURE
```

will yield the result

```
32.5
```

The statement

```
X=X+1
```

is an exception to the general rule that a variable must appear on the left hand side of an assignment statement before it appears in any BASIC statement in which its numeric value is used. In this special case the value of X on the right hand side is initialy taken to be zero.

An attempt to PRINT a variable before it has been assigned a value will give the error message ‘No such variable' If this occurs within a program the line number will also be given.

The BASIC keyword LET may be omitted in an assignment statement, although beginners are encouraged to use it for a while to become familiar with the concept of the use of the ‘equal’ sign, =, in an assignment statement to mean ‘assign the the value on the right to the variable on the left'. This is opposed to the normal mathematical meaning ‘is equal to’ in a statement of equality such as 2+2=4. In BASIC the equal sign is used at various times to mean ‘set equal to’ and ‘is equal to'. The second of these is used in tests for equality in IF statements. Both are used on the BBC microcomputer, the meaning being determined by the context. It is thus perfectly correct to have the statements

```
10 X=1
20 PRINT X
30 X=X+5
40 PRINT X
```

where line 10 sets the variable X to 1, and this is printed at line 20. At line 30, X is increased by the addition of 5 to the original value of X and is thus set to 6, which will be printed at line 40. In this program = has been used to mean ‘set equal to' .

Lines 10 and 30 could equally well have been written as

```
10 LET X=1
30 LET X=X +5
```

which makes the meaning more obvious.

Note that if line 10 is removed from the above program, attempting to RUN it will produce the message

```
No such variable at Line 20
```

The normal arithmetic operations of mathematics: addition, subtraction, division, multiplication, and raising to a power are represented in BASIC by + - / * and ^ respectively. The right hand side of an assignment statement may contain calculations involving any variables already given values and any sensible combination of these operations. For example

```
LET X=P*Q+R*S
LET D=Bˆ2–4*A*C
LET Y=(A+B)/(C+D)
LET AV=(X+Y+Z)/3
```

The usual rules of precedence for evaluating these operations and the brackets apply, as follows

1. Brackets are evaluated (innermost first).
2. Raising to the power.
3. Multiplication and division.
4. Addition and subtraction.

Where operations of the same level of precedence occur together, the Computer evaluates them from left to right. It is worth remembering this point if very large or very small numbers are involved in a calculation, as the following example will illustrate.

Example 2.1

```
10 BIG1=10E20
20 BIG2=20E20
30 BIG3=1E20
40 RESULT=BIG1*BIG2/BIG3
50 PRINT RESULT
```

It will be found that when this program is run the error message

```
Too big at Line 40
```

will be obtained, since BIG1*BIG2 is 2×10^{42} and the computer cannot deal with numbers larger than 10^{38}. However, rewriting line 40 as

```
40 RESULT=BIG1/BIG3*BIG2
```

the calculation will be performed in the different order, giving the correct answer 2E22.

If the variables BIG1, BIG2, and BIG3 are set to 10E-22, 20E-20, and 1E-20 respectively, then with the first version of line 40 the value printed will be 0, and with the second, preferred form of line 40 the correct result 2E-20 is printed. There is no error message ‘Too small'; instead the value is set to 0. But if the variable RESULT were used in other calculations in the same program after it had been set to zero, they would also give incorrect results and might even cause an error if a division by RESULT were attempted, when the error message

```
Division by zero
```

would be printed. This indicates the need for care in setting out calculations involving very small numbers as well as very large numbers.

2.1.2 GOTO statements

Computers normally execute statements one line at a time in the order in which they occur in the program. In BASIC the lines occur in the order in which they are numbered. However, it is often necessary to alter the order of execution in order to execute one block of the program several times, or to make the order depend on the values being processed during the program. Changing the order of execution is known as branching and the simplest way of branching is by means of the GOTO statement. The GOTO instruction alters the normal sequence and transfers execution to the line specified in the command. This line can be either before or afteir the line containing the GOTO statement.

Example 2.2

The following program prints out the squares of the integers.

```
10 M=0
20 M=M+1
30 PRINT M;" SQUARED = ";M^2
40 GOTO 20
50 END
```

In this example the computer goes into a never ending loop. The program can be stopped by pressing the <ESCAPE> key. It should be observed that any lines following line 40 would never be executed.

2.1.3 IF...THEN conditional statement

One way in which the previous program could be modified to cause the block from lines 20 to 40 to be executed a finite number of times, followed by execution of any subsequent part of the program, is by means of the

IF...THEN instruction. For example

```
IF M<N THEN GOTO 20
```

This statement allows the computer to take a decision. If the condition following the IF is TRUE, the computer executes the instruction following the THEN, otherwise the computer moves on to the next line. The previous exercise can now be modified by introducing line 5, and altering line 40 as follows

Example 2.3

```
 5 INPUT "NUMBER OF INTEGERS",N
10 M=0
20 M=M+1
30 PRINT M;" SQUARED = ";M^2
40 IF M<N THEN GOTO 20
50 END
```

The program now prints the squares of the first N integers, where N is input from the keyboard when the program is run. Each time the program passes through line 40 the test M<N is carried out. If the condition is TRUE then the program goes to line 20 next, but if the result of the test is FALSE, the program goes on to the next line, line 50.

The IF... THEN construction can be used with intructions other than GOTO. For example

```
IF X<0 THEN PRINT "NEGATIVE"
```

Note, also, that the GOTO can be omitted following the THEN, so that line 40 in the previous exercise could be written as

```
40 IF M<N THEN 20
```

On the BBC computer the THEN can be omitted from any IF...THEN construction. This is not standard BASIC.

String variables can be used in the conditional part of the IF ... THEN statement, as in the following example.

Example 2.4

```
10 INPUT "WHAT IS THE CAPITAL OF FRANCE" ,A$
20 IF A$="PARIS" THEN 60
30 PRINT "WRONG, TRY AGAIN"
40 INPUT A$
50 GOTO 20
60 PRINT "CORRECT"
70 END
```

This program will run until the correct answer is given. In this example exact equality of the strings is necessary to satisfy the test. The greater than (>) and less than (<) signs can be used for conditional tests involving strings. The test A$<B$ is true if the contents of A$ come before the contents of B$ alphabetically.

Exercise 2.1

Modify the program above to permit up to two attempts, after which the correct answer is given

On the BBC microcomputer several statements can be typed on the same line if they are separated by colons (:). In an IF...THEN statement, all the statements on the same line following the THEN will be obeyed if, and only if, the condition tested is true. For example,

```
20 IF A$="PARIS" THEN PRINT "CORRECT": GOTO 70
```

This feature can be very useful, but care must be taken if program lines are concatenated in order to save space, at a later date. Otherwise there is the chance that statements concatenated onto the end of a line containing a THEN may be inadvertantly skipped.

The above programs have used tests for ‘less than’ and ‘equal to'. These are relational operators, and the full range is given in Table 2.1.

Table 2.1 The relational operators.

Symbol	*Meaning*
>	greater than
<	less than
=	equal to
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

The order of the pairs of symbols must be as shown, and with no spaces between them. They can also be used to compare strings, as seen in the Example 2.4. The conditions in an IF statement can be combined with the Boolean operators AND, OR, NOT and EOR. This is fully discussed in the section on logical expressions in Chapter 5.

In the program above, the = sign is used to test for equality; that is, used to mean ‘is equal to'. The results in language form would be YES (TRUE) or NO (FALSE), but as the computer is digital, the result is expressed as a number. On the BBC microcomputer -1 is used to indicate TRUE and 0 is used to indicate FALSE.

2.1.4 FOR...NEXT

A very widely used program construction is the FOR...NEXT loop, that is used if a sequence of processes is to be repeated (or iterated) a number of times. The syntax is

```
FOR J=<lower limit> TO <upper limit> STEP <increment>
...
...   }  other program statements
...
NEXT J
```

where <lower limit>, <upper limit> and <increment> may be numbers or numeric variables or numerical expressions. The STEP <increment> can be omitted, in which case a step size of +1 is assumed.

The FOR...NEXT construction works like this: starting with a FOR statement a variable (J in this case) is given the value of the lower limit. J may then be used for some calculation within the loop. Eventually the computer comes to a line with the statement NEXT (meaning next value of the variable please). This sends the computer back to the FOR statement, increases the variable by the amount of the increment, and checks that the upper limit has not been exceeded, after which the cycle repeats. If the upper limit has been exceeded, the loop is complete, and the program will continue with the statement following NEXT.

To impose any other change than the default increment of +1 on the variable, the FOR statement should finish with STEP <increment>. This can be any real number (positive or negative, integer or fractional) so that the count variable can be incremented or decremented by any amount. For example

`FOR J=1 TO 5`	J will take the values 1, 2, 3, 4, 5
`FOR J=1 TO 10 STEP 2`	J will take the values 1, 3, 5, 7, 9
`FOR J=5 TO 1 STEP -1`	J will take the values 5, 4, 3, 2, 1
`FOR J=1 TO 2 STEP 0.25`	J will take the values 1, 1.25, 1.5, 1.75, 2

A FOR...NEXT loop is always executed at least once in BBC BASIC. The test to see whether J is greater than the upper limit is not carried out until the loop has been executed once. If the ‘increment’ is negative, J will be decreased each time NEXT is reached, and the test will be whether J is less than the ‘upper’ limit or final value. In this case, of course, the final value must be less than the starting value for J. For example

```
FOR J=5 TO 1 STEP -1
```

Would make J count correctly 5, 4, 3, 2, 1 but

```
FOR J=1 TO 5 STEP -1
```

Would only execute the loop once with J=1. J would then be decreased to 0 and the loop would terminate because 0 is less than the final value of 5. This prevents the computer from getting into an infinite loop.

If a fractional value is used for the increment, it should be remembered that simple decimal fractions such as 0.1 are recurring fractions in binary and therefore cannot be represented exactly. This means that a FOR loop such as FOR J=1 TO 5 STEP 0.1 may yield a value of 4.9999999 rather than 5. It is preferable to use integer variables in FOR...NEXT loops when integers are intended.

The program in Example 2.5 below illustrates the use of a FOR...NEXT loop with an increment of 0.1. It will be seen that the two numbers printed are different by a factor of 10 until I=2.4. At that value round off errors affect which element of the A(I) array is printed. (Use <CTRL-N> to select paged mode for the display before running the program.)

Example 2.5

```
10 DIM A(50)
20 FOR I=1 TO 50
30 A(I)=I
40 NEXT
50 FOR 1=1 TO 5 STEP 0.1
60 PRINT I,A(I*10)
70 NEXT
```

It is a useful exercise to change lines 20, 30, 50 and 60 as below

```
20 FOR I%=1 TO 50
30 A(I%)=I%
50 FOR I%=1 TO 50 STEP 1
60 PRINT I%/10,A(I%)
```

In this modified form the program will correctly print the numbers 10 to 50 alongside the numbers 1.0 to 5.0. The use of the resident integer variables A% to Z%, or integer variables, such as VAR%, overcomes any problem that is likely to occur due to round off error in cases where integer values are needed, such as for array subscripts.

The program in Example 2.6 is a summation program in which the sum of the ten prime numbers in the DATA statement is obtained.

Example 2.6

```
10 SUM=0: REM INITIALIZE SUM
20 FOR J%=1 TO 10: REM ITERATION TO FORM SUM
30 READ X: REM OBTAIN PRIME FROM DATA
40 SUM=SUM+X: REM ADD TO PREVIOUS SUMMATION
50 NEXT J%: REM LOOP BACK
60 PRINT "Sum of first 10 primes = ";SUM: REM PRESENT RESULT
70 END
80 DATA 1,2,3,5,7,11,13,17,19,23
```

Exercise 2.2

Write a program to calculate the sum of the first N whole numbers, where N is supplied in response to an INPUT statement by the user. This program should not need a DATA statement.

2.2 STRUCTURED COMMANDS AVAILABLE ON THE BBC MICROCOMPUTER

2.2.1 REPEAT...UNTIL statements

The BBC microcomputer has an alternative loop structure which is not normally available with BASIC. If a sequence of program statements is to be repeated until some condition is satisfied, rather than for a fixed number of times, then the REPEAT and UNTIL statements are appropriate. The sequence of statements is initialized by the REPEAT statement and ends with an UNTIL statement followed by a conditional expression. It is possible to have more than one UNTIL corresponding to the REPEAT provided that only one can possibly be executed. Thus we could have

```
200 IF FLAG=1 THEN UNTIL EPS>0.01 ELSE UNTIL EPS>0.0001
```

One common use of the REPEAT...UNTIL loop is to produce a delay during the operation of the program. This can be done with FOR...NEXT loops but better precision is obtained if the pseudo-variable TIME, available with the BBC microcomputer, is used. TIME is incremented by one every 0.01 second. A delay of 1.5 seconds will be obtained by use of the lines 40 and 50 in Example 2.7.

Example 2.7

```
10 REM Time delay of 1.5 second
20 PRINT "START OF DELAY"
30 START=TIME
40 REPEAT
50 UNTIL START+150>=TIME
60 PRINT "END OF DELAY"
```

An alternative is to change lines 30 and 50 to

```
30 TIME=0
50 UNTIL TIME>=150
```

The former is preferable in case the pseudo-variable TIME is being used for any other timing processes. Notice that in the UNTIL the conditional test is >=. It is written this way so that it is not necessary for the precise value to be satisfied at the instant the TIME variable is accessed. This might not occur due to the time taken to execute BASIC statements particularly if there were more lines between lines 40 and 50.

Exercise 2.3

Write a program to make the computer run a simple ‘MASTERMIND’ game in which the contestant has to guess the number selected by the computer in the range 1 to 10. The program should use a REPEAT...UNTIL loop in its main section. (Use RND(10) to generate the number.)

Exercise 2.4

Extend the last program so that the number selected is in the range 1 to 1000 and the computer gives one of two clues: ‘TOO LARGE’ or ‘TOO SMALL' until the correct number is guessed. (It should be possible to identify the number selected in 10 steps; if this is not achieved the program could be written to comment on the poor performance. Use RND(1000) to choose the number.)

2.2.2 IF...THEN...ELSE... structure

The IF...THEN statement is common to many implementations of BASIC. In BBC BASIC the statement is extended to include the ELSE condition. This construction is a valuable aid to clear, well structured programming in which the effect of statements in the program are immediately clear and it is not necessary to jump to different line numbers to follow the logic of a program. The IF...THEN...ELSE statement takes the form

```
IF <condition> THEN <statement(s)> ELSE <statement(s)>
```

The condition may be complex and contain any of the Boolean statements. For example IF (A>0) AND (A<10) THEN... is valid. The brackets are useful to clarify the order of operations. A simple example of the use of IF...THEN...ELSE is shown in Example 2.8.

Example 2.8

```
10 REPEAT
20 INPUT "A", A
30 IF (A>=0 AND A<=10) THEN PRINT "A in the range 0 to 10"
    ELSE PRINT "A out of range 0 to 10"
40 UNTIL A=999
```

The statements may also be complex within what can be contained in one line of BASIC. (Remember it is possible to compress a line on input by use of the abbreviations P. for PRINT etc.)

A single line of BASIC may contain more than one IF... THEN ...ELSE statement. The individual assignments or statements following THEN or ELSE are separated by a colon in the usual way. The statements used may include calls to functions and procedures. This feature is particularly valuable for clear structured programming and avoids the need for GOTO and GOSUB with incomprehensible line numbers.

In multiple use of IF...THEN...ELSE in a single statement, where there are choices amongst more than two alternatives, it can only make clear logical sense if the order

```
IF...THEN...ELSE IF...THEN..ELSE...
```

is followed. If the sequence

```
IF...THEN IF...THEN...ELSE...ELSE...
```

were to be used it would no longer be clear which test is associated with which ELSE. Where a further IF test is required to follow a THEN, the test can be inverted by use of NOT so that the structure becomes

```
IF NOT <test1> THEN <statement(s)3> ELSE IF <test2> THEN
<statement(s)1> ELSE <statement(s)2>
```

Note that <statement(s)3> can be a null statement if appropriate, leaving the structure as

```
IF NOT <test1> THEN ELSE IF <test2> THEN <statement(s)1> ELSE
<statement(s)2>
```

Example 2.9

The following program illustrates the above points; study the structure then use it.

```
 10 REM IF...THEN...ELSE + FUNCTIONS
 20 MODE 7
 30 PRINT
 40 PRINT "To end run enter E or give a negative"'"angle"
 45 REM NOTE THE USE OF ' TO GIVE A NEW LINE
 50 PRINT
 60 REPEAT
 70 INPUT "Choose function sine, cosine or tangent by entering
    key Letter onLy. ",A$
 80 INPUT "Give angte in degrees ",ANGLE
 90 ANGLE=RAD(ANGLE)
100 IF A$="S" THEN PRINT "RESULT ",SIN(ANGLE) ELSE IF A$="C"
    THEN PRINT"RESULT "COS(ANGLE) ELSE IF A$="T" THEN
    PRINT"RESULT "TAN(ANGLE) ELSE IF A$="E" THEN END ELSE
    PRINT"Key letter not recognized use S, C or T"
110 PRINT
120 UNTIL ANGLE<0
130 END
```

The use of the character ' to give a new line is fully explained in Section 11-4-1. Extensive use can be made of IF...THEN...ELSE but like many good things it can be over indulged and Example 2.10 below, while valid, does not have adequate clarity − ON...GOTO would be more appropriate. Note that the THEN can be omitted but it is not recommended unless space is at a premium. Also, in order to enter the line into the computer the abbreviation P. may have to be used.

Example 2.10

```
 5 REPEAT
10 INPUT "Choose an integer < 7 ",N
20 IF N/2<>INT(N/2) THEN P."Odd number":IF N=1 THEN P."One"
   ELSE IF N=3 THEN P. "Three" ELSE IF N=5 THEN P."Five":ELSE
   P."Even number":IF N=2 THEN P."Two" ELSE IF N=4 THEN
   P."Four" ELSE IF N=6 THEN P."Six" ELSE P. "Number out of
   range."
25 UNTIL N<0
```

Example 3.6 shows a version of the program using ON...GOTO/GOSUB. The use of the ON...GOTO statement is dealt with in Chapter 3.

2.2.3 The absent structured facility WHILE

An alternative construction to REPEAT...UNTIL in some languages is WHILE...ENDWHILE. Unfortunately BBC BASIC does not have this structure, which would have allowed the user to have a section of program that might not be executed at all but which on another occasion could be iterated many times. WHILE provides a facility that is not available with FOR...NEXT and REPEAT...UNTIL loops, which are always executed at least once. In a BASIC with a WHILE structure it is possible to write programs that do not need GOTO statements. The construction of the WHILE structure is

```
100 WHILE <condition>
...  }
...  }  Program statements of loop
...  }
200 ENDWHILE
210 REM NEXT STATEMENT
```

The easiest way to achieve the same effect is

```
100 IF NOT <condition> THEN GOTO 210
110 REPEAT
...  }
...  }  Program statements of loop
...  }
200 UNTIL NOT <condition>
210 REM NEXT STATEMENT
```

2.2.4 The LISTO statements

LISTO, the list option statement, is a non-executable BASIC instruction which is particular to the BBC microcomputer. It affects the manner in which the listing of a program is displayed on the video monitor or on a printer. It causes the computer to insert spaces in three situations either individually or collectively, as follows

1. After a line number.
2. To indent FOR...NEXT loops.
3. To indent REPEAT...UNTIL loops.

The mode of listing selected is indicated by a single number in the range 0 to 7 after the statement issued in immediate mode

```
LISTO 7 <RETURN>
```

(A syntax error occurs if it is used in a program.)

0 implies no inserted spaces
1 implies a space after the line number
2 implies spaces to indent FOR...NEXT loops
4 implies spaces to indent REPEAT...UNTIL loops

If several options are required they can be chosen at one time by adding the key numbers; hence 7 gives all options and zero none.

It should be noted that these are additional spaces to any that are stored as part of the text of the program. It is recommended that LISTO 7 is used to obtain listings which show clearly the structure of a program. If editing is done with LISTO 7 in force then additional spaces become part of the edited program. It is therefore advisable to return to LISTO 0 and relist the program before any editing is done. These LISTO options can conveniently be stored in the function keys by the immediate mode statements

```
*KEY0 LISTO 0 |M |0 <RETURN> and

*KEY1 LISTO 7 |M |N LIST |M
```

which enable the LISTO option to be changed by a single key stroke. The inclusion of <CTRL-N> and LIST in function key fl is convenient since this sequence of commands are often used during the development of a program. The LISTO 0 option is in force after a <BREAK> sequence.

Exercise 2.5

Define the red function keys as suggested above and use them and the other options to LIST some of the example programs.

2.3 ARRAYS

2.3.1 Introduction

It is possible to represent a whole set of values by a single subscripted variable called an array. They are written in the form ARRAY(J) where the name of the set of variables is ARRA Y and J is the subscript that identifies a particular value of the set. Arrays are particularly useful in conjunction with FOR...NEXT loops, since every element of the array can be accessed in turn by treating the control variable of the loop as the array subscript.

The array ARRA Y (J) must be declared once at the beginning of the program, before it is used, by a DIMension statement of the form

```
10 DIM ARRAY(10)
```

This sets aside eleven variables that can be identified by ARRA Y (J) where J is an integer from 0 to 10 (the array subscripts always start from zero).

Example 2.11

The program below, which sums the first ten primes, has been written to illustrate the use of an array in a FOR...NEXT loop, where its use is particularly appropriate. Compare this program with that in Example 2.6.

```
 10 DIM PRIME(10)
 20 SUM=0: REM INITIALIZE SUM
 30 FOR J=1 TO 10: REM ITERATION TO READ DATA
 40 READ PRIME(J): REM OBTAIN PRIME FROM DATA
 50 NEXT J: REM LOOP BACK
 60 FOR J=1 TO 10: REM ITERATION TO FORM SUM
 70 SUM=SUM+PRIME(J): REM ADD TO PREVIOUS SUMMATION
 80 NEXT J: REM LOOP BACK
 90 PRINT "Sum of first 10 primes = ";SUM: REM PRESENT RESULT
100 END
110 DATA 1,2,3,5,7,11,13,17,19,23
```

2.3.2 Multiple-subscript arrays

An array may have more than one subscript; for instance a set of x- and y-coordinates to be plotted on a graph can be stored in the array defined by

```
10 DIM XY(2,20)
```

where XY(1,J) represents the x-coordinates and XY(2,J) represents the y-coordinates. In fact, the array has a set of variables XY(0,J) available since the BBC microcomputer always allocates storage locations for an array from 0 to the values specified in the dimension statement. There is no limit to the number of subscripts an array may have; nor is it necessary to use all the elements, but remember that it can be extremely wasteful of memory to create multiple-subscript arrays which are too large.

For example the program

```
20 DIM A(10,10,10)
30 MODE 1
```

cannot be run successfully on a computer with a disc filing system fitted, with the normal PAGE setting! The error message

```
Bad MODE at line 30
```

is generated when the high resolution four colour graphics mode, which requires 20K of memory, is selected. If a 10×10×10 array is needed remember that the elements with subscript zero always exist, and line 20 should then be

```
20 DIM A(9,9,9)
```

which will enable the two line program to run successfully. In fact another 867 bytes are available for the remainder of the program.

The multiplication table program in Example 2.12 illustrates the use of a two-subscript array to store a table of values. In this case the table stored is the multiplication table from 1×1 to 12×12; a trivial example, which it would be more appropriate to calculate as required, but it shows how a table of values can be stored and accessed. This would be valuable if the data took a long time to calculate, (for example tables of values of sin and cos for a rotation program), or if the data had been obtained from measurements or observations and was needed within the program.

Lines 30 to 100 set up the table and print it out as it is formed. Lines 110 to 130 enable the table to be read, and line 140 permits the execution of the program to be terminated if a zero is entered.

Note that a correct answer is given for a product involving zero as array variables are all set to zero when the DIM statement is executed. Thus array variables can appear on the right hand side of an assignment statement before they have had a value assigned to them, in contrast to simple variables.

Example 2.12

```
 10 REM MULTIPLICATION TABLES
 20 DIM M(12,12)
 30 FOR J=1 TO 12
 40 FOR K=1 TO 12
 50 M(J,K) =J*K
 60 PRINT M(J,K);
 70 NEXT K
 80 PRINT
 90 PRINT
100 NEXT J
110 REPEAT
120 INPUT "Type 2 Numbers Under 13 ",A,B
130 PRINT ;A;" * ";B;" = ";M(A,B): PRINT
140 UNTIL A=0 OR B=0
```

Exercise 2.6

Write a program to read in and then calculate the product of a set of numbers. (Hint: the initial value of the product must be I, in contrast to the starting value of 0 for a sum.)

2.3.3 String arrays

String variables, like real variables, may have subscripts and be expressed in arrays. String arrays are denoted by the addition of the dollar sign, $, to the array variable name. Thus DIM NAME$(7) permits the setting up of an array of eight string variables. As string arrays will have to store strings of variable lengths (up to a maximum of 256 characters), no space is allocated for the string at the time the array is declared. The DIM statement merely allocates space for address pointers to the beginning and end of each string (four bytes for each element of the array). The consequence of the need for a large amount of additional space for the strings themselves, whilst a program is running, is that the computer may run out of room during the execution of the program. This causes the error message

```
No room at line XX
```

to be generated.

For example, the program

```
10 DIM A$(6326)
20 INPUT A$(1)
```

will work if the string given is two characters or less, but will give the error message

```
No room at line 20
```

if the string has three or more characters. After the program has been run it will be found that the command LIST gives the error message ‘No room' This is because memory is still allocated to the program. Typing CLEAR or OLD followed by LIST will give a listing of the program. (The size of the array chosen here is for a Model B microcomputer with DFS and default PAGE setting of &1900. PRINT ~PAGE will confirm the setting of PAGE.)

Clearly, if space is likely to be a problem (in programs using graphics modes, for example), it is advisable to declare only the minimum possible amount of string array space and to use it sparingly. In programs involving a large number of user-friendly prompts it may be advisable to keep the text in DATA statements and only read it when necessary. This means that text is not stored twice as it is when it is assigned to a string variable by an assignment statement. If this ploy does not give enough space then userfriendly material could be stored in a ASCII file for display at appropriate times.

Another memory problem occurs if short strings are later replaced by longer strings which cannot be stored in the same positions in memory. New space is allocated to them further on in the memory but the old space remains unused and is therefore wasted. If this is likely to occur it is better to allocate long strings − for example a string of 20 spaces, STRING$(20," ") − to each element of the array at the beginning of the program, even though this uses a lot of memory. At least, if this does run out of memory the error message will occur right at the beginning of the program and not half way through after a lot of data has been entered.