Chapter 4

STRINGS

4.1 STRING VARIABLES

A string variable enables the programmer to handle strings of alphabetic, numeric and other characters in a way that is very similar to the use of numeric variables for numbers. A string variable is denoted by a final $ sign in the variable name. String variables such as A$, NAME$, Text$ etc. may have character strings assigned to them in the same way that numbers are assigned to ordinary variables, except that in assignment statements the character string must be enclosed in quotation marks. Like numeric variables, string variables must be assigned before they are used. The PRINT and INPUT commands can be used for string variables.

Simple string variables are very useful in making programs ‘userfriendly’. For example, the following exercise uses a string variable to store the user’s name.

Example 4.1

```
 10 CLS
 20 PRINT "Please give your name"
 30 INPUT NAME$
 40 PRINT "If you trant help, ";NAME$;" type 1, othertwise type
    2"
 50 INPUT X
 60 IF X=1 THEN PROChelp
 70 REM
 80 REM FURTHER SECTIONS OF PROGRAM WOULD BE INSERTED HERE
 90 REM
100 PRINT "Please give your age, ";NAME$
110 INPUT AGE
    ...
    ...
    ...
    ...
```

```
 990 END
1000 DEF PROC_help
1010 PRINT "Help information"
1020 REM HELP INSTRUCTIONS WOULD BE
1030 REM INSERTED HERE
1070 PRINT "Press any key to continue"
1080 B$=GET$
1090 ENDPROC
```

4.2 PRINTING WITH STRINGS

4.2.1 INPUT

After the INPUT command, one or more strings of text between quotation marks can be printed as a prompt before any of the variables to be input. Each string of text and final quotation mark must be followed by a comma. Note that a string variable *cannot* be used instead of a string in this context, since it would be interpreted as a variable to be input. (In BASIC II *only*, semicolons may also be used as separators, both after the strings of text and between variables. The latter option makes BBC BASIC compatible with 'standard' BASIC, where a semicolon is used after a string of text, and commas between variables.)

Exercise 4.1

Modify Example 4.1 as follows

```
100 INPUT "Please type your age ",AGE
110 PRINT "Thank you ";NAME$
```

The data is entered on the same line as the prompt printed in the INPUT statement, which leads to greater clarity in programs.

The same effect can be achieved by typing a semi-colon at the end of the PRINT statement, so that the cursor doesn't move on to a new line.

Exercise 4.2

Modify Exercise 4.1 as follows

```
 95 C$=" please give your age "
100 PRINT NAME$+C$;
105 INPUT AGE
```

The question mark printed on the screen when an input is wanted can be unnecessary and irritating if you are using prompts. It can be suppressed by leaving out the comma after the string at the start of the INPUT statement. This can be done even with an 'empty' string, so that Exercise 4.2 could be modified to

```
105 INPUT ""AGE
```

4.2.2 Concatenation of strings

As demonstrated above, strings can of course be concatenated, or 'added' together in a PRINT statement. It is also possible to 'add' strings within a program and assign new values to a string by statements such as

```
X$=A$+B$+C$
```

Note, however, that no other arithmetic operations (not even subtraction) are possible with string variables.

Exercise 4.3

To demonstrate this, modify Exercise 4.1 again as follows

```
 95 C$=NAME$+" please give your age"
100 PRINT C$;
```

4.2.3 INPUT/READ statements for string variables

In assignment statements for strings, it is essential to use quotation marks, for example

```
A$="A word"
```

In an INPUT or READ statement the quotation marks are optional so long as it is not necessary to input a comma or semicolon. With the READ statement, if leading spaces, commas or semicolons are to be read in, it is essential for the whole string to be enclosed in quotation marks. On the other hand, quotation marks can be included in strings that start with some other character than a quotation mark (single inverted commas can be freely input in READ or INPUT responses). It is actually possible to input a quotation mark anywhere within a true string by typing a pair of quotation marks.

The same technique can be used with the INPUT statement, but there is an easier alternative. An extra item can be included within the items after INPUT. This is the word LINE, which must appear at the start of the INPUT statement; its effect is to take the whole line of input that is given in response to the statement and put it into the next variable. This gives a simple means of leading spaces, commas and quotation marks anywhere in a string. Obviously LINE is only useful in connection with string variables.

The following statement would be quite legitimate in BBC BASIC

```
INPUT LINE,"Length of strings",L,"First string",A$, "Second
  string",B$
```

Exercise 4.4

Verify the statements above by experimenting with different responses to the following program

```
10 CLS
20 INPUT LINE,X$,Y$,Z$
30 PRNIT X$,Y$,Z$
```

Modify the program to use READ and DATA statements

```
20 READ X$,Y$,Z$
40 DKTA "WHY, OH WHY",SAY "AH","","""AH""": REM TEST DATA
```

4.3 COMPARISON OF STRING VARIABLES

It is possible to compare string variables for equality in an IF statement. The result will be TRUE only if there is complete equality − a character by character correspondence between two strings of equal length. It is also possible to test for inequality (<>) and greater than (>) or less than (<).

Example 4.2

The following section of program shows a test for the answer to a question.

```
10 CLS
20 Y$= "YES"
30 INPUT "Do you want to continue? "YN$
40 IF YN$="NO" THEN PRINT"Program ended": END
50 IF YN$<>Y$ THEN PRINT "Please type YES or NO": GOTO 30
60 PRINT "Program continued"
70 REM FURTHER SECTIONS OF PROGRAM WOULD GO HERE
```

4.3.1 Alphanumeric ordering

String variables can be sorted into alphanumeric order by use of IF statements of the form

```
IF A$<=B$ THEN...
```

The IF test will be TRUE if a character by character comparison shows that A$ would come before B$ lexicographically. (In this case the lexicographical order is determined by means of a comparison of the ASCII codes of the characters − see Appendix H.)

Lexicographic order is basically the order in which words would appear in a dictionary, but including other types of character as well as letters. In particular, numbers come *before* letters, and most other symbols such as +, * and / come before numbers. Thus A+ comes before A1, which comes before AA. Note that a lexicographical comparison of numbers is quite different from a numeric comparison. "19" comes before "9".

Example 4.3

The next program uses this idea to sort two words into alphabetical order.

```
 10 REM ALPHANUMERIC SORT
 20 T$="Alphanumeric order is: "
 30 CLS
 40 INPUT "Type two wods: "A$,B$
 50 PRINT T$
 60 REPEAT
 70   X$=A$
 80   A$=B$
 90   B$=X$
100 UNTIL A$<=B$
110 PRINT A$,B$
```

X$ is an extra string variable used to hold the contents of A$ temporarily during the swap.

Exercise 4.5

Modify Example 4.3 to sort three words instead of two by adding the following lines

```
 40 INPUT "Type three words: "A$,B$,C$
 65 REPEAT
110 X$=B$
120 B$=C$
130 C$=X$
140 UNTIL B$<=C$ AND A$<=B$
150 PRINT A$,B$,C$
```

When you have done these modifications, LIST the program to see how it works. Try it with data that is already in the correct order, e.g. P,Q,R; in reverse order, e.g. Z,Y,X; and in mixed order, e.g. N,L,M.

4.4 STRING MANIPULATION FUNCTIONS

4.4.1 String analysis

BASIC contains three useful functions to break large strings down into smaller ones. These functions are of value when only part of a string is actually needed for a particular task or if extra characters need to be inserted inito a string.

To make proper use of these three functions it is frequently necessary to know the number of characters in the original string. Since the programmer not know the length of a string in advance, it is convenient to mention at this point a further function which counts the number of characters in a string. LEN(A$) returns the number of characters in the string A$. Try the following simple program which will print out the number of characters in the user's name.

Example 4.4

```
10 PRINT "Please type your name"
20 INPUT NAME$
30 NUM=LEN(NAME$)
40 PRINT "There are ";NUM;" characters in your name"
50 PRINT "Thank you ";NAME$
```

Two of the functions for breaking down strings are rather similar. LEFT$(A$,N) returns the first (leftmost) N characters of the string A$. RIGHT$(A$,N) returns the last (rightmost) N characters of A$. The functions are used when the characters at the beginning or end of the string are needed.

Example 4.5

A simple example of their use which identifies the first and last characters in the user's name is as follows

```
10 PRINT "Please type your name"
20 INPUT NAME$
30 FIRST$=LEFT$(NAME$,1)
40 LAST$=RIGHT$(NAME$,1)
50 PRINT "The first character in your name is ";FIRST$
60 PRINT "The last character in your name is ";LAST$
70 PRINT "Thank you ";NAME$
```

The third function will select *any* chosen part of a string. MID$(A$,N1,N2) returns N2 characters of the string A$, beginning with the Nlth character. MID$ is a more versatile function than LEFT$ or RIGHT$ and will in fact do all that the others can do. For example, if A$="INTENDED", then MID$(A$,4,3) will return "END".

MID$ can also be used with only two parameters to obtain the right-hand gnd of a string. If N2 is omitted, MID$(A$,N1) will return all characters from the N1th position to the end of the string. This can sometimes be simpler to use than RIGHT$, which needs the length of the remaining string supplied. For example, with A$ as above, MID$(A$,4) and RIGHT$(A$,5) will both return "ENDED".

When strings have been subdivided with these functions it is frequently necessary to reassemble them or to add to them which can be done by simply adding strings tbgether.

Example 4.6

This is a simple example of the use of MID$ is to search a string for the occurrence of a second string (as we shall see below, this facility is provided more directly by INSTR).

```
10 CLS
20 PRINT "Type the string to be processed"
30 INPUT A$
40 PRINT '"Type the string to be searched for"
```

```
 50 INPUT B$
 60 J=0
 70 REPEAT
 80 J=J+1
 90 IF MID$(A$,J ,LEN(B$))=B$ THEN PRINT "String found at
    position: ";J
100 UNTIL J = LEN(A$)-LEN(B$)+1
```

Note the use of the apostrophe in a single PRINT statement instead of repeated PRINTs to create multiple new lines in this and the following programs.

Example 4.7

As a more extensive example of the use of MID$, try the following program, which provides a search and replace facility for a user-supplied string. This is a primitive version of the system used in word processors.

```
10 CLS
20 PRINT "Type the string to be processed"
30 INPUT A$
40 PRINT '"Type the string to be searched for"
50 INPUT B$
60 PRINT '"Type the replacement string"
70 INPUT C$
80 C=0
90 L=LEN(A$)
100 DIM K(L)
110 FOR J=1 TO L-LEN(B$)+1
120 IF MID$(A$,J,LEN(B$))=B$ THEN C=C+1: K(C)=J
130 NEXT J
140 F$=A$: REM F$ WILL BE NEW STRING
150 FOR J=1 TO C
160 L=K(J)-1+LEN(F$)-LEN(A$)
170 L$="": IF L>0 THEN L$=LEFT$(F$,L)
180 L=LEN(A$)-LEN(B$)-K(J)+1
190 R$="": IF L>0 THEN R$=RIGHT$(F$,L)
200 F$=L$+C$+R$
210 NEXT J
220 PRINT '"There are ";C;" occurrences of ";B$;" in"
230 PRINT A$
240 IF C=0 THEN END
250 PRINT '"The new string is: "
260 PRINT F$
```

The actual search is carried out in line 130 as in the previous example. K(J) stores the positions of each occurrence and C counts their number. The new string is created at line 210, by concatenating the left and right hand parts of

the original string, obtained by LEFT$ and RIGHT$, and the new string, C$.

Exercise 4.6

Write a program to input a string, divide it into two equal (or nearly equal) halves, and to insert the string " EXTRA " in between the two halves.

Exercise 4.7

Write a program to input a string and then to print it out with the order of the characters reversed.

BBC BASIC contains two further functions not commonly found, which extend the versatility of the BBC microcomputer for manipulating strings.

INSTR provides a direct search facility of the type carried out in Exercises 4.6 and 4.7 using MID$.

INSTR(A$,B$) returns the position in the string A$ of the first occurrence of the string B$. If no match is found, INSTR will return a zero. If B$ is an empty string, INSTR will always return a 1.

An alternative form of INSTR is INSTR(A$,B$,N) which starts the search from the Nth character of A$. For example, with N=2 the search would omit the initial letter, which could be useful if, say, you were looking for a word which might or might not start with a capital letter.

Exercise 4.8

Try modifying the program of Example 4.7 to use INSTR by the following changes

```
 90 L=1
100 DIM K(LEN(A$)+1)
110 REPEAT
120 J=INSTR (A$,B$,L): C=C+1: K(C)=J: L=J+1
130 UNTIL J=0
135 C=C-1: REM REPEAT LOOP COUNTS ONE TOO MANY
```

Note that there is a bug in INSTR in BASIC I (documented in the BBC *User Guide*) whereby the stack is corrupted if B$ is longer than A$. The bug is actually worse than this, in fact. Where the form INSTR(A$,B$,L) is used, the bug occurs if the *remaining portion* of A$ from L onwards is smaller than B$ (even if the whole of A$ is longer than B$). The statement that ‘the stack is corrupted’ is likely to be meaningless to a beginner, and probably will not explain clearly even to the experienced programmer what may go wrong. The result can actually be a very subtle flaw in program execution. The program modification above *ought* to work in all cases, but in fact under BASIC I the program will crash if B$ occurs at the very end of A$ (for example, if searching for "END" in "DIVIDEND"). What happens is that, on the pass of the REPEAT loop following the one in which "END" is deleted (when L=6), we have L=7 and the INSTR only examines the portion "ND" of "DIVIDEND", which is shorter than B$. This ought to return a zero into the variable J, but because of the stack corruption this does not happen, and the REPEAT loop ploughs on until the program crashes. The bug has been cured

in BASIC II.

Exercise 4.9

Modify Exercise 4.8 to cater for the BASIC I bug in INSTR.

The second extra string handling function in BBC BASIC is STRING$ (not to be confused with STR$), which will produce multiple copies of another string. Thus

```
X$=STRING$(N,A$)
```

is equivalent to

```
X$=""
FOR J=1 TO N
X$= X$+A$
NEXT J
```

Its main use is likely to be in producing decorative features, such as strings of stars. For example

```
PRINT STRING$(40,"*");
```

will produce a complete line of stars. Similarly

```
PRINT STRING$(20-INT(LEN(A$)/2)," ");A$
```

will print A$ centred on a (40 character) screen line.

4.4.2 String conversion functions

There are four other special functions which relate to strings or string Variables. The first two are a complementary pair which enable the programmer to change from a number to a string and vice versa. STR$(expr) Converts a number or numerical expression into a string, for example

```
TEN$=STR$(10) or NUM$=STR$(NUM) VAL(A$)
```

performs the reverse process, finding the numerical value of a string, as in

```
TEN=VAL("10") or NUM=VAL(NUM$)
```

It follows therefore that VAL(STR$(A))=A and STR$(VAL(A$))=A$ (if A$ represents a number).

Exercise 4.10

Try

`A=VAL(STR$(45))` then `PRINT A`

The function VAL may not produce a sensible result if the string does not in represent a number. The action of VAL in this case is to take any numbers in the string up to the first invalid character. If the first non-zero character is invalid V AL returns a value of zero. Thus

```
VAL("ABC")
```
gives `0`
```
VAL("000XYZ")
```
gives `0`
```
VAL("12 BUCKLE MY SHOE")
```
gives `12`

But note

```
VAL("1.5E3")
```
gives `1500`

Try PRINTing these functions.

Exercise 4.11

Write a program to input a real number and determine whether it is of fixed point or exponent form (turn it into a string with STR$ and use LEN and MID$ to scan each character for a letter E). If it is in fixed point, set a new variable to the fractional part of the number (the part to the right of the decimal point) and then print the new variable. (Hint: find the position of the decimal point, discard the characters to the left with MID$ or RIGHT$ and finally form the numerical value with VAL).

1. Note that it is possible to generate the hexadecimal form of the number (see Section 10.1) by using a tilde before the brackets after STR$, for example STR$~(30).
2. Note also that there is a minor quirk with STR$: unless you alter the value of @% (see Section 11.4), the variable is converted to a string to full machine accuracy, whereas when the variable is PRINTed, it is rounded by one digit, which takes care of most rounding errors. Printing the STR$ of the variable will include any such rounding errors, of which you will not normally be aware.

Exercise 4.12

Try the following

```
A=2.1/10: B=3.1/10: C=3.2/10
PRINT STR$(A),STR$(B),STR$(C)
```

Try also printing values in hexadecimal, for example

```
PRINT STR$~(30)
```

To understand the last two functions we must first examine the way in which characters are read from the computer keyboard and stored in memory. There is a widely used standard code called ASCII (American Standard Code for Information Interchange) which allocates unique codes to each keyboard

character and also other kevs such as carriage return, escape, control draracters and so on. The full list of these codes is given in Appendix H. (Note that BASIC programs are not stored completely as ASCII codes, but in a more complicated fashion, with whole keywords stored as a single code, for instance.)

There are times when it is necessary to use ASCII codes to output certain characters, when interfacing the computer to some peripheral devices for example

The function CHR$(N) produces the character whose ASCII code is N, and the function ASC(A$) returns the ASCII code of the first character of the string A$.

Exercise 4.13

For example, try

`PRINT CHR$(34)` − outputs a quotation mark

`PRINT CHR$(91) and PRINT CHR$(93)` − output opening and closing square brackets.

Try also printing the characters with ASCII codes 7, 8, 10 and 12.

The ASC function is less used, but acts as the complement of CHR$, so check that ASC(CHR$(45))=45 and CHR$(ASC("M"))=M. (But note CHR$(ASC("LETTER"))=L.)

Exercise 4.14

1. Write a program to input any character and print its ASCII code.
2. Write a program to input an ASCII code and print the corresponding character.
3. Write a program to tabulate every ASCII code from 33 to 255 and the corresponding character.

On the BBC microcomputer, an alternative method of PRINTing characters is provided. This is via the VDU command.

```
VDU M, N,...
```

is exactly equivalent to

```
PRINT CHR$(M) ;CHR$(N);...
```

(except that you cannot have commands of the form VDU M,N,"STRING").

Thus if you want to PRINT several ASCII characters, and nothing else, it would be quicker to use VDU. For example

```
VDU 7,7,7
```

Would give a triple length bell ring.

VDU is of course normally used to issue special screen control codes. For instance, VDU 12 clears the screen. So does PRINT CHR$(12), therefore. By the same token, where a VDU command needs extra parameters, as in the case of VDU 19, trying to PRINT CHR$(19) will hang the computer, apparently inexplicably, until a few keys have been pressed.

In Mode 7 it is necessary from within a program to use either VDU or CHR$ to generate the special effect codes such as for colour and double-height text.

4.5 STRING ARRAYS

String variables, like real and integer variables, may have subscripts — they can be arrays − with the same limitation on the number of subscripts and default subscript sizes as for other arrays.

String arrays are extremely useful in many circumstances. For instance, Exercise 4.7 could be accomplished by loading the string one character at a time into a string array, then printing the array backwards in a decrementing loop

```
FOR J=LEN(A$) TO 1 STEP -1: PRINT A$(J);: NEXT J
```

where A$ is the original string.

Exercise 4.15

Repeat Exercise 4.7 using a string array.

With large arrays of strings, memory size can easily become a problem. This will not be apparent when the string array is DIMensioned, since memory is only used when strings are actually loaded into the string variable array elements (apart from an overhead of 4 bytes per array element).

Example 4.8

Consider the following program which simply keeps adding an extra character to each element of the array A$ to form the next element.

```
10 MODE 7
20 DIM A$(250)
30 A$(0)=""
40 FOR N=1 TO 250
50 A$(N)=A$(N-1)+CHR$(N)
60 PRINT ;N;" STRINGS STORED"
70 NEXT N
80 PRINT "PROGRAM ENDED"
```

If you run this program, you will see that it fails with error message ‘No room’, after approximately 220 array elements have been written to. Since each arrav element has as manv characters as its array index, this corresponds

to a total memory requirement of approximately $0.5 \times 220^2 = 24K$, which is indeed most of the available memory.

Example 4.9

The program below shows the usefulness of string arrays. It is a program to display any message you choose with an attractive border of stars, and works by setting up a two-dimensional string array with stars all round the edge and your message nicely positioned in the middle. It uses several of the string facilities we have been discussing.

Lines 130 to 200 show how easy it is to draw the border with string arrays, while the real heart of the program is line 450 which puts the message into the array, after sorting out all the centring problems, and line 500 which prints the display onto the screen.

```
 10 SCRHT=25: SCRLEN=40
 20 REM IF NOT IN MODE 7, ALTER APPROPRIATELY
 30 DIM A$(SCRHT-3,SCRLEN)
 40 DIM B$(SCRHT-5)
 50 CLS
 60 PRINT "Initializing array...."
 70 FOR J=1 TO SCRLEN
 80 FOR K=1 TO SCRHT-3
 90 A$(K,J)=" "
100 REM INITIALIZE ALL ARRAY ELEMENTS TO SPACES
110 NEXT K
120 NEXT J
130 FOR J=1 TO SCRLEN
140 A$(1,J )="*"
150 A$(SCRHT-3,J )="*"
160 NEXT J
170 FOR K=1 TO SCRHT-3
180 A$(K,1)="*"
190 A$(K,SCRLEN)="*"
200 NEXT K
210 REM GENERATE BORDER OF STARS
220 CLS
230 PRINT "Type in a message, RETURNing at the"
240 PRINT "end of each line (maximum ";SCRHT-6;" lines)."
250 PRINT "When finished, just type <RETURN>."
260 K=0
265 REPEAT
270 K=K+1
280 INPUT B$(K)
290 UNTIL B$(K)="" OR K>=SCRHT-5
295 IF B$(K)="" AND K>1 THEN K=K-1
300 CLS
310 IF K>INT((SCRHT-6)/2) THEN GOTO 380 320 FOR J=K TO 1 STEP -1
```

```
320 FOR J=K TO 1 STEP -1
330 B$(2*J-1)=B$(J)
340 B$(2*J-2)=" "
350 NEXT J
360 K=2*K-1
370 REM PUT BLANK LINE BETWEEN EACH LINE OF MESSAGE
380 L1=INT((SCRHT-5-K)/2)
390 REM L1 IS NUMBER OF BLANK LINES ABOVE MESSAGE
400 FOR KK=1 TO K
410 L=LEN(B$(KK))
420 L2=INT((SCRLEN-2-L)/2)
430 REM L2 IS NUMBER OF SPACES TO LEFT OF MESSAGE LINE
440 FOR J=L2+1 TO L2+L
450 A$(KK+L1+1,J+1)=MID$(B$(KK),J-L2,1)
460 NEXT J
470 NEXT KK
480 FOR K=1 TO SCRHT-3
490 FOR J=1 TO SCRLEN
500 PRINT A$(K,J);
510 NEXT J
520 NEXT K
```

4.6 FURTHER STRING HANDLING

String handling is probably the most important and useful non-mathematical use of the computer. The following program is a very simple word processor, which takes a continuously typed passage of text and displays it without words being ‘wrapped round’. (The version on the supplementary disc contains lines 30 to 190 and 460, which are not printed below, simply to give the user an option to run a prepared passage instead of typing one in.)

Example 4.10

```
 10 DIM T$(800)
 20 CLS
200 PRINT "Type a piece of text, ending with"'"<RETURN>"'
210 L=0
220 REPEAT
230 L=L+1
240 T$(L)=GET$
250 PRINT T$(L);
260 UNTIL T$(L)=CHR$(13)
280 PRINT '"Press any key to print"
290 PRINT "this text properly"'
300 A$=GET$
320 K=0
330 REPEAT
```

```
340 J=42
350 REPEAT
360 J=J-1
370 UNTIL T$(J+K)=" " OR T$(J+K)="" OR (T$(J+K)="-" AND J<41) OR
J=0
380 IF J=0 THEN J=40: REM WORD TOO LONG TO FIT
390 FOR M=1 TO J-1
400 PRINT T$(M+K);
410 NEXT M
420 IF T$(J+K)<>" " THEN PRINT T$(J+K);
430 IF J<40 OR T$(40+K)=" " OR T$(40+K)="" THEN PRINT
440 K=J+K
450 UNTIL K>=L
```

In this program, lines 340 to 370 are the key lines. Forty-one characters from the string of text are searched starting from the right-hand end until a space or hyphen is found at which the string can sensibly be broken. Line 440 then sets a counter to the start of the next 41 characters to be searched.

Another very common requirement is a program to sort strings. For instance, a class list may be stored as DATA statements, to which marks may be added, and then the list may be required to be sorted into mark order. There are many ways of sorting, but a simple and reasonably efficient method is the bubble sort. This method makes a series of passes through the list to be sorted, comparing each pair of entries in turn and reversing them if necessary. This means that the largest number in the list rises like a bubble to the end of the list.

The program remembers which numbers have reached their right place at the end of the list, so that each pass has less to sort than the previous one. The sort is complete when no interchanges are made during a pass. (In the program below, this is detected on line 210 when FLAG has the value 0.)

Example 4.11

The program below, which is intended to demonstrate how a simple bubble sort works, sorts strings up to 3 characters long, and shows each step of the process.

```
 10 REM BUBBLE SORT
 20 CLS
 30 PRINT "DEMONSTRATION BUBBLE SORT": PRINT
 40 INPUT "How many words? "N
 50 NN=N
 60 DIM A$(N): REM IN THIS EXAMPLE A STRING ARRAY IS USED
 70 PRINT '"Please type the words to be sorted,"
 80 PRINT "one to a line"'
 90 FOR I=1 TO N
100 INPUT A$(I)
110 NEXT I
120 PRINT ': PROC_printwords(N,NN)
130 REM SORT THE LIST
```

```
 140 REPEAT
 150 FLAG=0 : N=N-1
 160 FOR K=1 TO N
 170 REM COMPARE TWO WORDS
 180 IF A$(K)>A$(K+1) THEN PROC_swap(K)
 190 NEXT K
 200 PROC_printwords(N,NN)
 210 UNTIL FLAG=0
 220 PRINT '"The list is now sorted"
 230 END
1000 DEF PROC_printwords(N,NN)
1010 LOCAL I
1020 FOR I=1 TO NN
1030 PRINT A$(I);TAB(4*I);
1040 NEXT I
1050 PRINT 'TAB(4*N);
1060 FOR I=4*N TO 4*NN: PRINT "-";: NEXT I
1070 PRINT
1080 TIME=0: REPEAT: UNTIL TIME=500
1090 ENDPROC
2000 DEF PROC_swap(K)
2010 LOCAL X$
2020 X$=A$(K) : A$(K) =A$(K+1): A$(K+1)=X$
2030 FLAG=1
2040 PRINT TAB(K*4-4);A$(K);TAB(K*4);A$(K+1)
2050 TIME=0: REPEAT: UNTIL TIME=300
2060 ENDPROC
```

Try altering this program to sort words of any length, without the delay involved in the demonstration subroutine, by deleting lines 120, 200, 2040, 2050 and 1000 to 1080 to remove the illustrative procedure, and adding

```
225 PRINT '': FOR I=1 TO NN: PRINT A$(I): NEXT I
```

to PRINT the sorted list. This will show the speed of the sorting process. The program can also be used to sort numbers, if A$() and X$ are replaced by A() and X throughout.

Example 4.12

The next program uses the same bubble sort to carry out the class list problem. Ten student names are stored as DATA. The user must assign marks to each, and then the program lists the students by mark order.

The actual sort is carried out in lines 90 to 180 of the program.

```
  5 CLS
 10 DIM ST$(10),MARK(10),X(2)
 15 PRINT "Enter the mark for each student"'
```

```
  20 FOR J=1 TO 10
  30 READ ST$(J)
  40 PRINT ST$(J);TAB(10);"Mark - ";
  50 INPUT MARK(J)
  60 NEXT J
  70 N=10
  80 NN=N
  90 REPEAT
 100 FLAG=0
 110 N=N-1
 120 K=1
 130 REM COMPARE TWO MARKS
 140 REPEAT
 150 IF MARK(K)<MARK(K+1) THEN PROC_swap(K)
 160 K=K+1
 170 UNTIL K>N
 180 UNTIL FLAG=0
 190 PRINT '"Student        Mark"'
 200 FOR J=1 TO 10
 210 PRINT ST$(J);TAB(15);MARK(J)
 220 NEXT J
 230 DATA Adams,Black,Bloggs,Brown,Green
 240 DATA Johnson,Jones,Smith,Thompson,White
 250 END
1000 DEF PROC_swap(K)
1010 LOCAL X$,X
1020 X$=ST$(K): X=MARK(K)
1030 ST$(K)=ST$(K+1): MARK(K)=MARK(K+1)
1040 ST$(K+1)=X$: MARK(K+1)=X
1050 FLAG=1
1060 ENDPROC
```