PCN
micropaedia
Vol 11 — Part 3

BUGS BEATEN AND BANISHED
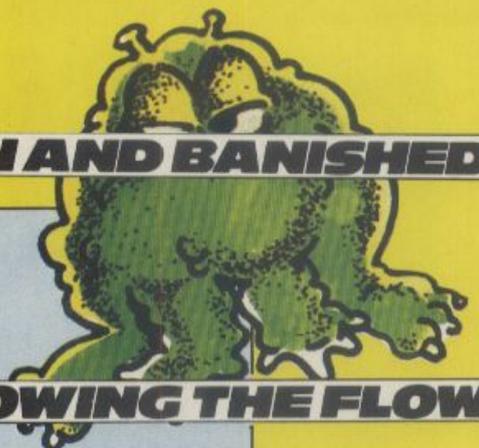
FOLLOWING THE FLOW
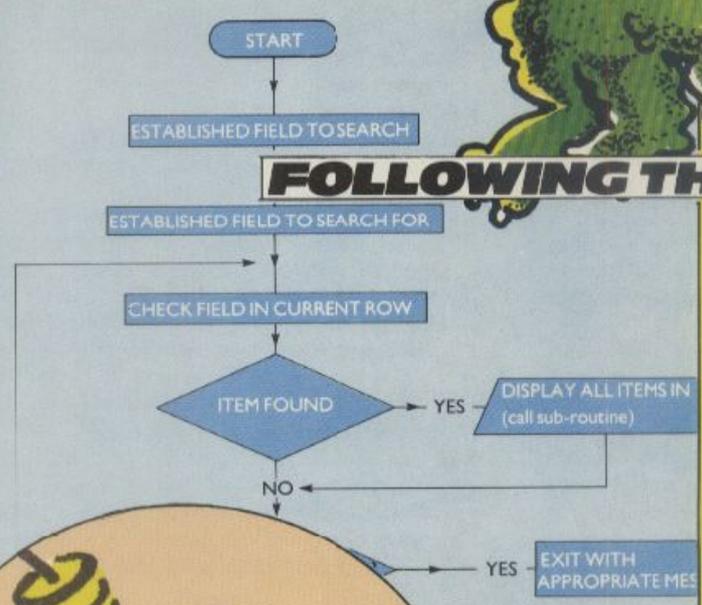
In Part 3 of Everything You Always Wanted to Know About Programming... But Were Afraid to Ask, we introduce you to arrays, show you how to follow flowcharts and add to your sketch code skills.

START

ESTABLISHED FIELD TO SEARCH

ESTABLISHED FIELD TO SEARCH FOR

CHECK FIELD IN CURRENT ROW

ITEM FOUND — YES → DISPLAY ALL ITEMS IN (call sub-routine)

NO

YES → EXIT WITH APPROPRIATE MES

SPACESHIPS AND SKETCH CODE

TWO WAYS TO ARRAY
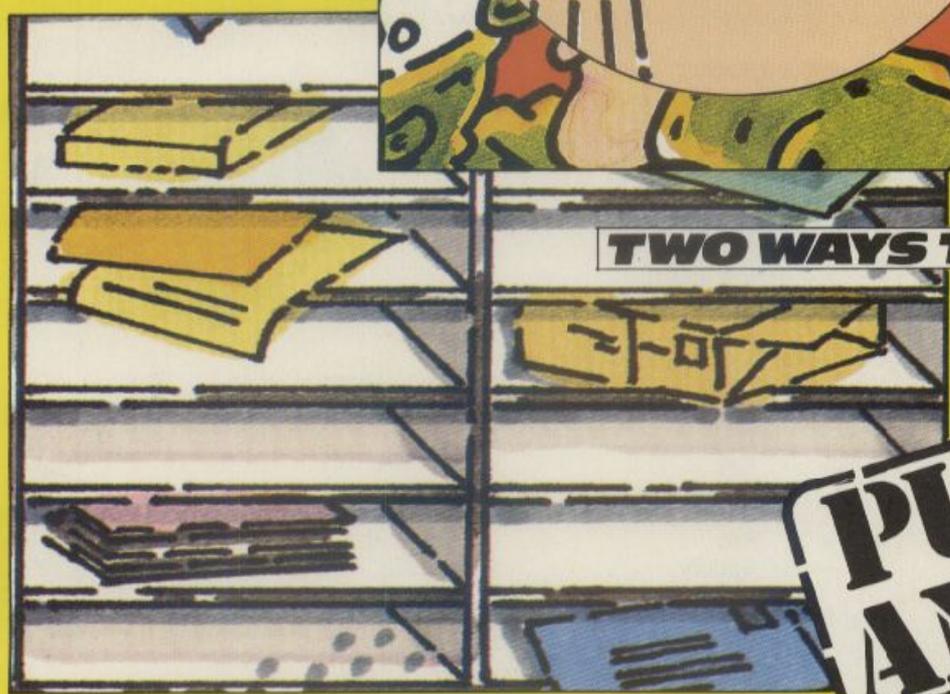
PULL OUT AND KEEP

ACTIONS AND OBJECTS

Last week you saw how the English language could be used to describe in outline the events in a program. This week we discuss the similarities between the actions (verbs) in a program and the objects (nouns).

We'll use the same type of 'sketch code' programming as last week to describe the elements that make up this form of design.

Something not often realised about so-called 'natural' languages is that we draw an artificial distinction between actions and objects. A distinction which is all the more absolute by virtue of its being unmentioned.

Considered as 'events', both actions and objects are identical. That is, as time passes, they 'occur'.

Since we have shown that a program may be quite adequately described in a natural language (English in this case), it is not impossible to apply identical rules to data. If you apply this principle to data-structures what can you do, if you use the best, most elegant, and correct grammar to describe it?

Well, look at this, for example ... a restatement of the job originally given. First we will separate everything which is concerned with describing data-objects into one place, leaving all the actions until later.

Cobol programmers will recognise the technique, it's the old DATA DIVISION again. Pascal, PL/1, Ada and C programmers will recognise it as a declaration, but Basic has never heard of the idea.
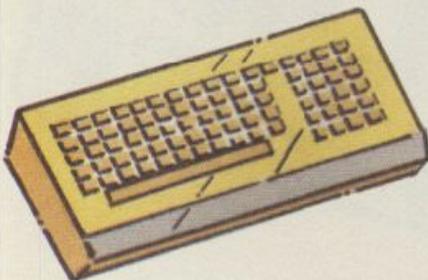
LOGIC OF THE LANGUAGE

Begin:
    There are things called:
        Object, Type,
    and lists called:
        Position, with two values (X, and Y),
        Application, with three values (Game, Mouse, Editor),
        Ikons, with some values (empty),
        Cursors, with four values (Flashing, Solid, Underline, Query),
        Things Like Spaceships, with some values (empty).
        Channel, with four values (Keyboard, JoystickX, JoystickY, Mouse),
        Reply, with four values (Key, JoyX, JoyY, Squeak),
        Move which has four lists called Directions (Up, Down, Left, Right), each being two lists called
            Key and Squeak, each being lists with three parts (("Q", "W" and "E"), ("E", "D" and "C"), ("C",
            "X" and 'Z') and ("Z", "A" and "Q")), and (8, 1 and 2), (2, 3 and 4), (4, 5 and 6) and (6, 7 and 8)),
            and a number called Bias value (−1, +1, −1, +1).
/* This could be written more clearly as:

```
Move which has
    four lists called Directions (Up, Down, Left, Right),
        each being two lists called Key and Squeak,
            each being lists with three parts      (("Q", "W" and "E"),
                                                    ("E", "D" and "C"),
                                                    ("C", "X" and "Z") and
                                                    ("Z", "A" and "Q")),
                            and                     ((8, 1 and 2),
                                                    (2, 3 and 4),
                                                    (4, 5 and 6) and
                                                    (6, 7 and 8)),
    and a number called Bias value (−1, +1, −1, +1).
```

or maybe:

Move which has four lists each being two lists, (each of which are lists with three parts), and a number, called Directions ((Up, Down, Left, Right), (Key (("Q", "W" and "E"), ("E", "D" and "C"), ("C", "X" and "Z") and ("Z", "A" and "Q")), and (Squeak, ((8, 1 and 2), (2, 3 and 4), (4, 5 and 6) and (6, 7 and 8)), and Bias (−1, +1, −1, +1).

which would be written more clearly as:

Move which has four lists each being two lists, (each of which are lists with three parts), and a number, called
Directions ((Up, Down, Left, Right),
        (Key   (("Q", "W" and "E"),
                       ("E", "D" and "C"),
                       ("C", "X" and "Z") and
                       ("Z", "A" and "Q")),
   and (Squeak,   ((8, 1 and 2),
   (2, 3 and 4),
                       (4, 5 and 6) and
                       (6, 7 and 8),
        and Bias   (−1, +1, −1, +1).

both of which are effectively the same as:

Move	Up	Key	("Q", "W" and "E")
		Squeak	(8, 1 and 2)
		Bias	−1
	Down	Key	("E", "D" and "C")
		Squeak	(2, 3 and 4)
		Bias	+1
	Left	Key	("C", "X" and "Z")
		Squeak	(4, 5 and 6)
		Bias	−1
	Right	Key	("Z", "A" and "Q")
		Squeak	(6, 7 and 8)
		Bias	+1

and carrying on with the code
*/
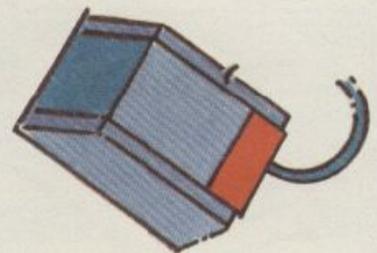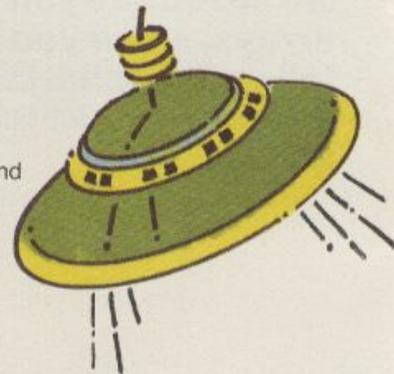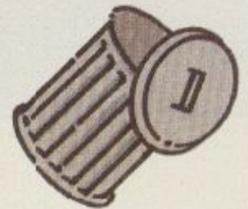
Select the Type from the list Applications
   If Type is a Game: the Object is in the list ThingsLikeSpaceships
   If Type is a Mouse: the Object is in the list Icons
and default to: the Object is in the list Cursors

loop around is here:
   go to the point given by positions (X and Y) and display the Object. For the number of Channels,
      Reply (number) is what's found in Channel (number).
   If Reply (Key) is "S", repeatedly, Reply (Key) is what's on the keyboard until Reply (Key) is "S" /*
      Loop until "S" is pressed again */
   If the Reply (Key) is not "L" do Orderprocessing, do Anythingelse, and go to Loop around.
   End.

Orderprocessing is:
   For each Move (direction), for Reply (Key and Squeak), if Reply (Reply number) is in Move (Key for
     the Direction) or Reply (which will be JoyY
     if Move (Direction) is UP or DOWN, and JoyX if LEFT or RIGHT) is the same as the Move (Bias for
     the direction) then the Move (Bias for the Direction) to Position (which will be Y if Move (Direction)
     is UP or DOWN, and X if it's LEFT or RIGHT).

/* Or in better English: */
   For each Direction of Movement, for the Replies from Key and Squeak, if the Reply N is in the Keys
     for the Direction of Movement or the Reply (which will be JoyY if the Direction of Movement is UP
     or DOWN, and JoyX if it's LEFT or RIGHT), is the same as the Bias for the direction of Movement
     then add the Bias for the Direction of Movement to the Position (which will be Y if the Direction of
     Movement is UP or DOWN, and X if it's LEFT or RIGHT).
   Go back.

Anythingelse is:
   (the rest of the program goes here)
   Go back.

SKETCH CODE VOCAB.

In the series of diagrams below you'll see what the sketch code program actually looks like. When the code indicates that the computer needs to know about input devices, it actually means keyboards, mice and joysticks. This could, however, be extended to include light pens, track balls and various other ways of communicating with the computer.

Similarly, the computer knows from the program that it must generate either a flashing square, solid square, underline or query when a cursor is required.

After the terms have been defined in the sketch code, the program can start to do things with them.

Begin
These are things called:
Object, Type

Application (with three values Game, Mouse, Editor)

Ikons (with some values empty)
Potential values

RUBBISH BIN

CLIPBOARD

GAME

MOUSE

EDITOR

FILING CABINET

CLOCK

CALCULATOR

(with two values,
POSITION, X and Y)

Cursors

Things like spaceships
(with some values (empty)
Potential value

Channel (with four values
Keyboard, Joystick Y, Joystick X,
Mouse)

KEYBOARD

JOYSTICK Y

JOYSTICK X

MOUSE

Reply with four values (Keyboard, Joystick, Y, Joystick X, Mouse)

EEK!

The final expression of the program discussed on this page would follow the order below. To help you understand what might otherwise be a meaningless mass of text we have included some graphic symbols that represent the various graphic choices.

Move – up, down, left and right

GO WITH THE FLOW

This week we continue our investigation of program flowcharts, how you write them and what you do with them, and follow it up with an introduction to algorithms.

```
PROCEDURE Get up;
BEGIN
    Get out of bed;
    Put on dressing gown;
END

PROCEDURE Wash;
BEGIN
    Wash yourself;
    Shave;
    Brush teeth;
    Comb hair;
END

PROCEDURE Eat breakfast;
BEGIN
    Sit down at table;
    Eat cereal;
    Drink coffee or tea;
END

PROCEDURE Go to work;
BEGIN
    Get coat and briefcase;
    Kiss wife goodbye;
    Leave for work
END

PROGRAM Breakfast;
BEGIN
    Get up;
    Wash;
    Eat breakfast;
    Go to work
END.
```

When writing Pascal in pseudo-code, the Pascal words are typed in boldface. In this example, you are shown the modules or "procedures" that comprise the business of breakfasting and getting going in the morning.

The four procedures are Get up, Wash, Eat breakfast, Go to Work and then in the main program all those procedures are "called" in the order in which they should be executed. You can see them being called in the "listing" of breakfast, the main program.

■**Program flowcharts** — So far the types of flowchart discussed, block and system, do not indicate the internal operations of the computer program *ie* the code.

Program flowcharts represent the operations to be carried out by the computer. There are two types of program flowchart — the outline chart and the detailed chart. It's up to programmers whether they wish to use one type or both, but for very large programs it's a good idea to use outline charts, as they will give you a good idea of the subtasks that your program will have to perform.

The outline flowchart is the first stage of turning a systems flowchart into the necessary detail to enable the programmer to write the program. They present the computer operations that are to be performed, but only in general. The standard programming symbols are used, and when the detailed flowchart is prepared, each symbol within the outline flowchart would normally be expanded.

The detailed flowchart represents the last stage of planning before the actual business of writing the program. The detailed charts will contain the necessary steps that the program must take. Note that the detailed flowchart doesn't contain any code in a particular programming language, but statements in English such as set count to 1, Read master file, and so on. The reason for this is to be independent of any particular language, although it is possible to write a detailed flowchart with actual programming statements.

When preparing a flowchart, remember that you should show the necessary detail of the problem to enable you to write code from the final chart. Be liberal with paper and do not clutter it with lines. Look at diagram 2 (page 245), and you can see that connectors have been used instead of flow lines. This makes the flowchart easier to read. Also note that the flow lines have an arrow pointing to the direction that the control is to follow. When writing the flowchart, keep it tidy. Two rules can be followed here.

First, write the chart from the top left of the page, and go down and to the right. Secondly, when branching, draw your flow lines to the right when going down and draw them to the left when going up, diagram 2 shows this with the exceptions of E and I.

When you have completed the flowchart, follow it through. This will show any mistakes,

which can then be corrected *before* any coding is done. Looking at diagram 2, you can see that the loop G could end up with the caller waiting for a long, long time before the phone is answered.

The flowchart also enables you to optimise your code. Diagram two shows us that the process of replacing the receiver has been done twice, at E and K. This is quite inefficient and should be avoided at all costs.

Finally, remember that flowcharts are used to aid and not hinder you. At first, it may seem complicated, but after using them for a while, they can be of great benefit.

Algorithms

An algorithm is quite simply a set of instructions which are used to solve a problem. They have been used by many people for many different reasons for a very long time. The early Greek mathematicians used algorithms to solve problems, and so do you. In fact, any set of instructions can be called an algorithm.

Here is an algorithm to open a door:

1) Put hand on handle.
2) Grasp handle.
3) Turn handle to left.
4) Take one step backwards.
5) Pull handle towards yourself.

Looking at the algorithm, it is plain to see that it doesn't cover all possible situations. We are assuming that the handle can be turned to the left, and also that the handle is one that can be turned. What the algorithm does do is show the steps involved in opening the door. Notice that instructions 4 and 5 are the ones that actually open the door — saying 'open the door' in instruction 5 would have been ambiguous.

Algorithms can therefore be used to solve problems as far as programming is concerned. Designing good and efficient algorithms can take a long time, but once mastered, will prove invaluable.

We are not concerned with the designing of algorithms in this article, but with the way they are written down for use by the programmer.

The product of writing down an algorithm for use by the programmer is called pseudo or sketch-code. They can be written down in English, Pascal-like code, Basic-like code, or your own method of writing down instructions.

But the problem with the latter is that if everybody writes down an algorithm the way they want to, then it could be very difficult for anyone else to understand it. Therefore two methods have been used frequently.

The first method is to write down the desired processes as a series of numbered steps. These can be written in plain English, and allow the reader to grasp what the algorithm is supposed to do and how it does it. You can write down the steps in any level you desire, but it is best to start from the top and then work down to a level where you can then start to program. The door-opening example is fine for small programs, but when larger problems are being tackled, it's best to define the algorithm as a number of smaller steps. The problem 'Breakfast' shows this quite well.

1) Get up.
2) Wash.
3) Eat breakfast.
4) Go to work.
1.1) Get out of bed.
1.2) Put on dressing gown.
1.3) Go to bathroom.
2.1) Wash.
2.2) Shave (if necessary).
2.3) Brush teeth.
2.4) Comb hair.
2.5) Go to kitchen.
3.1) Sit down.
3.2) Eat.
3.3) Drink coffee or tea.
4.1) Get coat and briefcase.
4.2) Leave for work.

This method is fairly easy to get used to. It is also fairly easy to abuse. The temptation to start from the lower levels and then work up is great. Avoid it. If you do this then you can end up with a mess when it comes to writing the final code. Working top-down is more structured and modular.

In the box below, you'll see four ways of solving the same problem.

The first method (numbered 10 to 60) is a simple basic program to work out the solution to finding the cosine of numbers between one and ten. The first line is merely a Remark statement, the second a statement to print on-screen the fact that the program will print a number and then the square root of it, the third begins a FOR ... NEXT loop to generate the numbers.

The second method is just an English-language six-line descripton.

The third method is a high-level language program: the one shown in the box below written in Pascal.


```
10  REM Simple program
20  PRINT "N", "Cosine"
30  FOR C=1 TO 10
40  PRINT C,COS(C)
50  NEXT C
60  END
```

1) Print message.
2) Set count to 1.
3) Print number and cosine.
4) Add one to count.
5) If count less than eleven then step three.
6) Stop.

```
PROGRAM Cosine;
VAR c;REAL
BEGIN.
WRITE ("Cosine");
    c:=1;
    WHILE c<11 DO
    WRITE (c,cos(c));
    c:=c+1;
END
```

Arrays can be easily imagined as a series of rows and columns whose intersections make cells which can hold information, rather like letter-racks.
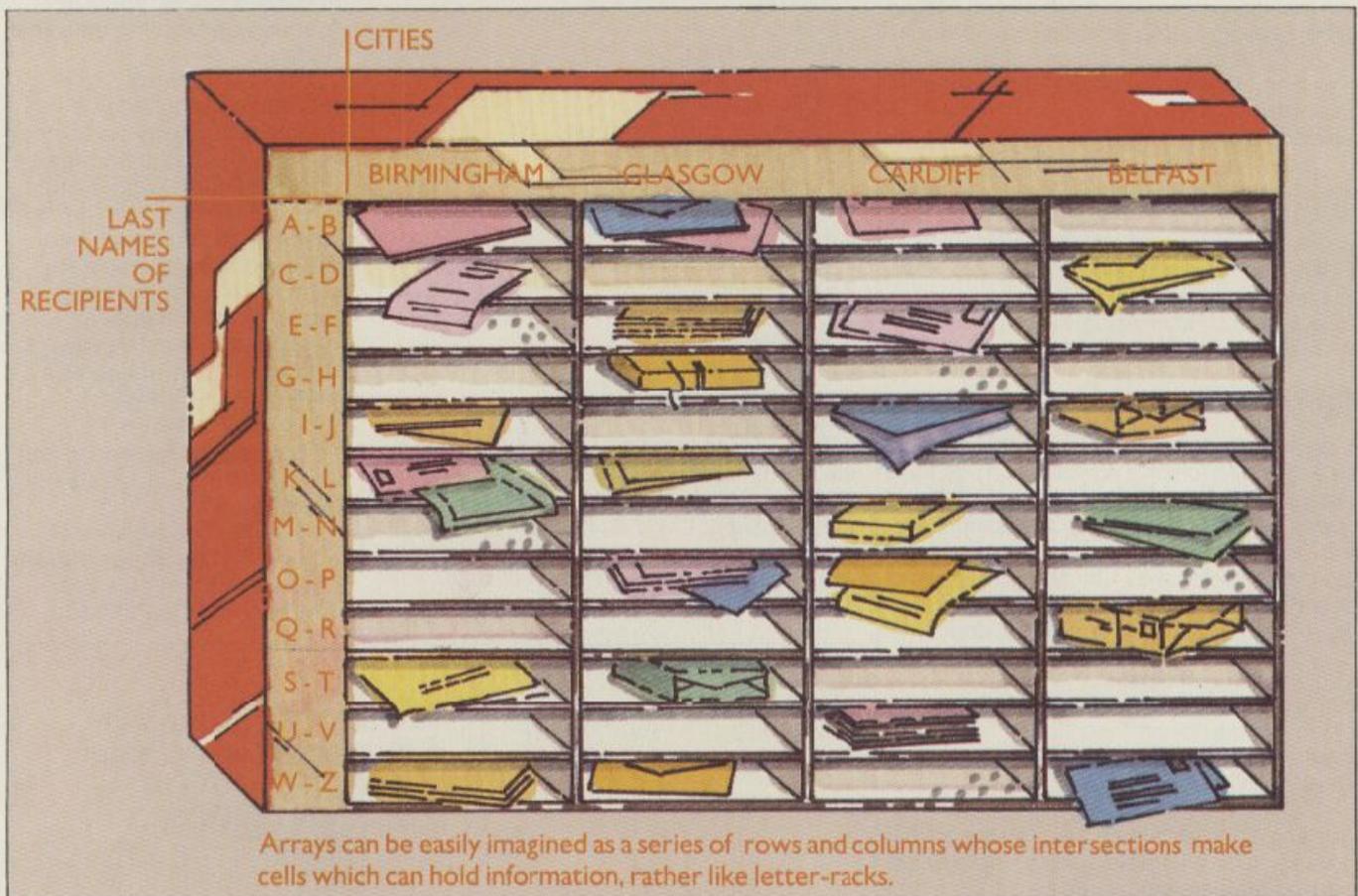
ARRAY FOR PROGRAMMING

Basic supports two types of arrays — numeric and string. Numeric arrays may *only* hold numbers, while string arrays contain characters and can therefore be used (by devious means) to hold numbers as well.

To set up an array you use the reserved word DIM. You must also give this word values known, incorrectly, as parameters. These numbers refer to the size of the array you want to use. Such numbers are also called subscripts and allow you to access and manipulate the contents of an array.

It is easiest to imagine arrays in the form of rows and columns whose intersections make cells which can hold information, rather like letter-racks. Another useful analogy is that of graph-paper, where a single point is referenced by both an X coordinate (column) and a Y coordinate (row). Arrays can have one or more rows and one or more columns. A single row array with four columns can be represented as:

```
        :Col1:Col2:Col3:Col4
Row1:    :    :    :
```

To create such an array in RAM you must give the array a name, RC$, for example. It should have a size:
10 DIM RC$(4)

Note that when you give DIM only one number it is assumed to be the number of columns and only one row is allowed. To create numeric arrays, you should leave out the $ symbol.

This particular DIM statement could be used only once in a program, although you can set up as many arrays as there is room for in RAM. This means that you cannot alter the number of rows or columns once the array has been DIMensioned. String arrays really do gobble up RAM, so keep your arrays as small as possible and CLEAR as much variable storage space as may be needed at the start of a program to avoid 'OUT OF STRING SPACE' or similar error messages.

To set up arrays with more than one row, you need to use two subscripts, row and column, in that order. To establish an array called RC$, having three by five columns, the statement would be:
10 DIM RC$ (3,5)

This sets up an array which can be shown diagrammatically as in figure 1.

This sort of array is often referred to as a two-dimensional string array, as it must be accessed by two values — row and column.

Each cell of a string array can be thought of as a normal string variable, and can thus usually contain up to 255 characters. But you should check this in your Basic manual and by experimenting, as you may find non-referenced limitations in this area.

To put information into the array you use statements like:
20 LET RC$(3,5) = "PERSONAL COMPUTER NEWS"

This puts the character string into that cell of RC$ which is referenced by the subscripts 3,5 — ie row 3, column 5. The following example should make this clear:
10 DIM RC$(3,5)
20 RC$(1,1) = "THIS IS"
30 RC$(2,2) = "HOW"
40 RC$(3,3) = "TO DO IT"
50 RC$(2,4) = "EASILY"

This gives the sort of format shown in figure 2. To get the information out, you can simply use PRINT, as in:
50 PRINT RC$(1,1)

This will display the contents of the cell

specified (*ie* "THIS IS"). Or you can assign the contents of a cell to a string variable as in:

60 LET A$ = RC$(2,4)

A$ will now 'contain' the string "EASILY".

To use numbers in string arrays, first convert the number to its string representation:

100 X = 99
110 X$ = STR$(X)
120 RC$(3,5) = X$

Don't forget that a string thus defined will have a leading space if it is positive and a negative sign if it's not (the length of the string made by A$ = STR$(9) is 2 — not 1 as you might expect).

To get numbers out, reverse the process using VAL:

200 X = VAL(RC$(3,2))

At this point it should be obvious that to access all the columns of a given row we can use a FOR . . .NEXT loop either to put information in or get it out. Thus, if we wanted to see the contents of the cells of row two, we would write:

400 FOR C = 1 TO 5 *for all columns*
410 PRINT RC$(2,C) *print contents of row 2, column C*
420 NEXT *next column*

We don't want to have to repeat this for every row, so we embed this loop in another:

390 FOR R = 1 TO 3 *outer row loop*
.................................*inner column loop*
430 NEXT R

A string array can be used to set up a simple database which can be sorted and searched. For example, you could set up an array to hold record titles (column one), artistes (column two) and classification, *eg* pop or classical (column three) to act as an index to a record collection.

The simplest way to get the data into the array is by using READ and DATA statements, although you could have the program ask the user for the information. The DATA statements would take the general form: *line no. DATA title, artiste, classification.*

For example:

5000 DATA SOCIAL STUDIES,CARLA BLEY,JAZZ
5010 DATA DISCIPLINE, KING CRIMSON, JAZZ
5020 DATA VAUGHAN WILLIAMS CONCERT,ACADEMY OF ST. MARTIN IN THE FIELDS, CLASSICAL

Notice that you cannot put commas into the information within the DATA statements because they are reserved to separate each item.

If we have 50 albums, the program could begin like this:

10 NR = 50:REM NUMBER OF RECORDS
20 NC = 3:REM NUMBER OF CELLS PER ROW
30 DIM RC$(NR,NC):REM SET UP DIMENSION IN RAM
32 REM
35 REM✶✶✶✶✶✶FILL THE ARRAY✶✶✶✶✶✶

40 FOR R = 1 TO NR:REM FOR EACH RECORD
50 FOR C = 1 TO NC:REM FOR EACH CELL
60 READ RC$(R,C):REM READ THE RELEVANT DATA ITEM INTO THE CELL
70 NEXT C:REM NEXT CELL
80 NEXT R:REM NEXT RECORD
90 REM&&&&&ARRAY FILLED&&&&&

Note that line 60 is a compact form of the two expressions:

READ A$:LET RC$(R,C) = A$

All that remains is to design a program to access the information in the array. To do this we must first decide exactly what users may be most likely to require. They will certainly want to be able to search the database for a title by a given artiste so we will concentrate on this first.

Next week we take a look at dimensioning arrays and continue to build up the simple database program.

Diagram 1
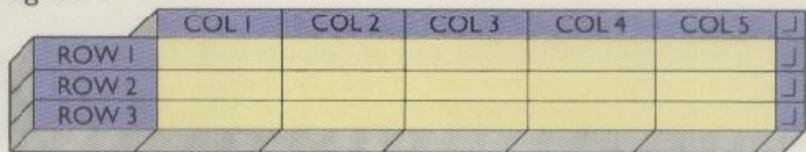
Diagram 2
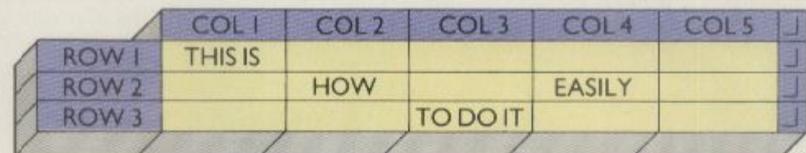
Diagram 3
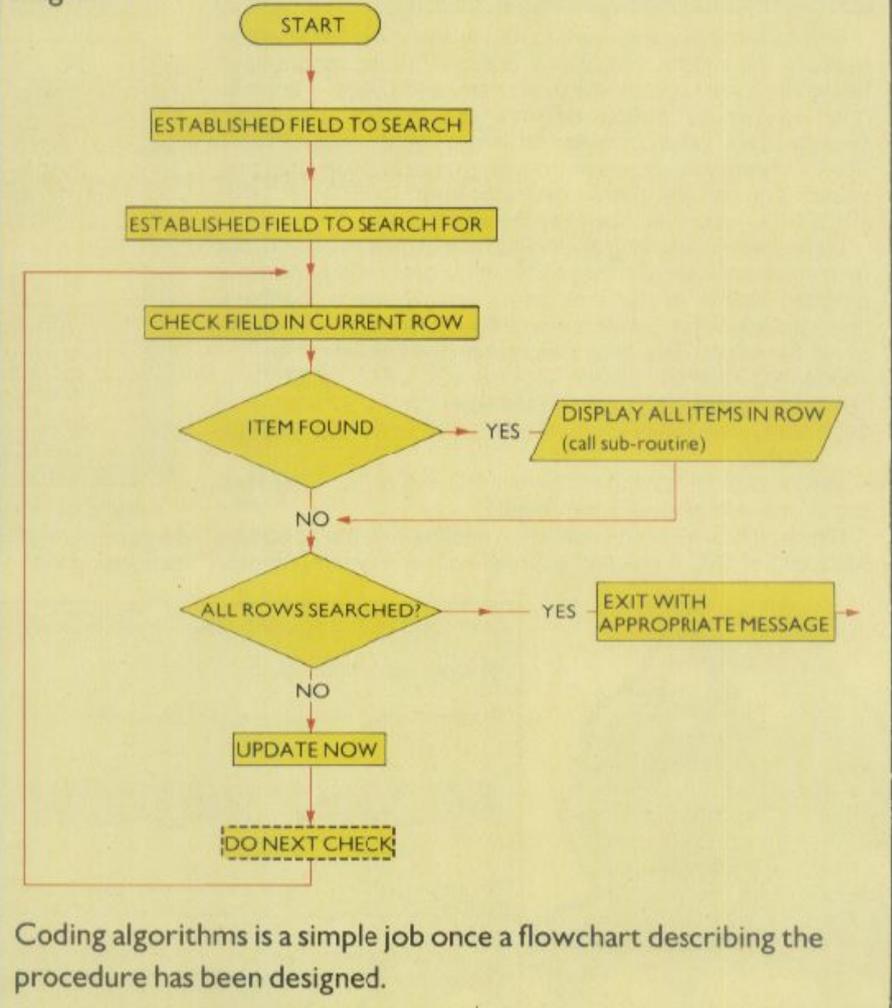
Coding algorithms is a simple job once a flowchart describing the procedure has been designed.

FINDING FAULT

Interpreter-reported error messages are useful in debugging programs, but they are by no means the whole story. Error messages can be misleading: they can point you to the line where the program stopped, rather than the one where the error occurred, and they can also be distinctly enigmatic.

So no matter how good your micro's error messages are, it's still useful to know a little about why errors occur in the first place.

Logical errors

These occur when your algorithm for solving a problem or performing some operation is flawed. First you should check exactly what you wanted to achieve, then go back to your notes — you did design the solution before coding it, didn't you? You might find it useful to redesign the routine on paper and recode it, then compare this with the section that doesn't do what it's supposed to do.
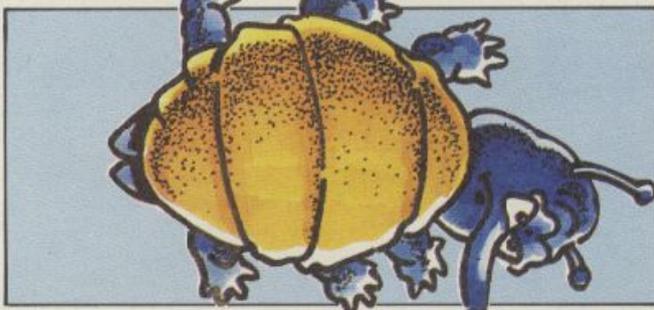
One easy mistake to make is to re-use a variable name, either because the routine that uses it is lost in a long program, or because you've forgotten the exact names you used — another good reason for having long variable names, but again something not usually provided for. It helps here to insert PRINT "X=";X statements are relevant points in order to follow changing values. You can also use an extended form: 100 PRINT "LINE 100,X=";X — but renumbering will mess this up.

Logical errors are naturally the hardest to give advice on, but here's one example often seen in various forms. It occurred in a program where, at the end, users were given a feedback message about their performance. If the score (SC) is taken to be out of 10, can you see what's wrong with the following lines?

```
1000 IF SC<5 THEN PRINT "THAT'S NOT VERY GOOD"
1010 IF SC >5 AND SC<9 THEN PRINT "THAT'S FAIR"
1020 IF SC = 10 THEN PRINT "THAT'S FULL MARKS—WELL
     DONE"
```

Obvious isn't it? (What happens if SC = 5 or SC = 9? How should these lines have been coded?)

Similar difficulties can arise when dealing with the operators AND and/or OR. If you find yourself having to write complex statements using these together, be careful. Use brackets to group statements and draw up a truth table if necessary. In Boolean logic most IF statements reduce to a basic form:
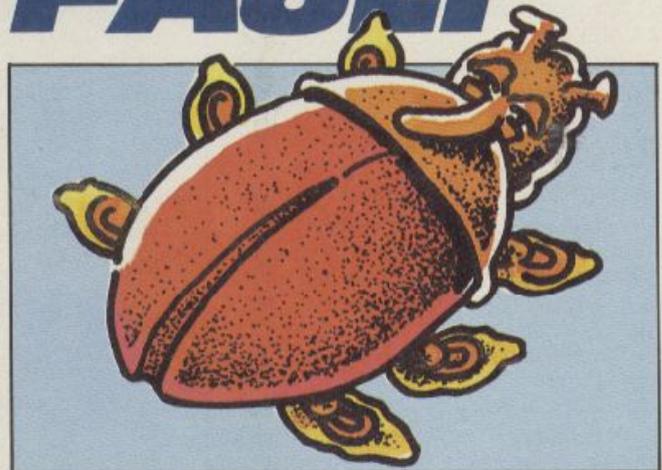
IF *(condition)* THEN *(operation)*

eg IF (X = 3) THEN GOTO 5000

The *operation* will only be carried out if the *condition* is TRUE; if the condition is evaluated as FALSE the operation will be ignored.

The condition may be complex, as in IF (X = 3 AND Y = 7) THEN . . . The two statements will be evaluated as, is X = 3, is Y = 7? If both these are TRUE then the operation will be executed. In a truth table the possibilities are shown clearly:

X = 3?	AND Y = 7?	OPERATION?
TRUE	TRUE	YES
TRUE	FALSE	NO
FALSE	TRUE	NO
FALSE	FALSE	NO

If we alter the AND to OR, see the difference it makes:

X = 3?	OR Y = 7?	OPERATION?
TRUE	TRUE	YES
TRUE	FALSE	YES
FALSE	TRUE	YES
FALSE	FALSE	NO

Setting out complex statements in detail like this should really be done during the development phase — not used as an *ad hoc* debugging tool.

NEXT WEEK

We give you the final word on Basic debugging, another word on numeric and string arrays, a lesson on getting out of a fix using flowcharts, and yet more on becoming the elegant program designer you've always wanted to be.

In the coming weeks you'll also get an introduction to the Forth and the high-level Pascal languages as we give you the answers to your questions about these increasingly popular program tools.