PCN micropaedia

Vol 11 Part 8

PASCAL AND PERIPHERALS

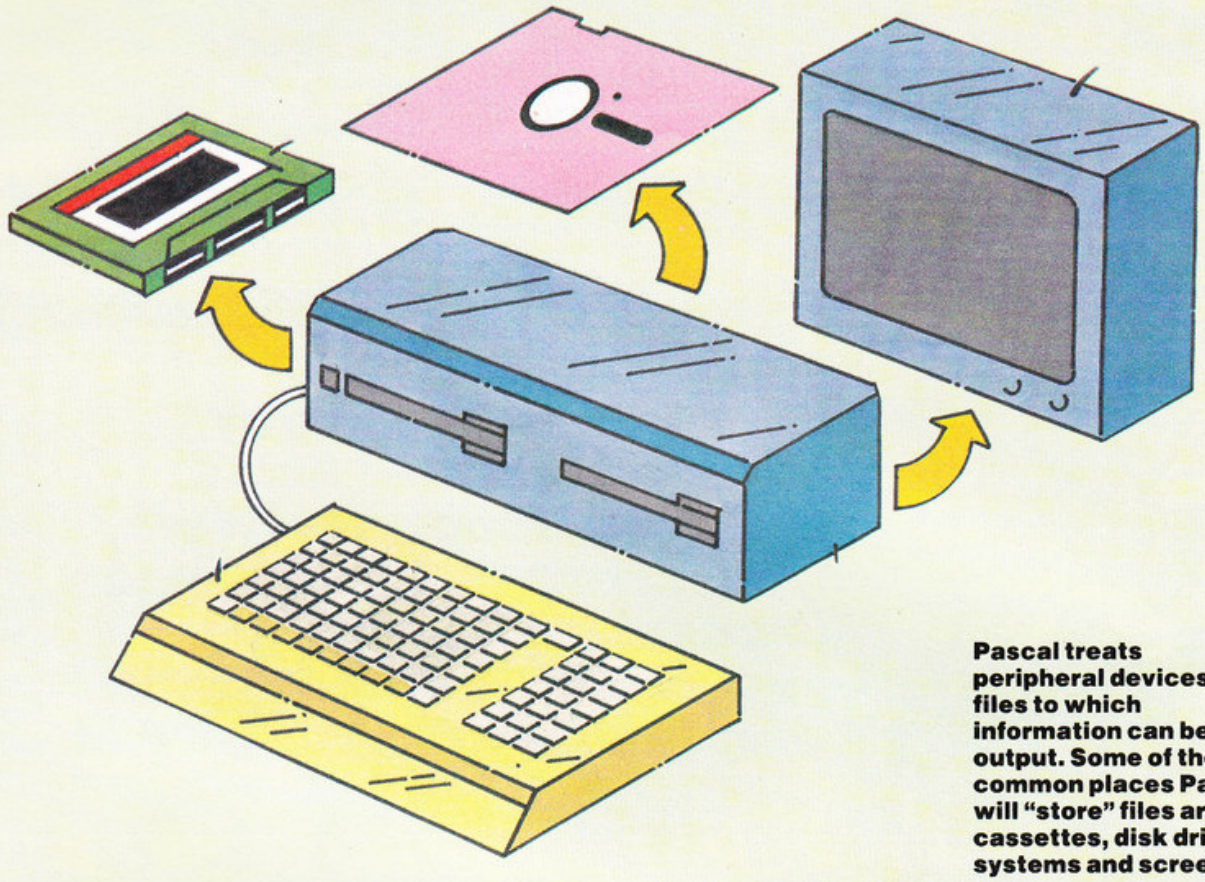
In the final part of Everything You Always Wanted to Know About Programming...But Were Afraid to Ask, we power up Pascal, arrive at an Adventure, deal with data and discuss civility in CP/M.

DIGGING INTO DATA STRUCTURE

WHEN IN CP/M...

ADVENTURES FOR ALL

PULL OUT AND KEEP

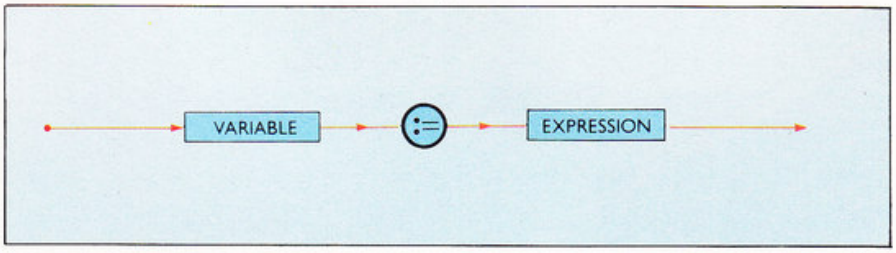


Pascal treats peripheral devices as files to which information can be output. Some of the common places Pascal will "store" files are cassettes, disk drive systems and screens.

FOUNDATIONS OF LANGUAGE

We introduced you to the Pascal programming language last week. This week we take you one step further with a look at Pascal statements, simple Input/Output and control structures as well as procedures and functions.

There are a number of types of statement in Pascal and so far we have come across the output statement WRITELN. Pascal also has assignment statements, for example A := B. Notice that ':=' is used to assign values, '=' is used in comparisons as in "if a equals b". The syntax for the assignment statement is as follows:



If there is to be more than one statement within a particular block, then they are referred to as compound statements. These must be bracketed by the reserve words BEGIN and END:

```
BEGIN
  A := 1;
  Y := A+A;
  WRITELN(y)
END;
```

Note that the semicolon is not required after the last statement within a block, although no harm is done if it is included. Also, after a block, the END should be followed by a semicolon, only the final END of the program should be followed by a full stop.

Simple I/O

As far as Pascal is concerned, all Input and Output is carried out between files. The file could be magnetic tape, disk or the visual display unit. At the beginning of a Pascal program, input and output is optionally specified within the program line:

```
PROGRAM DEMO (INPUT,OUTPUT);
```

So far, the input and output discussed has referred to the keyboard and terminal, but it is possible to assign files on different media to a Pascal program. Different versions of Pascal have different means of doing this, and the job of assigning the file to a peripheral is done at the operating system level. For the sake of brevity I/O with the keyboard and VDU will be dealt with here.

The two most common I/O statements are READLN and WRITELN. If LN is included in either, then input or output will not be expected on the same line. So one item per line will be read or written to the VDU or keyboard. READ or WRITE may be used and this means that input or output will appear on the same line.

With WRITE or WRITELN, it is possible to format the output so as to achieve neatly aligned reports. The field width can be specified and this is shown below, the dashes representing spaces.

```
WRITELN(23:6)
'----23'
```

Writing floating point numbers can be a problem, as the output is usually in scientific notation:

```
WRITELN(1.5)
'1.500000000000E+00'
```

Specifying a field width will only justify the number, but it is also possible to specify a second field width parameter which specifies how many decimal places will be printed:

```
WRITELN(1.5:7:1)
'----1.5'
```

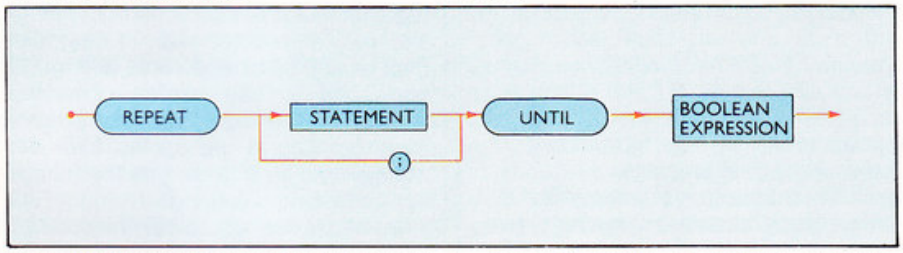
Note that the decimal point counts as one character space. Specifying field width parameters will work for all the standard scalar types.

Control structures

The real power of Pascal is in its control structures. They are quite simple yet they can be used in complex situations, so each will be discussed only briefly.

The need for repetitive statements is quite clear, if we had only sequential control structures we would have to do a lot of programming whenever we wished to repeat something. Pascal provides three types of automatic looping; REPEAT, WHILE and FOR. There is more than one because each has its own particular convenient features.

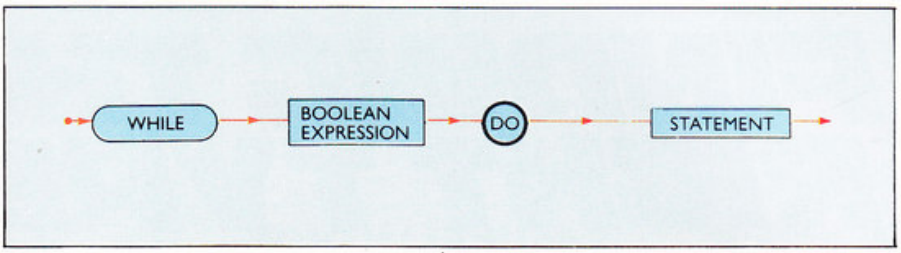
The REPEAT statement usually appears in the form 'REPEAT statements UNTIL condition is true'. The syntax for the statement is as follows:



The boolean expression after UNTIL will terminate the WHILE. The operators that may be used are the relational ones and any of the legal boolean operators. The statements within the REPEAT . . . UNTIL will be executed at least once, and the check or checks are made at the *end* of the loop. Here's a program that uses REPEAT . . . UNTIL:

```
PROGRAM COUNT (INPUT,OUTPUT);
(* COUNTS FROM 1 TO 25 *)
VAR
  I : INTEGER;
BEGIN
I := 1;
  REPEAT
    I :=I+1
  UNTIL I = 25;
END.
```

The second loop statement that Pascal has to offer is the WHILE statement. The



syntax for the statement is shown here.

Here we can see that the Boolean expression is situated at the start of the block. Note that any statement or statements that appear after the DO must be bracketed by BEGIN and END. There may be several levels of BEGIN and END within the WHILE . . . DO and the final termination will be achieved when the condition is met and the last END (followed by a semicolon) is 'matched'.

This loop differs in operation from the REPEAT . . . UNTIL insofar as the loop need not be executed at all. It should be pointed out that it is all too easy to mix up boolean conditions in Pascal REPEAT . . . UNTIL and WHILE . . . DO statements. In that case the loop(s) may never end, so great care should be taken. Below is another Pascal program which makes use of the WHILE . . .DO loop:

```
PROGRAM DEMO (INPUT,OUTPUT);
(* USES THE WHILE. . . DO LOOP *)
BEGIN
  I := 1;
WHILE I < 26 DO
  BEGIN
    WRITELN(I);
    I := I+1
  END;
END.
```

The third type of control structure available in Pascal is available in nearly all high-level programming languages, *ie* the FOR . . . . NEXT loop. This is unconditional as far as Boolean expressions are concerned, rather it uses counter variables (or constants) to specify a start and end for the number of times the loop is to be executed. The FOR . . . NEXT loop can shorten a Pascal program because the updating of the counter variable is automatic. It should be noted that operations on the counter variable within the FOR . . NEXT block are illegal. The syntax for the FOR . . NEXT is below:

```
PROGRAM DEMO (INPUT,OUTPUT);
VAR
  I := INTEGER;
BEGIN
  FOR=I1:= 1 TO 25 DO
    BEGIN
      WRITELN(I)
    END;
END.
```

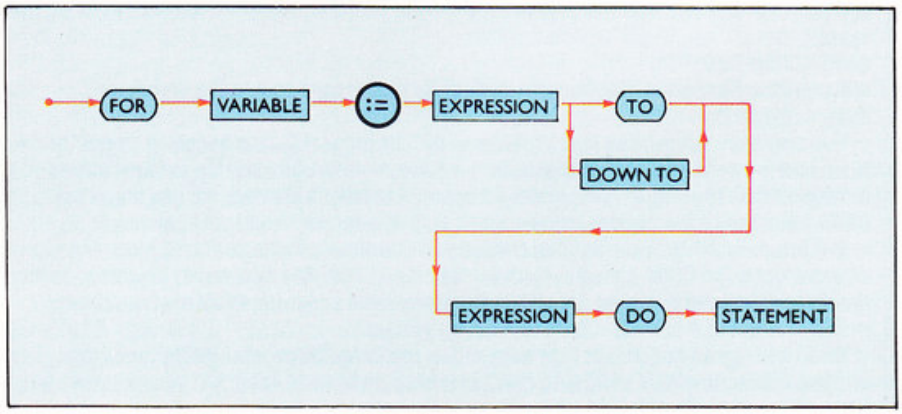
Procedures and functions

Finally, Pascal allows the programmer to define procedures and functions. These are the thing to use when the programmer wishes to name a group of statements and call them from the main part of the program. The difference between the two is that a function returns a value that may be used within an expression, while a procedure has no value associated with its name (although values may be passed between different procedures).

In addition to Forth, Pascal and Basic, there are a number of other high-level languages which can quite easily be run on popular micros.

Lisp is known as an arificial intelligence language which 'learns' about a subject as the user interacts with it. Its only drawback in being used with micros is the amount of memory it eats up.

Logo is a teaching language, and is often associated with the 'Turtle graphics' system popularised as a teaching aid in schools.



FINDING THE RIGHT TYPE

Computers are used for an enormous variety of tasks which involve the manipulation of many different kinds of data. To take just a few examples, the data may be: numbers in scientific and statistical programs; a mixture of text and numbers in commercial data processing; a combination of commands and data in interactive computing; joystick movements and the state of the screen display in an arcade game and formatting and other instructions together with text in a word processor.

Because data is so important, you have to define exactly what data a program has to handle before you start writing the program. In commercial data processing there is a division of labour between systems analysts — who prepare specifications giving precise details of the data a programe needs and what the program does with the data, and programmers — who work from the specification and decide how the program is going to do its job and write the program instructions that tell the computer what to do.

There is a big gap between the real world where data has a meaning, and the guts of a computer where everything is reduced to patterns of electrical impulses. The gap is bridged by the features provided in programming languages for representing, organising and manipulating data of various types.

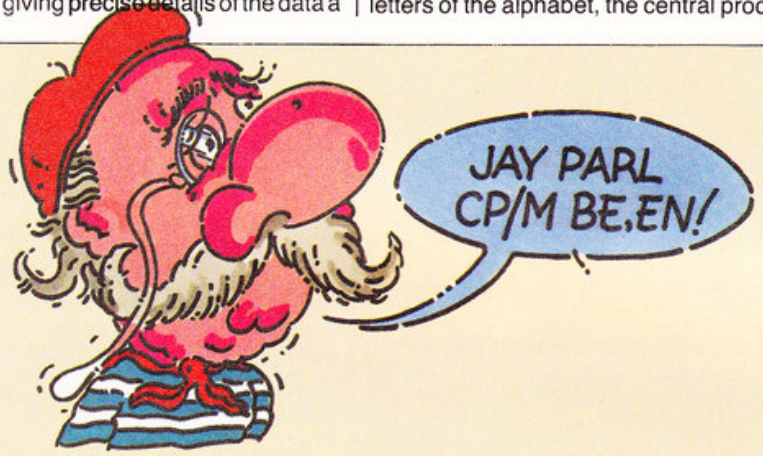
At the machine code level everything in the computer can be regarded as binary numbers. If a programmer writes machine code instructions for multiplying two number that were intended to be codes for letters of the alphabet, the central processor will quite happily carry out the instructions and produce a meaningless result, while in a high level language an instruction like x;="a"*"b" will be rejected. Most high level languages allow you to use certain fixed data types, enforcing rules on what counts as a particular data type and providing operations that are meaningful for each data type.

Most versions of Basic have two data types, floating point numbers and strings. Floating point numbers are written in the usual form, 100, 11.5, −3, etc, or in scientific notation, 5E10, −12.6E2 etc, while strings are written as sequences of characters enclosed in quotes, "HELLO", "123", "MICKEY MOUSE". You have to use different kinds of variable names for numeric variablesand string variables — the precise rules for variable names differ between Basics but the name of a string variable must end with a dollar sign — and you cannot give a numeric value to a string variable or vice versa. Thus, LET A = "123" and LET A$ = 123 are not allowed, the string "123" being regarded as different from the number 123.

The operations you are allowed to carry out are designed to be meaningful for the data types. You can perform any normal arithmetic calculation with numbers and numeric variables, or you can join strings together or extract sections from strings. You may sometimes want to treat numbers as strings or vice versa and for this you have to use explicit conversion functions — STR$ gives you a string corresponding to the printed form of a number and VAL gives you the number corresponding to a string.

Some Basics also allow integers and integer variables which can take only whole number values and not fractions. The notation for integer variables varies — the most common is that the name of an integer variable ends with a percentage sign. Integer arithmetic is similar to floating point arithmetic, but there is an important difference in division. Floating point division gives a fractional result (for example 3/2 is 1.5) but integer division always gives an integer result. So the integer division 3/2 gives the answer 1.

Other programming languages also provide a fixed set of data types. Fortran was designed for scientific programming and allows you to use integer, real (floating point), complex, and double precision numbers, but is very poor at string handling. The rules for variable names in Fortran do not provide special characters like $ and % at the end of a name to distinguish variable types. There is a



Just as a computer language resembles a spoken human language, an operating system is like the culture of the society that speaks a given language.

You may be letter-perfect in your pronunciation and handling of grammar, but if you don't know the context of the words and sentences you are writing or speaking, you are not likely to communicate effectively.

One of the most popular business operating systems is CP/M – implemented on lots of 8-bit micros.

One of the ideas behind CP/M is to give you English-language commands to carry out various simple tasks. For example, the command:

TYPE PCN

will print on-screen a text file with the name PCN. Similarly the COPY command, using the syntax:

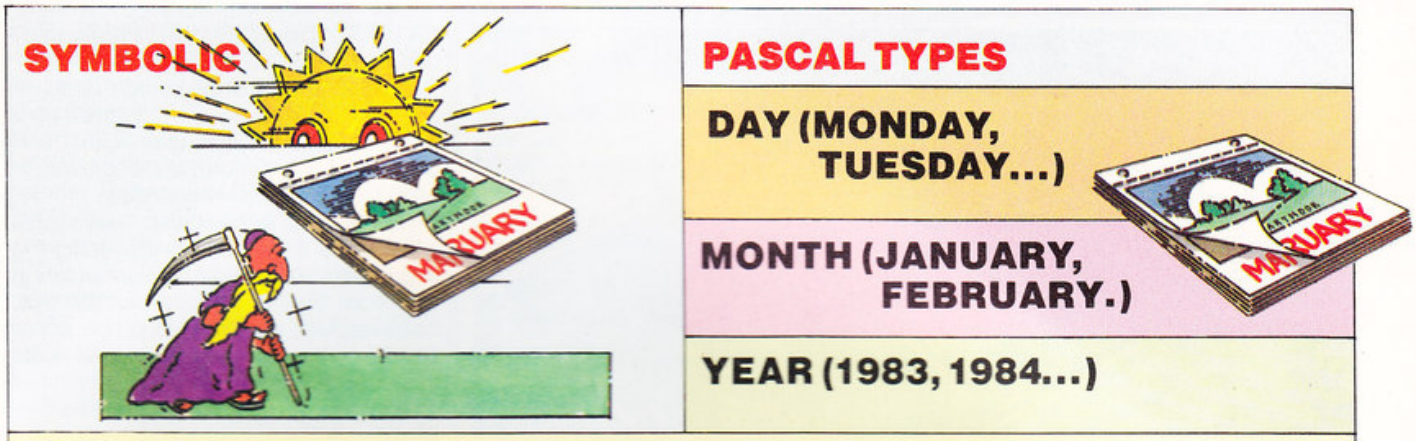
COPY A: PCN TO B: PCN

will copy the file called PCN from the disk-drive designated as A to the disk drive designated as B.

You may have guessed by now that one of CP/M's biggest jobs is handling computer files; making sure that each file goes in the correct place and carrying out operations between files. Therefore, it is almost a necessity to have a disk drive if you are to use CP/M – and hence the system has remained largely a tool of the micro business user.

But prices of CP/M systems have come down drastically in the past two years and you can now get a full CP/M-based system for less than £700. The new price, combined with the large volume of business programs already written under the CP/M system, should revive interest in this 'old reliable' operating system.

CP/M will run on computers that have either the Zilog Z80 or Intel 8080 processors. It needs a minimum of 20K of RAM to run, but will operate more happily if your machine has 64K.



The diagram above shows two methods of displaying data types: the first is "english-language" symbolic representations, while the second is Pascal data typing – where certain "Types" are described by words within those types (i.e. TYPE Day = (Monday, Tuesday...).

default rule that names beginning with the letters I to N are integer variables and other names are real variables, but this rule can be overridden by a declaration at the beginning of the program.

Fortran is much stricter than Basic about converting and mixing real and integer numbers. Although Basic may allow you to do such things as I%= 1.0, X=3, I%=X, X=I%, Fortran requires that integers are written without a decimal point, floating point numbers are written with a decimal point, and type conversion must be carried out explicitly with the functions FLOAT to convert from integer to floating point and IFIX to convert from floating point to integer. In legal equivalents of the above, Basic statements would be I=1, X=3.0, I=IFIX(X) and X=FLOAT(I).

Pascal is very rich in data types. As well as the built in data types; Boolean (logical), Integer, Real, and Char (character), it allows you to define your own data types and to define subtypes of any data type.

There is no distinction in Pascal between the forms of variable name allowed for different types, but all variables together with their type must be declared in the program before they are used. Pascal checks that any value given to a variable in the program is of the right type and gives an error message if there is a mismatch.

To define a new type you have to declare the name of the type and the values allowed in it, for example:

```
TYPE  DAY = (SUNDAY, MONDAY, ...,
          SATURDAY)
      MONTH = (JANUARY, FEBRU-
          ARY, ..., DECEMBER)
```

We could then define variables TODAY and THIS-MONTH by

```
VAR  TODAY : DAY
     THISMONTH : MONTH
```

and statements like TODAY := MONDAY or THISMONTH := SEPTEMBER would be allowed but statements like TODAY := 2 or THISMONTH := CHRISTMAS would be illegal. The advantage of this kind of type checking is it makes program errors and data input errors much easier to find.

Only the very simplest programs will just use isolated items of data. Usually the data is organised in some way, with a number of items belonging together being treated as a group, and perhaps with groups of items also being organised. As an example, the data for a payroll program would have the information for each employee, name, salary, tax code, etc, grouped into an employee record, and the employee records perhaps grouped into department files, and so on. Most high level programming languages provide precisely defined data structures, varying from language to language, and chosen for their suitability to the special applications the language is designed for.

The only data structure provided in Basic is the array. An array is a collection of items, all of the same type, which can be indexed, or referenced, by a number or set of numbers. For example, if you wanted a program to deal with monthly sales figures you could use an array called SALES that holds twelve numbers, and store the January sales total in SALES (1), February sales total in SALES (2), etc. This kind of array is called a vector or one-dimensional array because only one number is needed to access any item in the array. Most Basics allow two, three, or more dimensions in an array. If we extend the above example and suppose we have the sales figures from several departments for each month, we can store the figures in a two dimensional array, SALES (1,1) being January sales for department 1, SALES (2,1) being January sales for department 2, SALES (1,2) being February sales for department 1, and so on.

Basic allows string arrays as well as numeric arrays, and those versions of Basic that have integer variables may also allow integer arrays. The name of a string array must have a dollar sign at the end, and the data stored in the array must be strings. For example, we could have an array called M$ holding the names of the months, so M$(1) is "JANUARY", etc.

Before you can use an array you have to specify its size with a DIM statement, so in the sales example we would have to have for the first case DIM SALES (12) and for the second case, if there were 10 departments, DIM SALES (10,12). If you omit the DIM statement most Basics will give you an array of size 10 in each dimension.

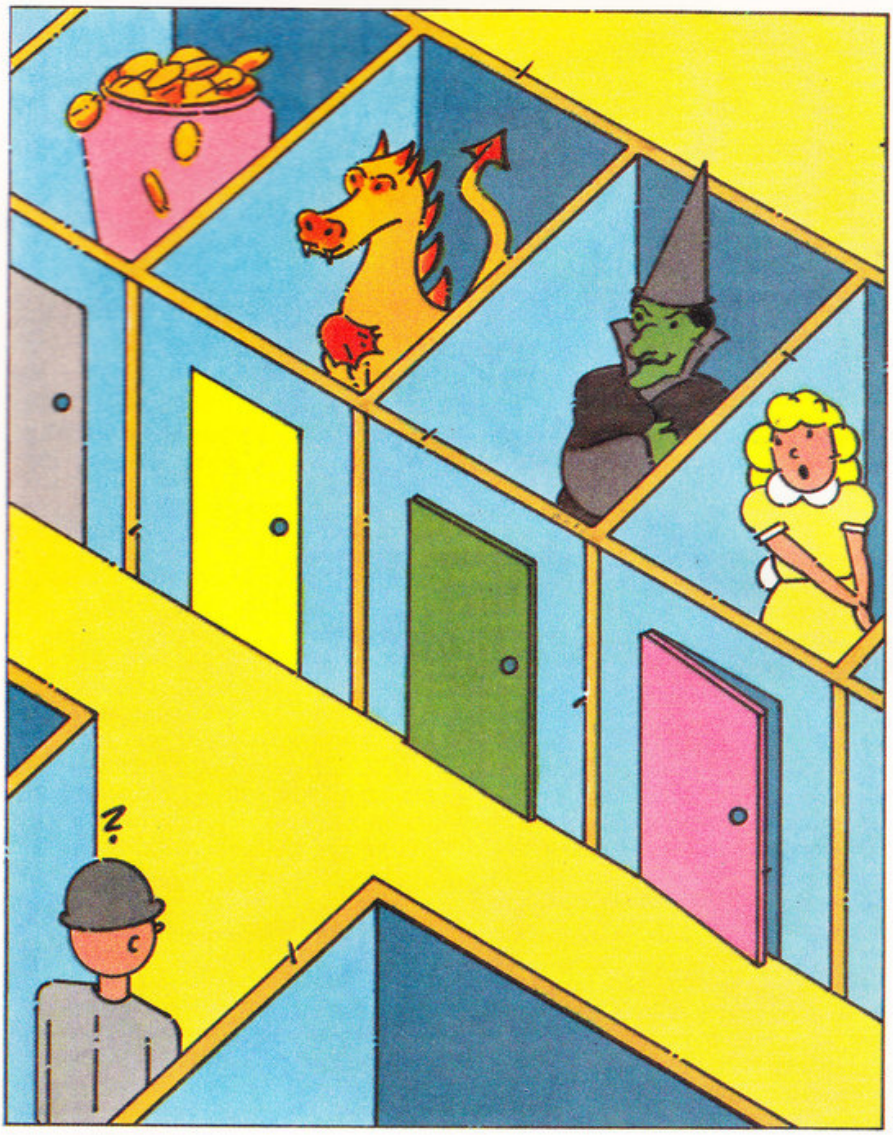
Arrays are provided in most high level languages, such as Algol, Fortran, Pascal, etc, but there are languages that do not have arrays provided, *eg* Forth and Lisp.

In FORTH everything is done with a data structure called a stack. Data items can be put on to a stack and taken off again, and the defining feature of a stack is that only the last item added to the stack can be retrieved, so the data comes back in reverse order. A stack is sometimes called by the longer name of Last In First Out list or LIFO which is more descriptive. The name stack comes from an analogy with a stack of plates or trays in a cafeteria, where the easiest thing to do is add items to the top of the pile or take them off the top again.

Some languages allow a data structure called a queue which is similar to a stack, but where items are put at one end and taken off from the other.

In Lisp the fudamental data structure is the list (Lisp is actually short for List Processing). A list is made up of a succession of items one after another, and the items in a list may be lists or atoms. An atom is the fundamental data type in Lisp, divided into two sub-types: numeric atoms which are numbers and literal atoms which are names or strings). The difference between a one dimensional array and a list is that with a one dimensional array you can pick out an item just by giving its position in the array, with a list you have to read through from the beginning, item by item, until you come to the one you want.

In an array all the items have to have the same data type. For many purposes this is inconvenient because we want to mix different data types in the same data structure. In the payroll example above, for instance, we want to group the employee's name (a string), salary (a number), and other items which may be of different types. A data structure like this is called a record and some languages, Pascal and COBOL in particular, provide extensive facilities for defining records.



OBJECTIVE DECISIONS

Last week, we looked at how you might want to design an adventure game and established that you'll probably want to have various rooms within the adventure that your character can visit. You can now start to think about coding a routine for placing a few items in them. In the example, you'll place each item in a room at random; the restrictions being that an item can only appear in one room and that there can't be more than three items in a given room.

```
a FOR I =1 TO OB:REM FOR ALL OBJECTS
b R=RND (NR):REM SELECT A RANDOM ROOM
c IF OI(R,O)=3 THEN GOTO b :REM ROOM FULL, CHOOSE ANOTHER
d OB(I)=0 :REM REMOVE FROM LIST—FLAG AS "IN USE"
e OI(R,O)=OI(R,O)+1: REM ADD ONE TO NUMBER OF ITEMS IN ROOM
  REM SUBSTITUTE MN+1 FOR 0 IF YOU CAN'T USE CELL0
f OI(R,OI(R,0))=I: PUT OBJECT NUMBER IN NEXT FREE CELL OF OI
g NEXT
```

This example deliberately doesn't use line numbers so you can put these routines where you want.

Line **e** needs explanation: initially this location [OI(R,0)] holds the value zero showing no objects present. Add one to this both to update the number of items in the room and to point to the next free cell — *ie* cell 1. Now put the loop counter (I) in here in line **g**.

You can simplify line **g** to the 2 statements: FR=OI(R,0) (or OI(R,MN+1) if you can't use zero) then OI(R,FR) =1.

If you want to stick certain items in special places you will have to draw up a table. You then either user READ and DATA items to perform the operation or code each placement separately.

Now go back to the screen — and you'll see there's a problem in assessing the information in OI() to tell the user what's in the room they're in. You can do this, however, with a for/next loop, the upper limit being the maximum number of items in a location. If a given cell in OI()is zero — as it might be if something is removed — then skip it. The numbers in OI() refer to the names in OB$() that were set up earlier to hold the object names.

So, first inform the user where they are and where they can go to (see above) and, bearing in mind that the current location number is IR, you might have:

```
a NI+OI(IR,0):REM NUMBER OF OBJECTS — USE MN+1 IF NECESSARY, NOT 0
b PRINT"YOU CAN SEE"
  c IF NI=0 THEN PRINT"NOTHING":GOTO)
d FOR I=1 TO MN
  e IT=OI(IR,I) : REM OBJECT NUMBER
f IF IT=0 THEN GOTO: REM SKIP IF NOTHING IN THAT CELL
g IT$=OB$(IT)
  h PRINT IT$
    i NEXT
      j RETURN
```

If you used OV() to hold object values, you could display the 'price' of an item by adding OV=OV(IT) to line **g** and amending line **h** to PRINT IT$;" (";OV;")" which would print an object's worth in brackets after its name.

To take the program a step further, you can allow the adventurer to move around within the map by asking for a command. This is the hardest area to deal with since the user can enter an infinite number of statements and you're not in a position to devise a program that understands a universal language quite yet. If you just limit yourself to PRINT"WHERE NEXT?", you can accept numbers for moves.

If you've run the program so far you will have noticed that when the screen display shows where one can go to from a place, the numbers given are 1, 2 or 3 *ie not* the 'real' numbers of the locations. These numbers refer to the columns of the array RM() so you can get the next IR from the correct place very easily.

The first thing to do is to get the user's choice. It's easier to do this with INKEY$ if you get fed up with having to press ENTER or RETURN all the time. Although you won't in fact use all the code that follows, it allows you to test out the map, placing and naming objects and so on.

```
a NP$=INKEY$: IF •NP$="" THEN GOTO a)
b NP=VAL(NP$): IF NP<1 OR NP>3 THEN GOTO a)
```

c IR=RM(IR,NP): REM UP DATE IR FROM CELL NP OF CURRENT ROW

You can now provide a clear screen and show the user where he is. But that's by no means the end of the matter. What happens if the way is barred *ie* a negative number is encountered in RM(IR,NP)? The program will crash because you can't access arrays using negative subscripts.

This requires some elaboration of the movement coding given above.

c LET TV+RM(IR,NP): REM TEST VALUE
d IF TV=? THEN PRINT "NO SUCH EXIT":GOTO a)
e IF TV<0 THEN PRINT "THE WAY IS BARRED":GOTO a)
f IR=TV: REM NOW IN A DIFFERENT LOCATION

Before moving on to do some analysis of the users' commands, take a quick look at the routines you'll need to allow the user who wants to remove items or leave them in the current location. As mentioned earlier, you'll need an array to hold the reference number of times the user has with them:— DIM HW(5) for example. Once again you'll have to use the first element of this array (HW(0)) to hold the number of times they have. It's not essential to do this but if you can't use column zero you'll have to work out a way round it.

Once you've checked that what the player wants to take away really is there and that the item is currently in column CN of OI(), you must put the object reference number into the next free cell in HW(). Then add one to the number of items our intrepid adventurer has, remove the item from OI(CN) by putting a zero in it and finally subtract 1 from OI(IR,0) to keep the number of times in the room up to date.

a IF HW(0)=5 THEN PRINT"YOU'LL HAVE TO LEAVE SOMETHING FIRST": RETURN
b REM CAN ONLY CARRY UP TO 5 ITEMS
c IT=OI(IR,CN): REM NO. OF OBJECT TO TAKE
d HW(0)=HW(0)+1: REM ADD ONE TO NUMBER OF TIMES BEING CARRIED
e HW(HW(0))=IT: REM PUT ITEM NO. IN NEXT FREE CELL OF HW()
f OI(IR,CN)=0: REM DROP ITEM FROM ROOM
g OI(IR,0)=OI(IR,0) — 1: REM DECREMENT NO. OBJECTS IN ROOM
h RETURN

You should be able to see how this would be modified to do the opposite *ie* for the user to leave something in the room.

Dealing with the user's input is one of the hardest tasks facing any would-be adventure programmer. This example sticks to the simplest case where the user enters either a number (as used above to check routines for room linkages etc) or two words; a verb and a noun *eg* "TAKE SPOON.

To begin with you must check if the INPUT is a number in the range 1-3 as above for movement between rooms.

Then you need to split the Input (if it's not a number), taking the left-most characters up to the first space as a verb. The right-hand characters will be dealt with similarly for nouns. There are two ways of doing this, one for the unfortunates who lack INSTR; the other is more elegant. Before we set out the code, however, we must get some verbs into the program.

Just as before, this is done most efficiently in an array using READ and DATA statements to fill it:

```
NV=10: REM NUMBER OF VERBS
DIM VB$(NV)
DATA TAKE, LEAVE, THROW, DROP,
  GIVE, USE . . . . .
FOR I=1 TO NV: READ VB$(I): NEXT
```

Now the routine. Unfortunately you'll have to drop the single key press for movement used earlier since we are dealing with more complex entries. This routine is called as a GOSUB, as are most of the modules presented here.

```
1000 PRINT "WHAT NEXT";: INPUT
  WD$:IT=O:VN=∅:OB$="":VB$=""
1010 IF LEN WD$>1 THEN GOTO 1070:
  REM NOT A VALID NUMBER CHECK
  FOR VERBS ETC.
1020 NP=VAL(WD$)
1030 IF NP<1 OR NP>3 THEN PRINT
  "INVALID ENTRY, TRY AGAIN":
  GOTO 1000
1040 TV=RM(IR,NP): IF TV=0 THEN
  PRINT"NO EXIT THERE":GOTO 1000
1050 IF TV<0 THEN PRINT " THE WAY
  IS BARRED": GOTO 1000
1060        IR=ABS(TV):RETURN:REM
  MOVED TO NEW LOCATION
1070 SP$=CHR$(32): REM A SPACE
1080 P=INSTR(1,WD$,SP$): REM P
  GIVES POSITION OF SPACE IN
  INPUT
1090 IF P=0 THEN GOTO 1210: REM NO
  SPACE-NOT A VALID COMMAND
1100 VB$=LEFTs(WDs,P-1): REM
  MIGHT BE VERB
1110 IT$=RIGHT$(WD$,LEN(WD$)-P):
  REM MIGHT BE OBJECT
1120 VN=0: REM VERB NUMBER
1130 FOR I=1 TO NV: IF VB$=VB$(I)
  THEN VN=1: REM FOUND A VERB
1140 NEXT
1150 IF VN=0 THEN GOTO 1210: REM
  NOT A VALID VERB
1160 IT=0: REM OBJECT NUMBER
1170 FOR I=1 TO OB: IF IT$=OB$(I)
  THEN IT=1: REM FOUND AN OBJECT
1180 NEXT
1190 IF IT=0 THEN GOTO 1210: REM
  NOT A VALID OBJECT
1200 RETURN: REM ALL OK—BACK TO
  MAIN PROGRAM
1210 PRINT "I DONT UNDERSTAND":
  GOTO 1000: REM GENERAL PUR-
  POSE ROUTINE
```

This routine prompts the user for an entry. If a number between 1 and 3 is INPUT then

-VERBS-	-NOUNS-
TAKE	SPOON
LOOK	DRAGON
GRAB	PRINCESS
SAVE	GOLD
KILL	WITCH
ATTACK	KNIFE

In a simple adventure game, the basis for giving commands is a verb, then a noun. This usually takes the form of something like "Get, Sword" or "Kill, Dwarf" or some other equally violent combination of words. The commands are invariably given with the verb first and then the noun it acts on.

this is taken as a move, as earlier. If the INPUT doesn't have a space in it then I DON'T UNDERSTAND is displayed. And it will appear whenever the user enters verbs or objects that are not in the program's vocabulary. When the user makes a 'valid' entry, the routine RETURNS with VN as the number of the verb *ie* its position in VB$(); VB$ is the verb itself. Similarly, IT points to the location of IT$ (the item) in OB$(). Obviously once the routine has returned you will need to pass these numbers and strings to yet another subroutine to handle the different effects of each verb or check whether actions are allowed and so on.

If you don't have INSTR, you will have to replace line 1080 with the following routine to find a space in a string:

```
a P=0
  b I=1
    c IF MID$(WD$,I,1)=SP$ THEN
      P=I: GOTO f
d I=I+1
  e IF I< LEN(WD$) THEN GOTO c)
    f . . . REST OF ROUTINE . . .
```

A glance at figure 3 will show where you are up to. All that remains is to have the program act on the INPUT (section d) Figure 3

```
a INITIALISATION
  b GOSUB DISPLAY INFORMATION
    c GOSUB GET AND ANALYSE
      INPUT
d (ACT ON INPUT)
  e GOTO b
```

In fact, **d** turns out to be quite easy. All you require is an ON VN GOSUB . . . statement to pass control to the appropriate routine dealing with each verb. In this example it might take the form ON VN GOSUB 2000, 3000, 4000, 5000 . . . where you find at line 2000 a routine to handle 'taking', at line 3000 a section to handle 'leaving' an object and so on.

There are still a few loose ends to tie up. First is the question of determining the precise whereabouts in HW() of an item. You'll need to do this when the user wants to leave something behind, and you'll need to do something similar to establish whether or not an item is truly in a location if the player wants to take it. This was seen in the analysis of INPUT, where VB$() and OB$() are searched for the user's words. If the user has entered "TAKE FORK", and the coding has been done correctly, after the input analysis VN should be 1 and IT sould be 2. The routine at line 2000 will have to check first that the number 2 appears in row IR of OI(). This can be done as follows:

```
a PT=0: REM POINTER TO OBJECT
  LOCATION IN OI()
b I=1: REM ANOTHER POINTER — TO
  CELLS OF ROW IR IN OI()
c IF OI(IR I)=IT THEN PT=I: GOTO h:
  REM FOUND ITEM AT PT
d I=I+1
  e IF I<=MN THEN GOTO c
    f PRINT "BUT THERE ISN'T
      A";IT$;"HERE"
g RETURN
  h REM ITEM IS A CELL PT OF ROW IR
    i IF HW(0)=5 THEN PRINT
      "YOU'LL HAVE TO LEAVE SOME-
THING": RETURN
j and so on.
```

What is now a problem is the adventurer leaving something not at the 'end' of HW(). This means you'll have to add the following lines to the routine above to find the next free location in HW(), which will be shown by a zero:

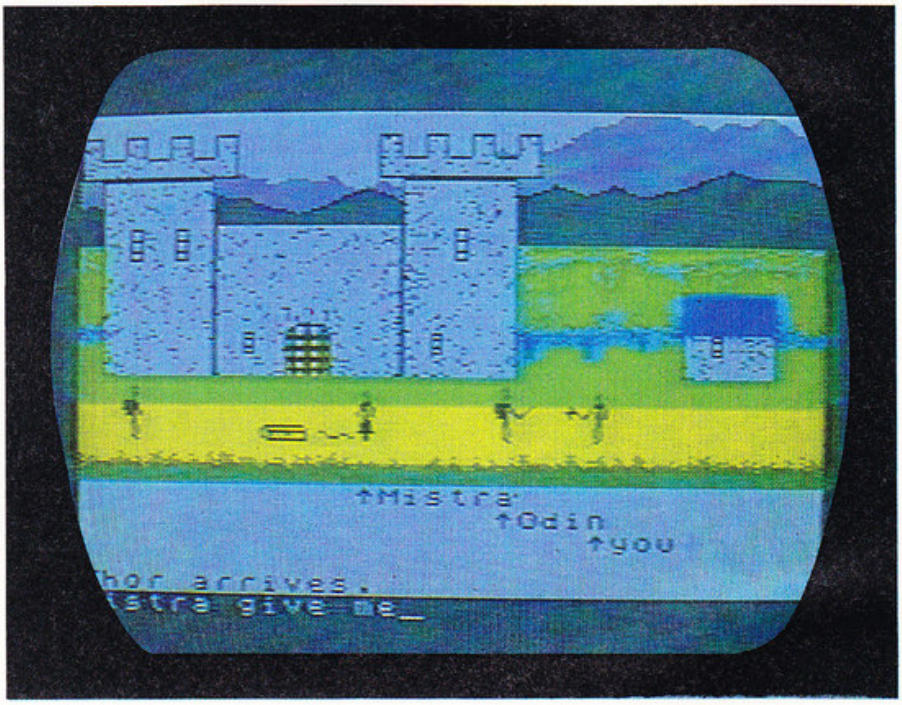
```
a NF=0: REM NEXT FREE CELL
  b I=1
    c IF HW(I)=0 THEN NF=I: GOTO e
      d I=I+1: IF I<6 THEN GOTO c
        e REM NF POINTS TO A FREE
          LOCATION IN HW()
f HW(NF)=IT: REM PUT ITEM IN HW —
  IE USER TAKES IT
g HW(0)=HW(0)+1: REM USER NOW
  HAS AN EXTRA ITEM
h OI(IR,PT)+0: REM REMOVE ITEM
  FROM LOCATION
i OI(IR,0)=OI(IR,0)−1: REM ONE LESS
  ITEM IN LOCATION
j RETURN
```

Of course a routine has to be added to the Screen display to let the user know what they're carrying.

The block diagram of the program now looks like this:

```
a INITIALISATION
  b GOSUB DISPLAY
    c GOSUB GET AND ANALYSE
      INPUT
d ON VN GOSUB 2000, 3000, 4000, . . . .
  e GOTO b
```

You should now be able to draw all these points together and recode the routines to suit your own ideas. If you've grasped the techniques, write the ultimate quest?



This is the Valhalla adventure game for the Spectrum – which represents "state-of-the-art" adventure gaming that combines both graphics and text to incorporate some "action" in the game.

Contributors: Ted Ball, David Janda and Bryan Skinner
Illustrators: Virginia Armstrong and John Hallett
Design: Nigel Wingrove
Micropaedia Editor: Geof Wheelwright

NEXT WEEK

We present a special one-part Micropaedia all about the life of computers and the small screen: Monitors. We'll tell you the difference between the three major types of computer monitors and take you inside the most popular type of monitor on the market: the ordinary home TV.

And in two weeks we begin a massive three-part buyer's guide to computer hardware, software and peripherals.