FVN

NOT TO BE REMOVED

RC 702 PICCOLO MICROCOMPUTER

PROBLEM ANALYSIS AND PROGRAMMING WITH C O M A L - STRUCTURED BASIC

A practical and educationally sound
approach to computing for people who
are non-mathematical or non-technological,
or both or neither.

By Roy Atherton.

CONTENTS

Chapter

1. The Background
2. Preliminary Ideas
3. Repetition
4. Decisions and Procedures
5. More Complex Structures
6. Handling Data
7. Complexity, Modularity, Procedures
8. Sequential Files
9. Direct Access Files
10. Case Studies
11. Problems and Projects

PROBLEM ANALYSIS AND PROGRAMMING WITH COMAL

By Roy Atherton.

"Ah love ! could thou and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits and then
Re-mould it nearer to the Heart's Desire."

- Rubaiyat of Omar Khayyam (Fitzgerald)

INTRODUCTION

In 1968 Edsger Dijkstra published a paper entitled "GOTO Statement Considered Harmful". That year also contributed to the name of a programming language called Algol 68. Bot these events were moves towards establishing principles for methodical computer programming which implies methodical problem analysis.

Perhaps the most significant event in this movement was the publication in 1971 of a definition for Pascal by Jensen and Wirth. This language was invented as a vehicle for teaching programming but its theoretical basis is so sound and its authors are such practical people that it became a major computer language inside and outside of educational circles in less than a decade.

To put this achievement in perspective one has to remember that there is a huge investment in existing languages and other software and it is not easy for new languages to break in. What are the strengths of Pascal that have made it popular ? Undoubtedly the most important quality of Pascal is that it incorporates exactly those features which enable and encourage methodical problem analysis and programming. The "Harmful GOTO" becomes almost redundant and the writing of a correct program in Pascal is almost certain to be of educational value to all but the very experienced and expert programmers.

Curiously an almost parallel development was the language BASIC (Beginner's All-Purpose Symbolic Instruction Code) invented by Kemeney and Kurtz in the mid-sixties. This language has such an easy syntax and simple vocabulary that it is cheap to implement, it can be used with the smallest machines, and it seems amazingly easy to learn. Typically a user would be surprised and pleased at his seemingly rapid progress in the early stages of learning to program, though he would be likely to become slightly puzzled at the difficulties experienced with even moderately complex problems.

The cheapness and deceptive simplicity of BASIC made it very popular with schools, small colleges and other low-budget organisations, and of course it was the "natural" language for the personal computer market based on the cheap microcomputers of the late seventies. Pascal, on the other hand, was very much a part of the mainstream of development of computer science in the seventies. It helped to spread those concepts which had been identified as fundamental to methodical problem analysis and programming. It has been suggested that these simple structural ideas can do for the structural aspects of computer programming what Newton's laws of motion did for mechanics - provide a simple and precise set of concepts which constitute a framework for the methodical analysis and solution of problems and also provide a sound basis for further theoretical discussion and development. The writer is aware of the power and universality of Newton's laws of motion (notwithstanding Albert Einstein) and the comparison may not be fully justified but certainly something important with an air of permanence has occurred in the world of computing. But was it a case of separate development - a kind of apartheid in computing ? Were the advocates of BASIC unaware of the other major stream of activity ?

/....

The answer is a resounding "yes". Certainly in UK and probably in most other countries. Teachers of computing in schools and colleges were usually mathematicians with a little knowledge of computing, perhaps Fortran (a precursor of BASIC) or Algol 60 (a precursor of Pascal) but they would quickly discover that only the brighter well-motivated pupils could cope with these languages. BASIC arrived like mannah from heaven and suddenly it seemed possible to teach computing and develop syllabuses and curriculum material for the lower age groups and the average ability pupils.

But some teachers were unhappy about BASIC. They noticed how easy it was for them and their students to get into a tangle. Borge Christensen of the Teacher Training College in Tonder, Denmark, tells of student, John, who asked for help in debugging a rather convoluted program. After some patient re-analysis the bug was discovered and so were two statements which had become redundant in the processes of amendment and re-amendment. The writer's own worst example of control paths which Dijkstra has likened to a plate of spaghetti is illustrated (A1).

The most successful attempt to deal with this problem - to diminish the gulf between progressive computing practice and school work - is COMAL. Christensen realised that Algol and Pascal were a bit too difficult for much of school work and decided to work on the enhancement of BASIC instead. His idea was to identify the minimum requirements for well-structured programs at school level and incorporate the necessary features into BASIC - to combine the simplicity and easy operating environment of BASIC with the structural strength, theoretical correctness and practicability of Pascal.

These ideas were published in 1975, just about the time that the program in A1 was discovered. In education the growing popularity of a seriously defective language, in the context of minimal teacher experience and negligible teacher training, created severe problems. Their severity was not less because it was obscured by the excitement and enthusiasm for this topical new subject.

In the second half of the nineteen seventies COMAL (Common Algorithmic Language) was developed, tested and refined until finally a standard version, COMAL 80, was agreed by a number of manufacturers and educationists based in Denmark but with good connections in other countries including USA and UK.

There are other moves towards structured BASIC such as the French LSE, ANSI structured BASIC and individual manufacturers' developments but a computer language is a work of art as well as a science and it is easy to get it out of balance - to do too little or too much or spoil it in some way. In the writer's opinion COMAL is right for the job though it will undoubtedly be developed further, probably on the data processing side. COMAL can do for school computing what Pascal has done elsewhere.

While the writer takes full responsibility for any ideas or opinions expressed in this work, he wishes to acknowledge with gratitude the debt he owes to many computer people on whose work he has drawn freely. In particular, to Edsger Dijkstra, Nicklaus Wirth, Ken Bowles, Peter Grogono, David Barron and Borge Christensen.

Chapter one

THE BACKGROUND

"There is always a simpler way."

- Tom Gilb, Computer Consultant.

1. THEMES AND ORGANISATION OF THIS BOOKLET

This booklet is written for anyone who wishes to make a start on learning computer programming. Following the ideas of Ken Bowles of the University of California, San Diego, pictures and text will be the main source of illustrations rather than mathematical concepts. This approach will obviously appeal to a wider audience. Mathematically inclined readers will have no difficulty in finding suitable problems for practice.

Topics are not treated in depth because the detailed rules of syntax can be safely left to a second stage. The first stage aims to cover enough syntax and technique to enable the reader to begin to develop the essential skills of programming and problem analysis almost immediately. This seems a more pleasant way of tackling the subject and is likely to maintain motivation. A second stage will return to previous topics and cover some new ones in a manner which aims at comprehensive coverage rather than easy learning though there is very little in COMAL that could be described as difficult. However, that does not preclude the solution of some extremely difficult problems in COMAL if that is necessary or desired.

In the earlier parts of stage one the reader can be assured of progress by following the text and typing everything enclosed in the rectangular boxes. This is intended to ensure that operational details are learned quickly and correctly by doing. Later, when such troublesome but essential details have been well treated a more natural format will be used.

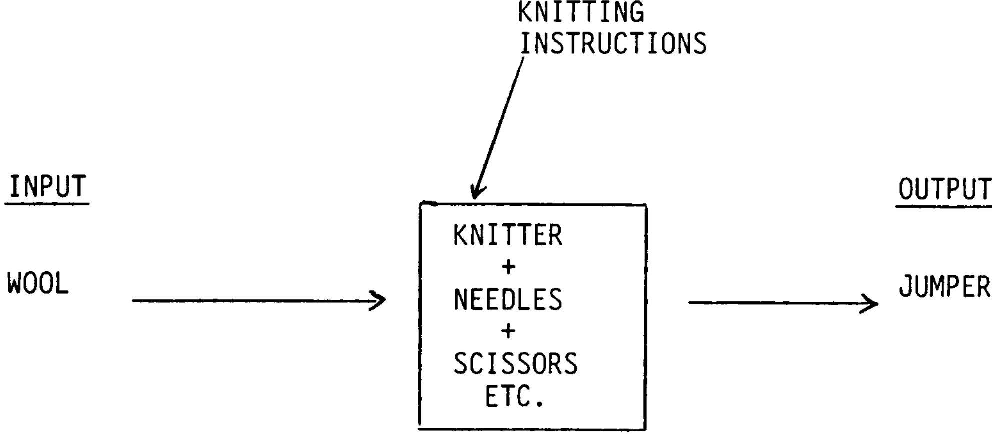
As will be explained in the next section, computer programs are sets of instructions for manipulating things such as pictures on screens, texts, files of information, accounts, simple or complex mathematics and so on. But it is much more pleasant and fruitful to learn the essentials by manipulating simple things even though this may seem slightly unpractical at the time. Accordingly the following items will be used in illustrative programs and exercises in most of this book.

Numbers
Random numbers
Alphabetic characters
Random alphabetic characters
Words and short sentences
Simple displays on the screen (lines and rectangles)

One final point follows the growing practice of differentiating between keywords and other words of a program with lower and upper case letters. The writer observes the Piccolo COMAL usage of putting keywords in lower case. Identifiers (names chosen by the user) are in upper case. This is simply to break up the solid appearance of a program and to make it easier to read and understand. The Piccolo system will force words of a program into this pattern so the user need not worry about the shift keys unduly.

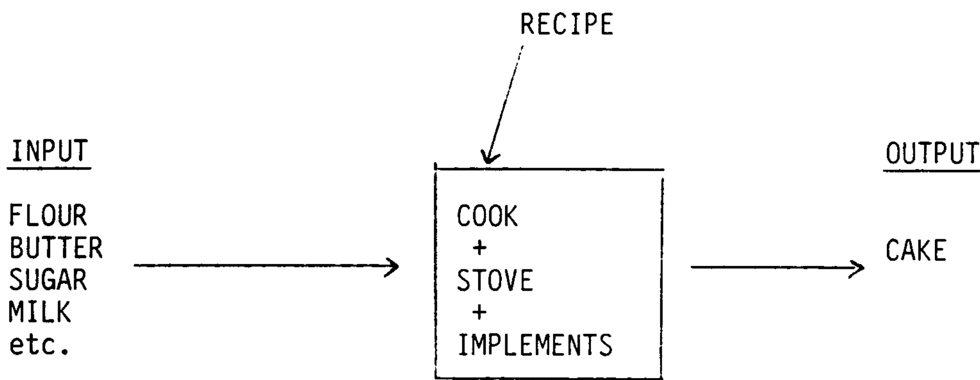
2. WHAT IS COMPUTER PROGRAMMING ?

A knitting pattern is carefully sequenced precise set of instructions for a knitter to operate on wool to produce, say, a woolly jumper.

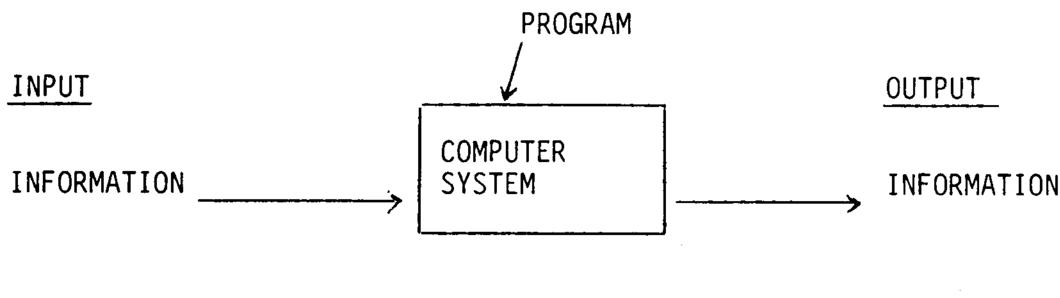


The knitter is programmed to process the wool (input) and turn it into a jumper (output). The pattern writer had to start with the idea of a jumper and break down the task of knitting it into subjobs until eventually he produced a set of instructions using concepts understood by the knitter. Such concepts would include the words PURL, CAST OFF, STITCHES, REPEAT, ROW and others.

A cook is similarly "programmed" by a recipe for a cake.



A computer program is analogous to the knitting pattern or recipe, but the input is information instead of wool or food,



and the output is information instead of a jumper or a cake. The output may be numbers, words, pictures, or data for the control of machines but the programmer must understand both the nature of the problem and the concepts in which the solution or program may be expressed. These concepts are embodied in the particular computer language in which the programmer will express his solution. Different languages would allow different ways of expressing the solution. For example many languages would allow a programmer to write something equivalent to :

sum ⟵ first + second

or in BASIC :

S = F + S

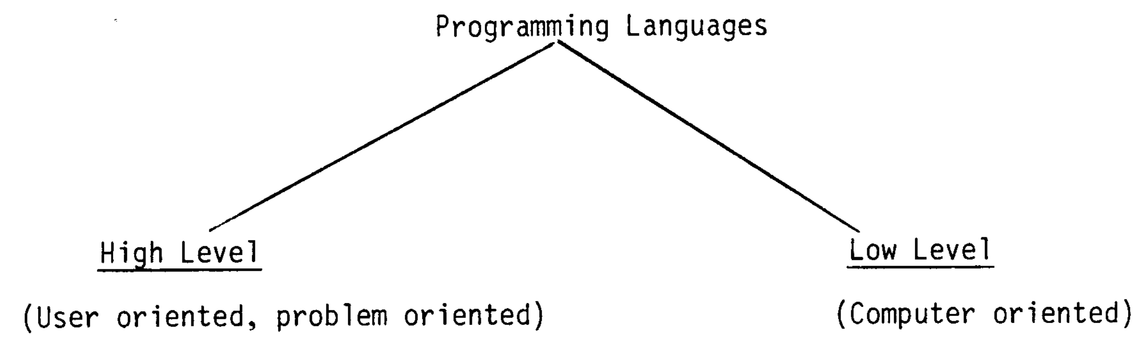
Both these statements would cause a computer to take two numbers from two memory locations, add them and store the result in a third location. They are user-oriented types of statement because a user could easily and naturally understand them. They are examples of "high-level" (user or problem-oriented) languages.

The same job could be programmed in a machine oriented language as follows,

```
LDA     FIRST
ADD     SECOND
STORE   SUM
```

This sequence of three separate instructions is closer to the machine that the user reflecting more closely the actual operations of a computer rather than a human person's natural way of thinking and expressing his thoughts.

It is important to understand that computer programming can occur at these different levels. But there are many more concepts in computer programming than simple addition and there are many design criteria for computer languages. The results of several decades' work has been to produce hundreds of languages in thousands of "dialects" and the most common way of attempting to classify them is illustrated below.



This is a crude classification and in fact there is a whole spectrum of languages. Some high level languages are higher than others in the sense that they provide more problem/user oriented concepts than others.

The point is laboured because the essential fundamental concepts necessary in the analysis of nearly all non-trivial problems were only identified firmly and positively about a decade after high level languages had begun to appear. Thus a language like Fortran (c.1958) is primitive compared with Pascal (c.1971) but quite a lot better than working with a machine oriented language.

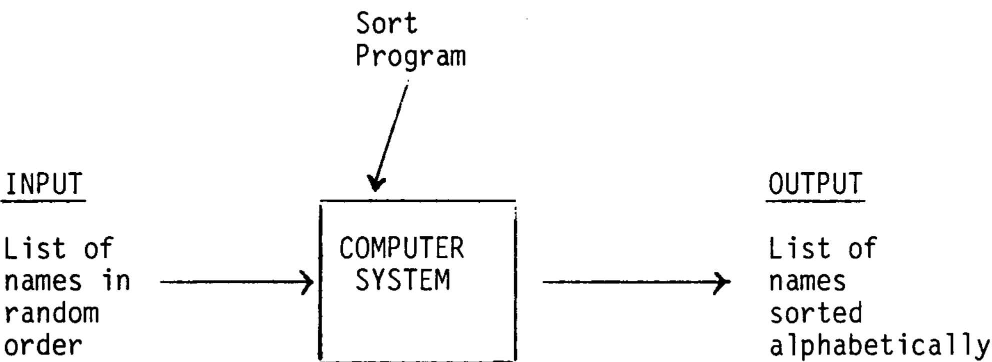
The purpose of the discussion is to establish that within the general classification of high level languages some are much more helpful than others. And one of the most profound ways in which a language can help a user is to allow him to articulate easily and explicityly exactly those constructs which are essential to programming. This is particularly important in an educational context. The details will be explored later but now we should consider the elements which are processed by a computer program.

By and large the items manipulated by most computers are numbers, text, graphics and control signals though there are other items such as sounds which may become increasingly important. Numbers and text (words) are familiar items and computer control of machines is outside the scope of this book but graphics need some discussion.

Generally speaking a programmer can draw his pictures in terms of points, straight lines and rectangles. He can draw them on special plotting devices or on a TV screen. Other facilities may be available to him such as "flashing" a portion of a screen or "fading" out a picture. Mathematical routines may help him to draw curves or rotate a picture giving a three dimensional effect. It all depends on the particular combination of hardware and software at his disposal.

In summary a computer program is a sequence of instructions written in a particular language. The instructions will cause a computer to process input information to produce a needed form of output information.

Example



A sort program can be written in many ways in many languages using a variety of styles. The following chapters will shed some light on the problem of making these choices appropriately.

3. GETTING READY

Follow the separate instructions for starting the computer and setting up COMAL. The stages are as shown.

	COMPUTER
1. SWITCH ON	OPERATING SYSTEM

The operating system becomes live and is used to load COMAL.

2. COMAL IS READY	OPERATING SYSTEM · COMAL

Control has passed to COMAL.

3. COMAL RECEIVES INSTRUCTIONS	OPERATING SYSTEM · COMAL · INSTRUCTIONS
4. COMAL EXECUTES INSTRUCTIONS	OPERATING SYSTEM · COMAL · INSTRUCTIONS

Notice that some parts of the operating system, COMAL, and possibly a program of instructions, can be stored simultaneously though all three cannot be "live" at the same instant. It is quite easy to move control between the three items of software as necessary.

For precise details of the starting procedure see appendix.

Chapter two

PRELIMINARY IDEAS

"It is possible to be a great computer (sic) without having the slightest idea of mathematics."

- NOVALIS, Schriften, Zweiter Teil,Berlin 1901.

1. PRINT statements

Type

```
print "Football Match" ↙
```

Notes

1. Type exactly what is in the rectangle now and throughout the booklet.
2. ↙ means RETURN key.

The computer should have obeyed the statement instantly but now type,

```
10 print "Football Match" ↙
```

This time nothing seems to happen but the computer has stored the information as part of a program. Now type,

```
run ↙
```

and the stored program will be executed.

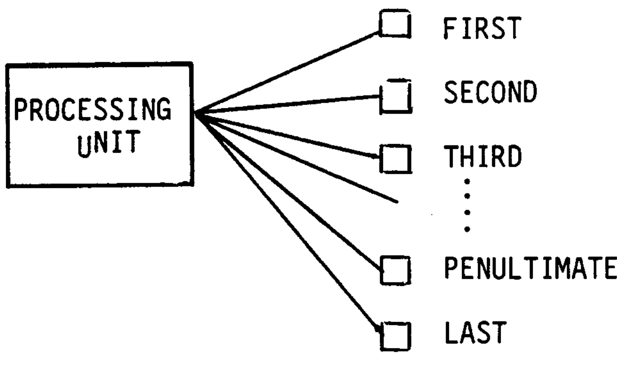
More about print statements later, but for the present note that you have written a PRINT statement.

print is a KEYWORD

"Football Match" is a STRING CONSTANT because it is a string of characters which cannot change. The quotes are used to define the beginning and end of the string. They are not strictly part of it.

2. NUMBER STORES AND ADDRESSES

A very simplified but useful view of the computer is to imagine it as a set of pigeon holes with addresses (chosen by the programmer) all connected to a processing unit.



/....

Each pigeon hole can store one number at any one time though it may change during the running of a program.

The instruction

```
let FIRST = 35 ↙
```

will cause the number 35 to be stored in the pigeon hole with the address FIRST. In order to check that this has happened type,

```
print FIRST ↙
```

(Note the absence of quotation marks)

This instruction, without quotation marks, means

"Copy the number from pigeon hole "FIRST" onto the screen"

Now type,

```
let SECOND = FIRST↙
print SECOND ↙
```

It is easy to infer that

SECOND = FIRST

means "copy the number from pigeon hole "FIRST" into pigeon hole "SECOND"."

The notation

FIRST ⟵ SECOND

would reflect the process better but we must use =. <u>let</u> may be omitted in the following program. The use of line numbers causes the instructions to be stored but not executed.

```
10 FIRST = 7 ↙
20 SECOND = FIRST↙
30 THIRD = FIRST + SECOND ↙
40 print FIRST, SECOND, THIRD↙
```

Before typing `run↙` write down what you expect as output from the program.

The addresses we have used are called numeric identifiers and the general form of the assignment statements is,

<numeric identifier> = <numeric expression>

We will not give a more formal definition but it is useful to think of such statements as

<address> ⟵ <sequence of operations>

Such a view will make statements like

COUNT = COUNT + 1

make sense. A conventional mathematical interpretation would be quite different.

An <expression> or <sequence of operations> can be very simple or quite complex as will be seen later.

3. WORD PROCESSING

Computers store and process words (strings of characters) just as easily as numbers but the storage arrangements are slightly different. If a programmer writes FIRST the system will allocate a standard amount of memory sufficient to store any number but a string of characters can be almost any length and the computer needs to be told how much storage to reserve. A dollar sign must be appended to the identifier of a string address for similar reasons. The instruction

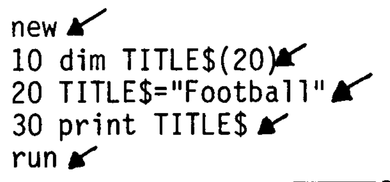
dim TITLE$(20)

means

"Set up a store for up to 20 characters and call it "TITLE$".

This is one example of dimensioning data storage, i.e. telling the computer how much space to reserve and how to organise it.

Type



<u>Note</u> <u>new</u> caused the old program to be deleted and made ready for a new one.

The computer should print "Football" but it has done a good deal more. Firstly it has stored the program. Secondly, during the running (execution) of the program it did three things.

<u>line 10</u> caused a portion of memory to be reserved for up to twenty characters and gave an address "TITLE$" to that list of memory. The $ indicates a STRING address rather than a number address.

TITLE$

<u>line 20</u> caused the characters "Football" in the form of codes to be stored.

TITLE$

Football

<u>line 30</u> caused the contents of the store with address TITLE$ to be printed.

<u>Notes</u>

1. TITLE$ is a STRING VARIABLE because it refers to a store for character strings which may change during the running of a program.

2. TITLE$ = "Football" is called an ASSIGNMENT STATEMENT because it assigns a "value" to the variable TITLE$. It could also be written

   let TITLE$ = "Football"

   and for this reason it is also called a <u>let</u> statement. The keyword <u>let</u> is optional whether numeric addresses or string addresses are involved.

Now type (do not type new)

```
15 dim GAME$(10),HEADING$(30)↙
30 GAME$="Match" ↙
40 HEADING$=TITLE$+" "GAME$↙
50 print HEADING$ ↙
list ↙
```

The complete program should be listed. Check it to see if it makes sense and then type run ↙. The computer should output

Football Match

4. EDITING

By typing lines with appropriate numbers we can add, insert or replace existing lines.

20↙ will delete line 20 and 100,200↙ will delete all lines from 100 to 200.

new↙ will delete a complete program.

The reader should experiment with the EDIT feature by typing, for example, EDIT 80. The four "arrow" keys can then be used to move the cursor, delete characters or create space for insertions. auto↙ will cause the system to provide line numbers automatically and the ESC key will terminate this.

5. RANDOM NUMBERS AND CHARACTERS

Type

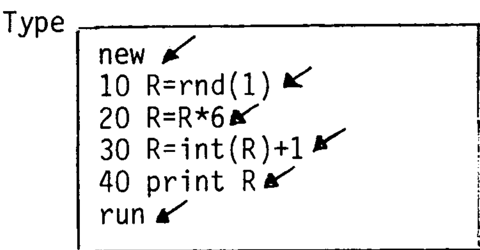
```
print rnd(1)↙
```

and a "random" number will appear. It will be in the range 0 to 1. (If it contains an E do not worry. This is a special form in which the number after E indicates the necessary shifting of the decimal point).

These raw random numbers are not very useful but we can tailor them to our needs. For example to simulate throws of a die we must do the following.

1. Multiply the random number by 6.
2. Cut off the fractional part.
3. Add 1

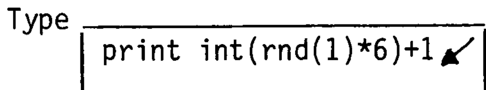
Type



```
new ↙
10 R=rnd(1) ↙
20 R=R*6↙
30 R=int(R)+1↙
40 print R↙
run ↙
```

All this can be combined in a single statement,

Type



```
print int(rnd(1)*6)+1↙
```

The psuedo random numbers are actually computed and a program will always produce the same sequence unless something like :

```
5 randomize ↙
```

is placed at the beginning of a program. The line number may be different.

6. RANDOM ALPHABETIC CHARACTERS

We can turn random numbers into random letters quite easily. The internal codes for the capital letters are :

A	65
B	66
:	:
Z	90

(see appendix A3).

The random number range must be 65 to 90 and this is easily achieved by

```
CODE = int(rnd(1)*26) + 65
```

In order to print the character corresponding to this code we must use chr(CODE).

Type

```
new ↙
10 CODE=int(rnd(1)*26)+65 ↙
20 print chr(CODE)↙
run ↙
```

7. STORING PROGRAMS ON DISCS

To save a program on disc you need to know (or decide) the program name which may be up to eight characters which must be letters or digits.

For example, to save a program called PROG1 on a disc, type

```
save PROG1↙
```

This program may be brought from the disc in main memory by typing,

```
load PROG1↙
```

A list of programs and other files on a disc may be obtained by typing,

```
lookup ↙
```

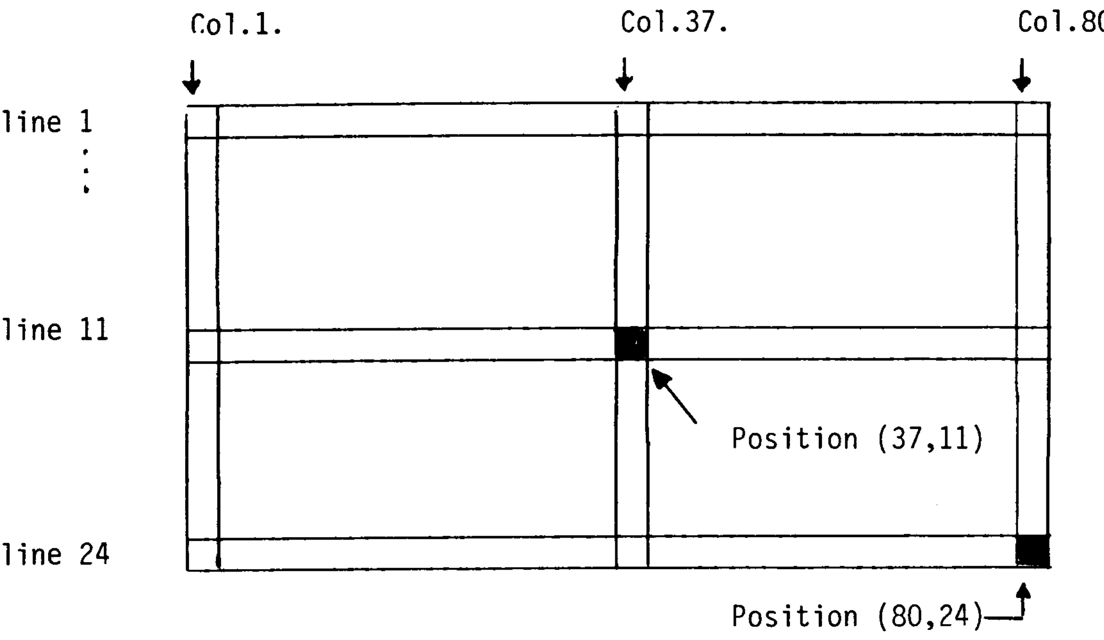
for the disc in the drive.

When using SAVE the user should ensure that an existing program is not unintentionally overwritten. THE SYSTEM DOES NOT CHECK.

/....

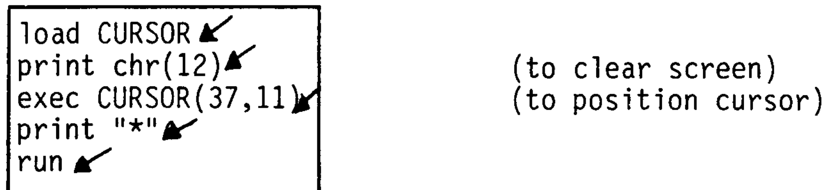
8. SCREEN DISPLAYS

The screen consists of 24 lines of 80 character positions as shown.



The special procedure CURSOR will be used for some simple introductory work with the screen. A fuller description of Piccolo graphics capability is given in appendix A4.

Load the program CURSOR and add lines as follows,



```
load CURSOR
print chr(12)                (to clear screen)
exec CURSOR(37,11)           (to position cursor)
print "*"
run
```

This should cause an asterisk to be plotted at position 37,11 on the screen. Any character may be placed in any screen position using this method.

9. INPUT

We have seen that numbers or strings can be sent into computer addresses by such statements as :

```
FIRST = 6
WORD$ = "blue"
```

A more flexible method of inputting data is to use a data statement (usually placed at the end of a program) together with read. The effects of the above two assignments could be achieved as follows.

Type

```
new ↙
10 dim WORD$(4)↙
20 read FIRST,WORD$ ↙
30 print FIRST;WORD$↙
40 data 6,"blue"↙
```

This is a trivial example but moderately large amounts of data may be handled in this way.

Exercises 1.

1. Use the auto command to help you to enter a program which does the following.
   (a) Print the words "First Program"
   (b) Store a random number in the variable "random"
   (c) Print the contents of "random"

   Use the ESC key to escape from auto mode.

2. Run the program and store it on a disc giving the program your initials as its name (no dots). Establish that the program has been stored by typing new ↙ before retrieving the program.

3. Edit the program to make it store a random letter of the alphabet in a variable "RANLET$" and print the result.

4. Write program which prints a row of ten A's starting at position (50,3) on the screen.

5. Delete the program using the line numbers (not new) and type list ↙ to show that the program is deleted.

6. Use a data statement to record the numbers 3,4,5. read them into three addresses FIRST, SECOND, THIRD and print their sum.

10. SUMMARY

We have dealt with three distinct types of communication with the computer.

(1) Statements like print 3 + 4 which are not stored but executed immediately.

(2) Control commands which are really part of the operating environment rather than language statements. These commands are run, list, new, edit and save. There are some others.

(3) Numbered statements which are stored as a program but not executed immediately. In this category are print and let.

We have also covered some introductory ideas.

(1) Data storage concepts relating to both numbers and strings of characters.

	NUMBERS	STRINGS
CONSTANTS	1 2 3.6	"Football" "Match"
VARIABLES (addresses, identifiers)	FIRST, THIRD	TITLE$ HEADING$

(2) Some simple operations on numbers (+,*) and strings (+).

(3) Random numbers and simple uses.

(4) Cursor control for screen displays.

(5) Simple editing of programs.

(6) Use of disc storage for programs.

(7) First ideas about input of data.

Chapter three

REPETITION

"Full thirty times hath Phoebus' cart gone round
Neptune's salt wash and Tellus' orbed ground,
And thirty dozen moons with borrow'd sheen
About the world have twelve times thirties been..."

- Player King in Hamlet

1. REPEATED OPERATIONS-1. FOR LOOPS.

The power of computers could not be exploited properly if every computer action was separately programmed. By programming the repetion of certain sequences a user can "magnify" his effort. Clearly, it will be necessary to specify very carefully :

(1) The start of the sequence
(2) The number of times it must be repeated (or some other way of ending the loop).
(3) The end of the sequence

It might be expected that a programmer should be able to write something like :

```
repeat 6 TIMES
print "This is a repeat"
END OF SEQUENCE
```

This is not so and a different construction is used,partly for historical reasons,and partly because experience has shown that it is often useful for the loop counter (which the computer must set up) to be made quite explicit by the programmer.

Type

```
new
auto
for COUNT=1 to 6
  print "This is number"; COUNT
next COUNT
ESC
run
```

Notes

1. The symbol for RETURN will no longer be placed at the end of each line. The reader will understand that the RETURN key must be pressed even though the symbol is omitted.

2. The for statement :

   (a) sets up a counter and defines its initial and final values;
   (b) defines the start of the repeated sequence (the next line).

3. The next statement simply defines the end of the repeated sequence.

4. Notice the "indenting" of the print statement. This is an instance of an important feature of COMAL. It has two advantages :

   (a) Readability of the program is enhanced.
   (b) In more complex work if successive indenting is not fully reversed by the end of the program an error is revealed.

   In the rest of this booklet indenting will be shown but the user need not provide it. After typing in a program list will reveal the correct indenting of each line.

Exercises 2.

1. The number of repetitions may be as few or as many as necessary. Use the edit 10 command to alter the 6 in line 10. Note especially the effects of making it 1 or 0.

2. In a football match simulation there are two halves and the overall structure of such a program might be :

```
for HALF = 1 TO 2
 <complex sequence of instructions>
next HALF
```

   Write and test a three line program to simulate this structure using one line such as print "Half a game" to represent the complex sequence of instructions.

3. Test the rnd(1) function by typing

```
new
auto
for CHANCE = 1 to 10
  print "Number"; CHANCE; "is"; rnd(1)
next CHANCE
ESC
run
```

   Note that the "random" numbers generated are in the range 0 - 1.

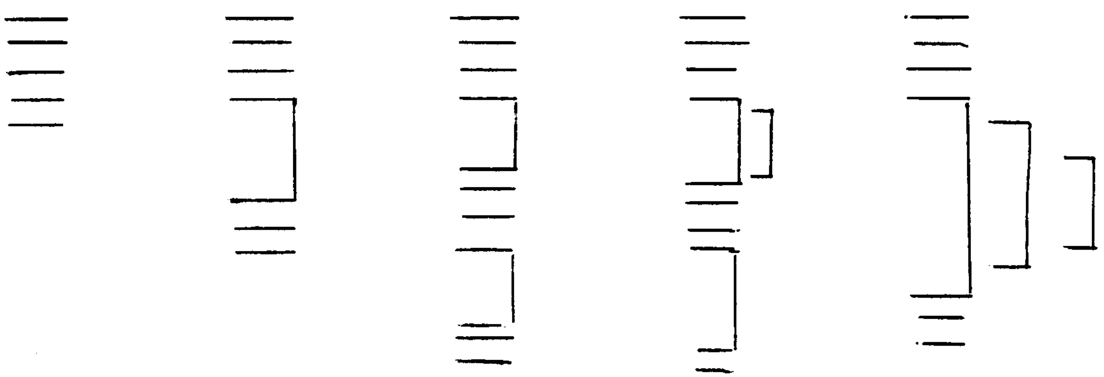
Exercises 3.

1. Alter the above program to draw a rectangle whose top left hand corner is position 30, 10 and size 20 units across, 5 down.

2. Alter the program resulting from exercise 1 so that instead of drawing 20 columns 5 units high it draws 5 rows 20 units across. Time the running of each program. Which is faster and why ?

3. PROGRAM STRUCTURE-1.

What has been achieved and understood so far is of far-reaching importance and the reader is now familiar with a number of elements of programming which have taken the form of simple sequences or loops. The relationships between these elements are also worth noting - they may be in succession or they may be "nested". If we denote linear sequences by a dash and loops by] then all the following are possible valid program structures.



Nesting may be as deep as one would normally wish though there are limits. The interesting thing about this is that, apart from a few exceptional circumstances, there is only one other necessary type of relationship between program elements, though there are several more types of element. One can create other more complex relationships in infinite variety but, unless there are special reasons, such things are unnecessary, hard to read and educationally obstructive.

4. REPEATED OPERATIONS-3. repeat.....until

Sometimes the end of repetitions of a sequence of instructions is decided by a condition rather than by some predetermined number. For example, no one except the referee knows exactly when the half time or final whistle will be blown. In our simulation we can use the rnd(1) function to introduce an element of randomness. We introduce a degree of realism by splitting each half into 540 five second time slices. In each slice an event such as pass, tackle, gain ground, shoot etc. can occur but after the 540 time slices there will be a few more to make up for stoppages at the referee's discretion. Eventually a point is reached where the game must end soon and we allocate, say, a 20% probability of the game ending in each time slice. We want to say something like :

```
repeat
  game continues
until end of half
```

The actual syntax is very similar.

Type

```
new
auto
repeat
  print "Event"
until rnd(1)< 0.2
print "End of half"
ESC
run
```

Repeated runs of the program will not produce different effects but if the word randomize is inserted at line 5 each run will produce an unpredictable effect.

Exercises 4.

The statement D = 6*rnd(1)+1 will produce some number in the range 1 to about 6.99 and store it in a pigeon hole with address "D". The function int(D) will chop off the fractional part. Thus the statement DICE =int (D) will produce some whole number in the range 1 to 6.

1. Simulate dice throws until a six appears, then stop.
2. Simulate throws of a pair of dice (as in monopoly) printing the total score each time until a double is thrown, then stop.
3. Repeat the experiment in (1) twenty times and find the average number of throws it takes to get a six.
4. The triangular numbers are :

.	.˙.	.˙.˙.	etc.
1	3	6	

   Find the first triangular number which is bigger than 1000.
5. Generate and print letters of the alphabet randomly until a Z appears, then stop.
6. Print the letters of the alphabet in sequence up to R.
7. What is the result of doubling 1 twelve times ?
8. Find the sum of the first ten natural numbers.
9. Write a program which prints the times of buses scheduled every fifteen minutes from 7.00 p.m. to 11.00 p.m.

Chapter four

DECISIONS AND PROCEDURES

"There's a rule saying I have to ground anyone who's crazy." ....
"Sure there's a catch. Catch-22. Anyone who wants to get out of combat duty isn't really crazy."

- Doc Daneeka in "Catch 22"

1. STRING VARIABLES AND STRING OPERATIONS

We know from long experience about numbers and operations with numbers, e.g. 7 x 3 = 21, and we know about string constants, e.g. "This is a string constant". It is now necessary to look at string variables (names and addresses reserved for storage of strings) and some operations with them.

We can reserve strings storage space by the statement :

```
dim TEXT$(100), WORD$(10), CHAR$(1)
```

This would allow us to store a modest sentence in TEXT$, a word in WORD$ and a single character in CHAR$.

The main operations with strings are concerned with :

joining strings
taking parts out of a string
putting parts into a string

We have already seen how strings may be joined. The idea of indexing provides a neat unified approach to the other two jobs. The Piccolo automatically provides each character in a string with an unseen index. Consider the string :

"A herb for a lamb chop"

placed in text$. The system will index it as follows.

A		h	e	r	b		f	o	r		a		l	a	m	b		c	h	o	p
1	2	3	4	5	6	.	.	.	.				14	15	16	17	.	.	.	.	22

We can copy the word "lamb" into WORD$ with the instruction

```
WORD$ = TEXT$(14:4)
```

which places in WORD$ the string obtained by starting at the 14th. position and counting 4.

The same technique can be used for placing something into a string. For example,

```
TEXT$(14:4) = "pork"
```

will change the string to :

A herb for a pork chop

There are three other useful functions for manipulating strings and characters.

len(TEXT$) would return the value 22 (length of string)

ord(TEXT$) would return, 65, the value of the code of the first character.

chr(65) would return the character "A"

Enter and run the following program.

```
dim TEXT$(100), WORD$(10), CHAR$(1)
TEXT$ = "A herb for a lamb chop"
WORD$ = TEXT$(14:4)
print WORD$
```

This should produce the word "lamb".
Now add the extra statements and run again :

```
TEXT$(14:4) = "pork"
print TEXT$
print len(TEXT$)
print ord(TEXT$)
print chr(66)
```

The output should be :

```
A herb for a pork chop
22
65
B
```

2. DECISION MAKING - IF...THEN...ELSE

We have already programmed the computer to make decisions because it has had to decide when to finish looping. A more explicit type ofdecision making is needed in programming as in everyday life : "If the weather is good I will walk, otherwise I will catch the bus". This sentence would be formalised in COMAL as :

```
if the weather is good then
  I will walk
else
  I will catch the bus
endif
```

But a computer cannot do these things, only simulate, so the actual syntax would be something like :

```
if WEATHER$ = "good" then
  print "Walk"
else
  print "Catch bus"
endif
```


/.....

Something in the preceding part of the program would determine the "value" of WEATHER$ and the decision would be made accordingly. As in loop structures the sequences of instructions may be of any length. For example PRINT "Walk" could be replaced by a number of statements, some of which might constitute a loop or even another _if_..._then_..._else_ construction.

Example

Search the string "A herb for a lamb chop" and print the indices of "lamb".

```
 10 dim TEXT$ (100),WORD$(10)
 20 TEXT$="A herb for a lamb chop"
 30 for TEST=1 to len(TEXT$)-3
 40   WORD$=TEXT$(TEST : 4)
 50   if WORD$="lamb" then
 60     print "Found at indices"; TEST; TEST+3
 70   else
 80     print "not found this time"
 90   endif
100 next TEST
```

3. PROBLEM ANALYSIS AND PROGRAM STRUCTURE

The "Lamb Chop" program is worthy of study not just for examples of programming techniques but also for its structure. The following points should be noted.

(1) The program consists of simple statements and "Control structures" (_for_ loop and _if_..._then_ structure)

(2) These elements are related to each other by being in sequence or by nesting. The _if_..._then_ structure is nested within the _for_ loop.

(3) Indentation indicates nesting and there are three "levels". The _dim_ statement is at level 1. The _if_ statement is at level 2 and the _print_ statements are at level 3.

(4) Every control structure must be opened (by _for_ or _if_) and closed (by _next_ or _endif_)

(5) After the opening keyword the indentation starts and it is reversed by the closing keyword.

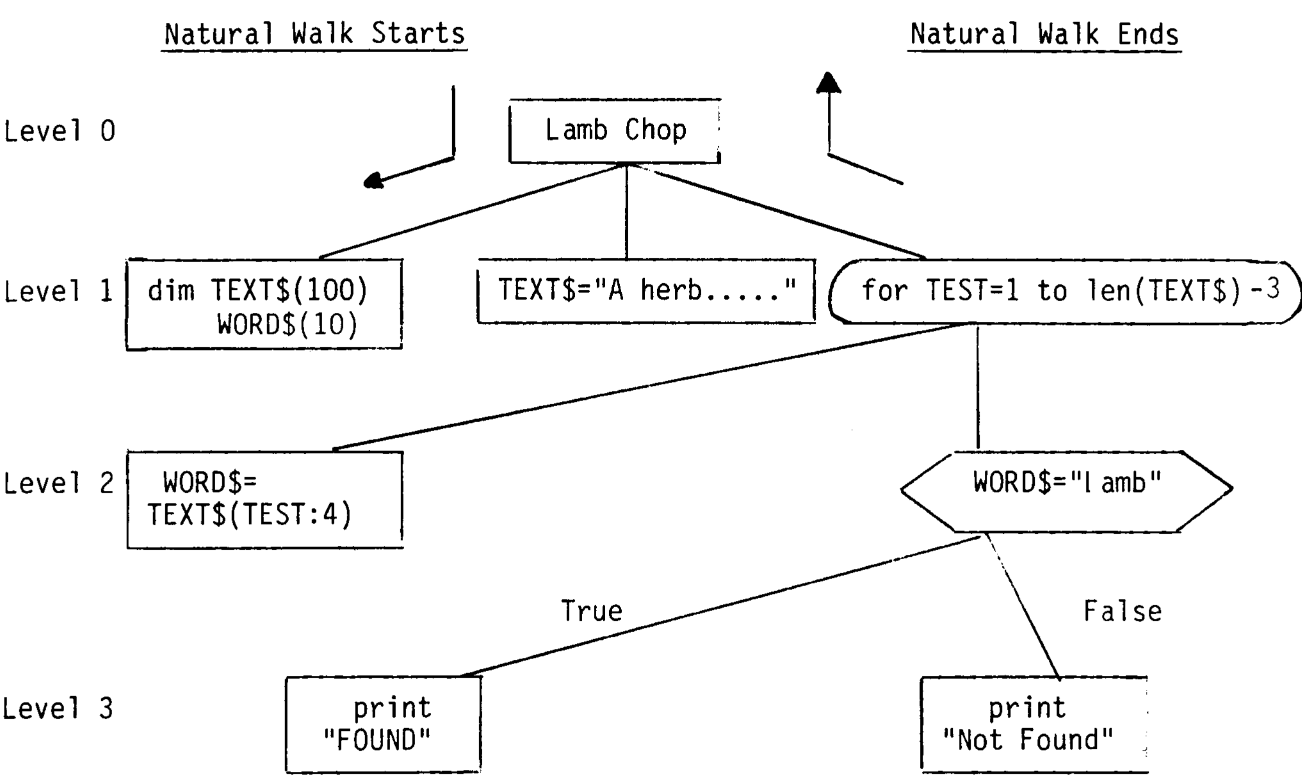
(6) In addition to making the structure visually apparent the system of indentation provides a check on correctness. The indentation should "come home" at the end of the program.

Readers with experience of other computer languages may have wondered why flowcharts have not been used yet. The answer is that the emphasis in COMAL (or any well-structured modern language) should not be on the flow of control but rather on the elements which constitute the program and the proper relationships between them. Once these concepts are understood, the analysis of a problem becomes a matter of identifying the necessary elements and fitting them together.

Programmers have been urged to use flowcharts to develop an algorithm but how often has the flowchart been written after the program ? And how often has the flowchart been updated along with program updates ? Flowcharts are useful for certain special jobs such as conveying complex procedures but for everyday programming they have serious drawbacks and they are unnecessary if one adopts systematic methods using the proper concepts.

Even so, there is much to be gained from some form of visual representation of an algorithm. In the early stages ideas may be encapsulated, tested and modified until they seem right. An outline of a job may be represented and details filled in later. Finally a correct details visual representation of an algorithm is more easily followed by the programmer or by someone else. A useful way of representing algorithms visually is by means of a structure diagram.

STRUCTURE DIAGRAM



Symbols and Terms

(1) A loop is represented by a box with curved ends

(2) A branching is represented by a lozenge shaped box

(3) Counting the program title as level zero the levels of the diagram correspond to the levels of indentation in the program.

(4) The diagram is technically known as a "tree" in computer science, graph theory or school mathematics.

(5) The lines are called "branches" and the boxes are called "nodes".

Programs from Diagrams

When learning to program some elements must be learned first before a tree has any meaning but later some programmers on some occasions would draw the diagram before encoding the program. If the diagram is available there is an easy way of writing the program by using the rules below.

(1) Follow a "Natural Walk" around the tree. Imagine all the lines are walls and walk around by starting at the title and keeping a wall always on the left as you "walk".

(2) When you come to a box or node write down a simple statement for a rectangular box.

(3) If the box represents a for structure you will come to it once on the way down and once on the way up. Write down the opening statement the first time and the closing statement the last time.

(4) If the box represents an if...then structure you will come to it three times corresponding to the three keywords if...else...endif.

Relationships

There is a little more to be learned about structure diagrams but the main ideas have been covered. However, it is worth emphasising that two types of relationships between elements of a program are exhibited clearly in structure diagrams.

(1) Sequential relationships are found by moving from left to right along the same level.

(2) Nesting is denoted by moving to the next level, down when the structure is opened and up when the structure is closed.

In summary the structure of a program should reflect its function in three ways.

(1) The program design should reflect the job it is performing.

(2) The coding should reflect this function. It should be easy to relate the coding to the original analysis of the problem. "Clever tricks" should not be used unless there is some specific advantage to be gained and if they are used they should be well explained.

(3) The presentation of the program should reflect its function. Headings, comments, remarks and the indentation help the reader to see more easily what the program does.

There is nothing really new about these ideas. Good programmers have followed them for years but the programming languages invented in the 1980s not only make it easier to follow the principles, they positively encourage systematic methods. Thus the people who program computers properly are themselves properly programmed with correct concepts and good methods.

Edsger Dijkstra summarised the main objective in 1968; "We should do our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

4. SIMPLIFYING A PROBLEM - PROCEDURES

The idea of splitting a complex job into a number of simpler jobs is fundamental in human affairs. Even the most egalitarian of politicians recognises the general responsibility of a prime minister and the division of this responsibility amongst his cabinet colleagues. They in turn will further subdivide amongst parliamentary secretaries and they amongst senior civil servants and so the process goes on tunil a job is simple enough to require no further sub-division. The same is true of the organisation of industries or colleges. Hierarchical systems are not a political theory so much as a practical way of handling complexity - making it manageable without losing the overview of the original major objectives.

Essentially a procedure is a self contained sub-program or subroutine which can be called by the main program to perform a particular sub-job. The lamb chop program can be rewritten using procedures for the two print statements. This is artificial but it will illustrate the syntax, vocabulary and structure diagram symbols for procedures in a context which is already understood.

Main Program

```
 10 dim TEXT$(100),WORD$(10)
 20 TEXT$="A herb for a lamb chop"
 30 for TEST=1 to len(TEXT$)-3
 40   WORD$=TEXT$(TEST : 4)
 50   if WORD$="lamb" then
 60     exec FOUND                    ─┐
 70   else                             ├─> PROCEDURE CALLS
 80     exec NOTFOUND                 ─┘
 90   endif
100 next TEST
110 stop
```

Procedure Definitions

```
120 proc FOUND
130   print "Found at Indices"; TEST; TEST+3
140 endproc
150 proc NOTFOUND
160   print "Not found this time"
170 endproc
```

Chapter five

MORE COMPLEX STRUCTURES

A Cambridge Don well known for his discriminating judgement of wines once observed that for many years water had never passed his lips. An undergraduate, thinking to catch him out, promptly asked : 'But, Sir, surely you clean your teeth ?' 'Yes, of course,' he replied, 'for my teeth I use a light Moselle.'

- Quotable Anecdotes,Leslie Missen.

1. WHILE LOOPS

A well known problem in computing is the "Drunken Duncan" simulation in which Duncan starts in the middle of a grid of squares and staggers, in a random manner, North, East, South or West to the next square. A first analysis of this problem might be :

```
repeat
  choose a direction
  move to the next square
until he is off the grid
```

This could equally well be achieved with a WHILE loop :

```
while he is on the grid
  choose a direction
  move to the next square
endwhile
```

Note the condition for exit is turned round so that we have :

```
while (reason for looping)
  (action)
endwhile
```

This contrasts with the repeat loop :

```
repeat
  (action)
until (reason for finishing)
```

Both loops can achieve the same result. Sometimes one seems more natural than the other and that is a good reason for choosing but sometimes the while loop will do what the repeat loop cannot. If the "reason for looping" were false at first entry to the loop no action would take place at all and control would go to the point after endwhile. If a repeat loop had been used the aotion would have taken place at least once even if this was not wanted. An example may clarify this.

Example

Write a program to simulate a person going to bed at midnight and getting up at some time after seven o'clock according to the following rules,.

(1) Count time in units of one minute

(2) Until exactly seven o'clock there is a one percent chance of the person waking up in any one minute but he goes to sleep again immediately

(3) After seven o'clock there is a five percent chance of his waking up in any one minute and when he awakes he gets up and the simulation ends.

Analysis

1. For simplicity we will count the time in minutes

12.00	-	12.01	1
12.01	-	12.02	2
	:		
12.58	-	12.59	59
12.59	-	1.00	60
	:		
6.59	-	7.00	420
7.00	-	7.01	421

2. The first part from time = 1 to 420 is just a for loop.

3. We may try to do the second part with a repeat/until loop and write something like the following.

```
for TIME = 1 to 420
  if rnd(1)< 0.1 then print "Woke up at"; TIME
next TIME
repeat
  TIME = TIME +1
until rnd(1)< 0.05
print "Got up at";TIME
```

Unfortunately this has a slight flaw. Due to the way the for loop works the value of time on exit is 421. The man has no chance of waking up at time 421 because this value is increased to 422 before the until statement causes a test to be made. We could reset time to 420 before entering the repeat loop but this is a well known type of situation and there is a standard solution for it.

The trouble is that a sequence in a repeat/until loop will always be executed at least once and there are many situations where there must be the possibility of the sequence not being executed at all. The right technique here is the while loop.

```
while rnd(1)> 0.05
  TIME=TIME+1
endwhile
print "Got up at"; TIME
```

The meaning of while must be carefully defined. It has the meaning "During the time that". For example "While your mother is out you must not open the front door". It does not have the meaning applicable in some parts of England : "I am waiting while (until) my mother gets home."

Notice also that the condition has been turned round. We say in effect

```
while he hasn't woken up
  increase the time count
endwhile
```

but in the other version we say

```
repeat
  increase the time count
until he wakes up
```

The while loop is a powerful concept and has the general form

```
while (reason for looping)
    sequence
endwhile
```

Generally speaking the repeat/until loop is a little easier to use and more natural to read, but there are situations in which a while loop is a more natural solution.

It is worth mentioning at this point that almost all repeat situations can be covered by choosing one of the for, repeat or while constructions. In all of these loops, on exit, control goes to the element which follows the structure. Exit is determined by a programmed number, by a condition tested at the end of the sequence or by a condition tested before the sequence is started. There are some special circumstances, such as aborting a program because of some error condition where a different technique is needed.

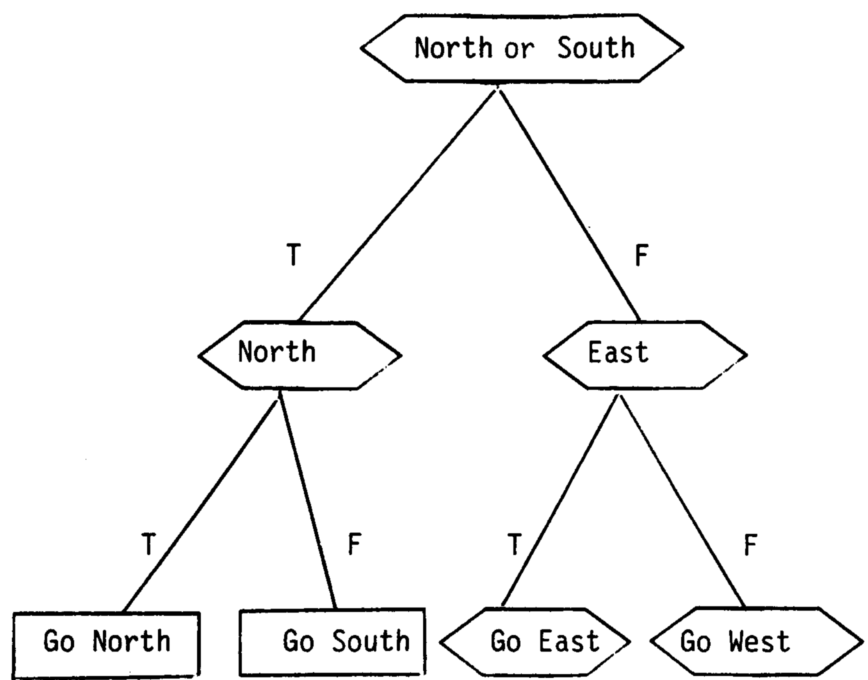
Exercise

Write and test a program to simulate the sleeper described in this section.

Investigate the value of time at different points in the program.

MULTIPLE DECISION MAKING-2. CASES

When there is a binary quality about a decision the if statement is the natural way to do it. Any decision with well-specified criteria could be split into a sequence of binary decision but the result might not reflect the nature of the problem. For example, if we wish to program the "Drunken Duncan" simulation, in which a man moves randomly from the centre of a grid of squares, we might group North and South together.

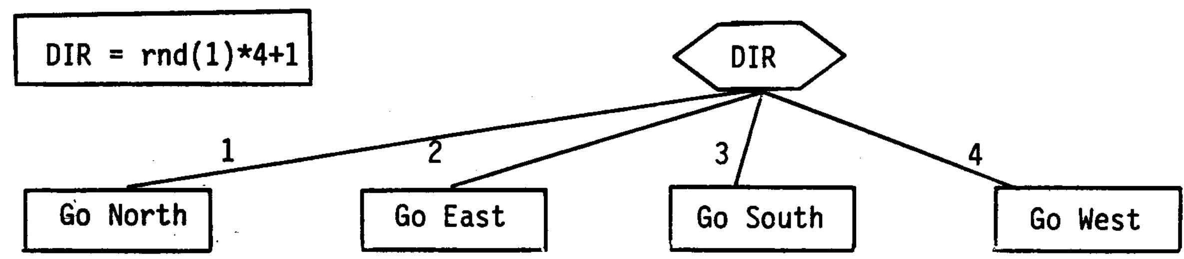


This can be programmed using the if/then/else construction, but the result is not natural to read because the decision required is one from many rather than two genuinely binary choices. A more natural construction* would be as follows.

```
DIR = rnd(1)*4+1
if DIR = 1 then
  Go North
endif
if DIR = 2 then
  Go East
endif
if DIR = 3 then
  Go South
endif
if DIR = 4 then
  Go West
endif
```

* The CASE statement of COMAL-80 is preferable when it becomes available.

The corresponding structure diagram is shown using the multiple decision concept.



The simulation can be watched by running a program which uses the PLOT routine.

1. Load CURSOR from the disc.
2. Add the following program starting at line 50.

```
AC = 40; DN = 10
print chr(12)                (clears screen)
repeat
  exec CURSOR (AC,DN)
  print "*"
  DIR = int(rnd(1)*4)+1
  if DIR = 1 then DN=DN-1
  if DIR = 2 then AC=AC+1
  if DIR = 3 then DN=DN+1
  if DIR = 4 then AC=AC-1
until AC<20 or AC>60 or DN<1 or DN>20
print "Duncan exits at position"; AC; DN
```

Notes

1. See appendix for explanation of CURSOR routine and the use of special character codes.

2. Duncan will always follow the same route in repeated runs but the use of randomize at about line 55 will make him more random.

3. Duncan can be made to show every move bv inserting a delay loop after exec CURSOR (AC,DN) and following the delay loop with print " ".

4. A delay loop is just a for statement followed by a next statement.

Chapter six

HANDLING DATA

"Number, the inducer of philosophies,
The synthesis of letters, ..."

- Aeschylus.

1. MORE ABOUT INPUT

We have seen some simple types of input from data statement, but handling data is about the avoidance of errors and the detection of the few that remain.

Type in the following program and run it.

```
for COUNT = 1 to 5
  read ITEM
  print ITEM
next COUNT
data 23,17,36,27,22
```

You will observe that the statement

```
read ITEM
```

causes a number to be taken from the data statement and placed in address ITEM. Data is taken in strict sequence every time a read statement is executed.

It is easy to set up and edit data statements, and COMAL (or BASIC) provides a natural facility, for handling small and medium amounts of data, which has no rival for simplicity in any other language. However, one must be ultra-careful about possible errors if our methods are to survive when applied to larger sets of numbers. The possible errors and their avoidance are listed.

	Error	Avoidance
1.	Wrong item	Use whatever validation methods are possible. e.g. check possible range of each number.
2.	Too few items	Try to know or make the program know exactly how many numbers to expect and use a for loop. Too few items will cause an error.
3.	Too many items	Place an item as a control at the end of the sequence. It should be recognisably different from the others - negative, zero or very large or an asterisk. Test for this item at the end of input.
4.	Compensating omission and later insertion of extra item	Break the sequence of numbers down into "records" and test end of record characters as well as end of file marker.

/.....

Example

```
for RECORD = 1 to 3
  for COUNT = 1 to 5
    read ITEM
    if ITEM>0 and ITEM<100 then print ITEM
  next COUNT
  read ENDREC
  if ENDREC = -1 then print RECORD; "OK"
next RECORD
read ENDFILE
if ENDFILE = 999 then print "File correct"
data 17,21,18,32,23,-1
data 16,27,31,19,24,-1
data 16,30,28,21,33,-1,999
```

Methods on these lines can be used successfully with substantial amounts of data but users must be aware that control and correction of errors is absolutely necessary.

data statements can also be used for non-numeric data.

```
dim ITEM$(10)
for COUNT=1 to 3
  read ITEM$
  print ITEM$
next COUNT
data "Blue", "Green", "Red"
```

Sometimes a program needs to interact with a user so that further processing takes account of the user's response. The input statement causes the program execution to pause so that data can be typed by the user. Three things are needed :

Program Pauses   Message to User   Data is received

and the corresponding statement is :

```
input "Please type a number and RETURN key", ITEM
```

The program pauses, the message appears and the user types, say, 18
The number 18 is stored in item and the program proceeds.

The following program tests arithmetic skill,

```
FIRST = rnd(1)*10; SECOND = rnd(1)*10
THIRD = FIRST + SECOND
print FIRST; "+"; SECOND;
input "=", ANSWER
if ANSWER= THIRD then print "Very Good"
```

The input statement may seem to be quicker than using read and data but it should only be used for interactive purposes. Whenever possible the read and data methods with careful error avoidance are preferable. In the long run this will take less keyboard time and provide an efficient and psychologically rewarding method of handling data.

One further useful input function is

key(0)

This will accept a single character from the keyboard without the need for pressing the return key. The function should be used in an assignment statement because the result must be stored somewhere.

example

```
print "Answer yes (Y) or no (N)"
ANS = key(0)
if ANS = 121 then print "Yes"
```

Notes

(1) The input character is not automatically displayed on the screen.

(2) The particular advantages of this function over the input statement are :

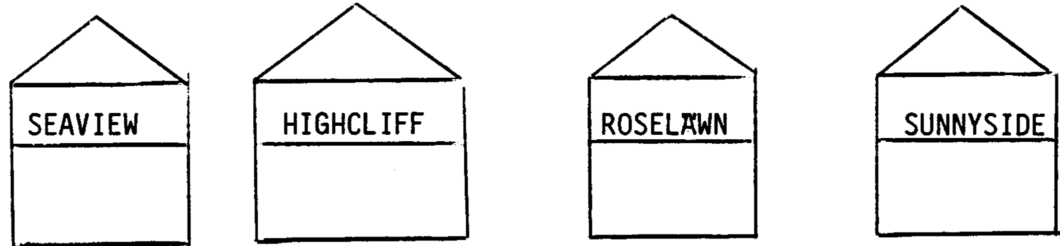
(a) No need for the RETURN key.

(b) Programmer chooses whether to display result.

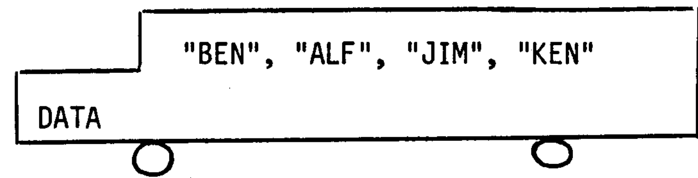
(3) The ASCII code of a character is stored in the function key(0). The ASCII code of "Y" is 121 (denary) as illustrated in the example above.

21. VARIABLES ADDRESSES - ARRAYS

Suppose we have four boarding houses in Brighton and a travel agent wishes to place four holiday makers, one in each

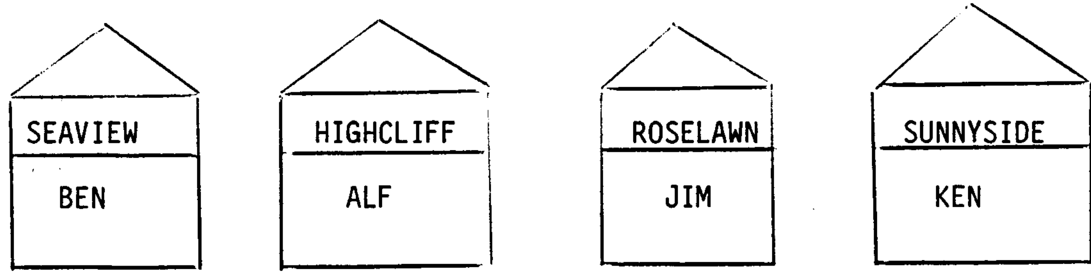


The holidaymakers arrive in a DATA statement.



The agent programs the holidaymakers into the houses.

```
read SEAVIEW$
read HIGHCLIFF$
read ROSELAWN$
read SUNNYSIDE$
```



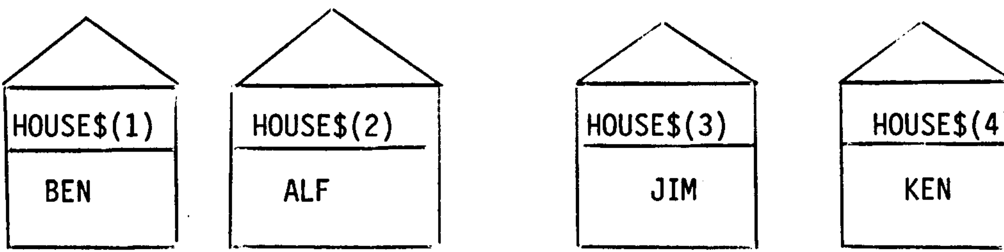
Thus we have assigned values (holidaymakers) to the variables (houses), but if there were 40 or 400 holidaymakers and houses a better method would be needed. We wish to do the same process to a sequence of houses - we want the address in the process to vary. An array enables this to be done. We declare four houses (string array) of three characters each.

```
dim HOUSE$(4,3)
```

Assuming the same data as before a simple for loop will place a holidaymaker in each house

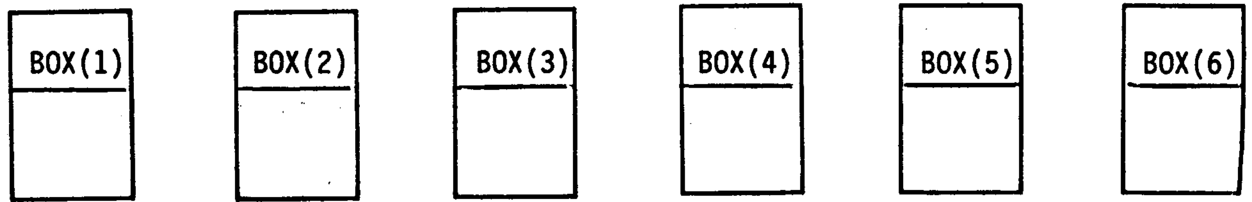
```
for NUMBER = 1 to 4
  read HOUSE$(NUMBER)
  print HOUSE$(NUMBER); " "; NUMBER
next NUMBER
```

The method can be applied to any number of houses. Try the complete program.



NB The notation HOUSE$(2) when referring to a string array address means the whole string.

Consider the numerical problem of simulating one hundred throws of a die and counting the frequency of each score. We set up an array of six (numeric) boxes,



When a score is generated we increase the count in the corresponding box. We could proceed as follows,

```
for COUNT = 1 to 100
  DIE = int(rnd(1)*6)+1
  if DIE = 1 then BOX(1) = BOX(1)+1
  if DIE = 2 then BOX(2) = BOX(2)+1
  if DIE = 3 then BOX(3) = BOX(3)+1
  if DIE = 4 then BOX(4) = BOX(4)+1
  if DIE = 5 then BOX(5) = BOX(5)+1
  if DIE = 6 then BOX(6) = BOX(6)+1
next COUNT
```

It is much easier to take advantage of the variable address concept more directly,

```
for COUNT = 1 to 100
  DIE = int(rnd(1)*6)+1
  BOX(DIE) = BOX(DIE)+1
next COUNT
```

Finally the program below reads ten numbers into an array and sorts them into ascending order. Write a short explanation of why it works.

```
for COUNT = 1 to 10
  read BOX(COUNT)
next COUNT
for RUN = 1 to 9
  for PAIR = 1 to 9
    if BOX(PAIR)>BOX(PAIR+1) then
      TEMP = BOX(PAIR)
      BOX(PAIR) = BOX(PAIR+1)
      BOX(PAIR)+1 = TEMP
    endif
  next PAIR
next RUN
for COUNT = 1 to 10
  print BOX(COUNT);
next COUNT
data 2,5,3,9,8,1,7,6,4,0
```


/......

This is called a bubble sort or a shuttle interchange sort. Its efficiency can be improved in two substantial ways but it is still about the slowest type of computer sort process and time taken increases approximately as the square of the number of items. Its merit is that it is very short and easy to understand.

The power of arrays lies in the fact that in using them we are able to vary addresses. We have seen from the early stages the idea of a memory location in which the contents vary. We see this in a conceptually convenient way as the value of a variable changing as the program runs.

An array enables the addresses (and therefore the memory location itself) to vary as the program runs. Thus we can make a process apply to whole sets of data quite easily. The idea of arrays seems comparable in importance to the stored program concept, repetition (loops), decisions or procedures (modularity).

Chapter seven

COMPLEXITY, MODULARITY, PROCEDURES

"The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty ?" he asked.
"Begin at the beginning", the King said, very gravely, "and go on till you come to the end: then stop."

- Alice in Wonderland.

It is not easy to discuss the handling of complex problems without actually dealing with a long job. Apart from logical problems or tricky bits of programming the sheer size of program makes it highly desirable to try to break it into sub-jobs.

The examples that follow, although not long or as complex as they might be, do illustrate methods of breaking down a job into smaller jobs and using procedures. There is a small overhead in using a procedure : at least the words exec, proc and endproc are extra. In these examples the results could be achieved by other methods but they are written in such a way as to illustrate the best ways of approaching larger, more complex programs.

The first example is short and, once a procedure for printing lines has been conceived and written the rest is easy. It seems unnecessary to use an elaborate analysis or a structure diagram. However, the second example is quite complex and will be explained in great detail. Structure diagrams will also be used. The reader should not be too concerned if some details are difficult to understand. What should be appreciated is the way the job is broken down into parts with clear relationships to each other.

Example 1. Numbers for Psychology Testing

A psychologist requires eight sets of random digits in the range 0 - 9 for short-term memory tests. In each set there should be ten lines of numbers starting with three to a line and increasing to twelve to a line, followed by another ten lines which start at twelve and decrease to three.

Analysis

1. Suppose we have a procedure LINDIG (NUM) which prints NUM digits on a line.

2. This procedure should be called by a for loop which varies NUM from 3 to 12 and then by a second for loop which does the reverse.

3. The whole should be with a forloop which does the job eight times.

Program

```
 10 randomize
 20 for SET=1 to 8
 30   for NUM=3 to 12
 40     exec LINDIG
 50   next NUM
 60   for NUM=12 to 3 step -1
 70     exec LINDIG
 80   next NUM
 90   print
100 next SET
110 proc LINDIG
120   for DIGIT=1 to NUM
130     print int(rnd(1)*10);
140   next DIGIT
150   print
160 endproc
```

Sample of Output

Two of the eight sets are shown

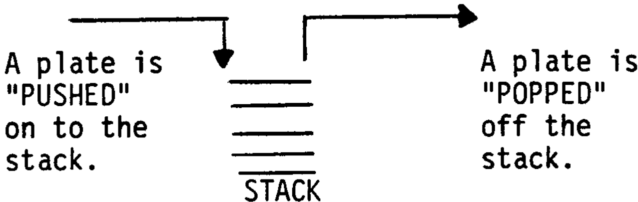
```
3 6 0
7 4 4 1
7 9 3 9 2
0 6 9 0 9 8
9 0 3 6 9 8 7
7 4 9 8 5 6 5 9
9 3 9 5 3 9 5 8 0
6 9 0 1 2 7 2 6 4 5
1 5 6 1 3 1 9 1 0 4 4
4 6 6 8 9 5 0 3 8 7 8 3
0 9 0 3 8 5 8 3 6 9 1 2
3 9 9 7 2 4 7 3 1 1 3
0 6 4 2 5 5 1 4 6 1
8 5 1 5 7 0 4 9 7
8 0 3 0 8 5 1 7
0 9 8 6 3 7 0
6 4 4 3 1 1
1 4 6 3 8
6 1 8 5
0 1 5

2 0 4
3 9 4 8
4 3 8 2 1
5 8 2 9 1 4
8 0 6 1 5 2 9
5 8 0 0 3 1 4 8
7 3 3 8 0 6 0 1 5
6 1 4 7 7 7 3 9 6 4
9 3 5 4 3 2 9 4 9 4 1
0 9 5 7 3 0 2 7 6 0 9 5
9 3 4 1 6 5 4 2 4 0 2 7
5 4 8 9 8 5 5 2 8 2 2
4 3 6 8 9 0 3 5 0 2
5 5 7 3 3 3 3 5 9
5 0 1 6 3 2 1 2
0 6 2 3 3 9 9
7 0 0 4 6 6
2 6 0 3 5
3 0 6 2
8 0 3
```

Example 2. Quicksort

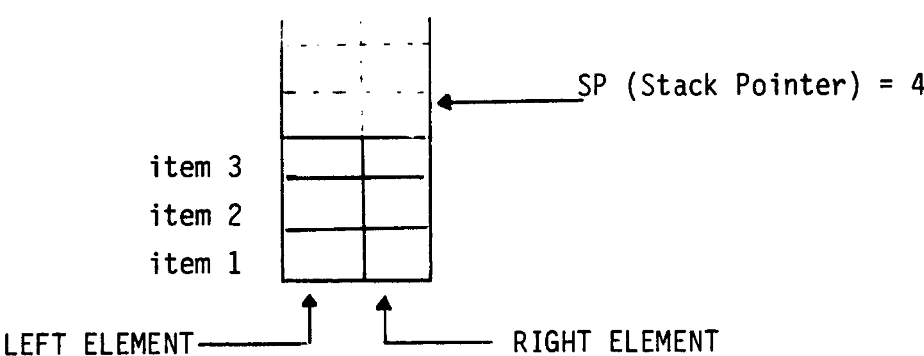
Bubblesort even with various enhancements is not very practical because the time taken increases with the square of the number of items. Thus it takes 100 times longer to sort 100 items than to sort 10 items.

A number of sort processes based on the idea of exchanging elements have been used and documented. Quicksort is one of the most interesting and is claimed to be one of the fastest available.

First it is necessary to understand the idea of a "stack". A stack is an array of data which behaves like a stack of plates - the last on the stack is the first off.



In quicksort our "plates" will be pairs of numbers which define the start and end of sub-arrays of the main array of names to be sorted.



The stack pointer (SP) will always point at the next empty place on the stack which will be a two dimensional array

STACK (20,2)

This short program shows the working of a stack in such a way that the user can PUSH pairs of numbers on to it, POP them off, or finish by running out of numbers on the stack. There will be two procedures PUSH and POP.

```
proc PUSH
  INPUT "Enter two numbers",LEFT,RIGHT
  STACK (SP,1) =LEFT
  STACK (SP,2) =RIGHT
  SP = SP+1
endproc
```

```
proc POP
  IF SP=1 THEN STOP
  SP = SP-1
  LEFT = STACK(SP,1)
  RIGHT = STACK(SP,2)
  PRINT LEFT,RIGHT
endproc
```

The main program simply establishes the stack, sets the stack pointer and inputs commands from the user.

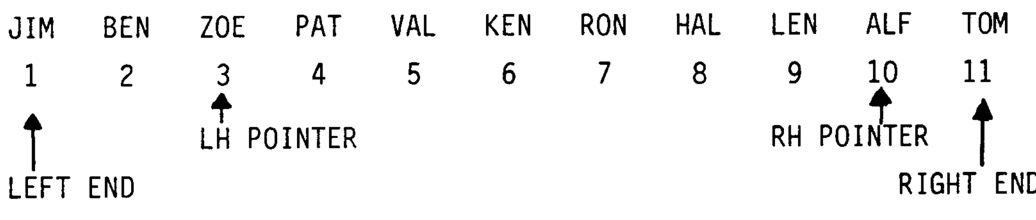
```
dim STACK(20,2), OP$(4)
SP = 1
repeat
  input "POP or PUSH?"; OP$
  if OP$ = "POP" then exec POP
  if OP$ = "PUSH" then exec PUSH
until 2 = 1
```

Program

```
 10 dim STACK(20,2),OP$(4)
 20 SP=1
 30 repeat
 40   input "POP OR PUSH ? ",OP$
 50   if OP$="POP" then exec POP
 60   if OP$="PUSH" then exec PUSH
 70 until 2=1
 80 proc PUSH
 90   input "Enter two numbers. ",LEFT,RIGHT
100   STACK(SP,1)=LEFT
110   STACK(SP,2)=RIGHT
120   SP=SP+1
130 endproc
140 proc POP
150   if SP=1 then stop
160   SP=SP-1
170   LEFT=STACK(SP,1)
180   RIGHT=STACK(SP,2)
190   print LEFT,RIGHT
200 endproc
```

The Quicksort Method

Suppose we wish to sort the eleven names into alphabetical order.



/.......

1. Choose a "comparator" KEN by taking the number ½ (LEFT END + RIGHT END) = ½(1+11)=6

2. Move LH POINTER from LEFT END until it encounters a name which should go to the <u>right</u> of KEN

3. Move RH POINTER from RIGHT END until it encounters a name which should go to the <u>left</u> of KEN

4. Exchange ZOE and ALF and continue moving pointers, exchanging where necessary, until the pointers cross

5. PAT will exchange with HAL

6. VAL will exchange with KEN and we have :

JIM	BEN	ALF	HAL	KEN	VAL	RON	PAT	LEN	ZOE	TOM
1			4		6					11

We have divided the original setinto two subsets 1 to 4 and 6 to 11 with KEN in his correct position because everything to his left will stay to his left and everything to his right will stay to his right whatever further shuffling might occur.

7. We place the right hand set pointers on a stack (6 and 11).

8. Continue the process with the left hand set.

9. If there is no left hand set (i.e. it is down to one element) pop a pair of numbers of the stack and continue with them.

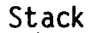
10. Stop when the stack is empty.

<u>List of Identifiers</u>

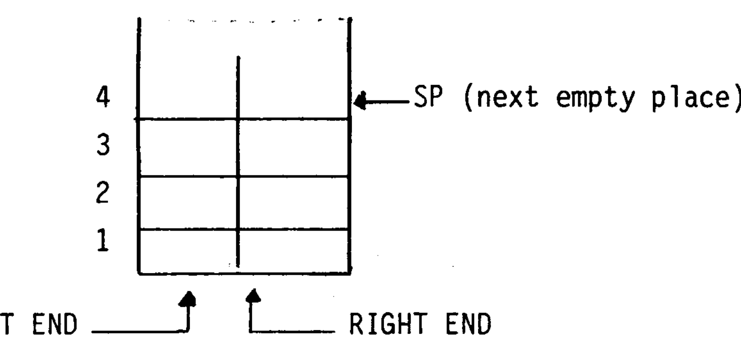
LEND,REND	Left and Right ends of sub-array
LHP,RHP	Left and Right pointers
LEFT,RIGHT	Left and Right items on stack
COMP$	Comparator
STACK(20,2)	Stack
ITEM$(11,3)	Array of names
SP	Stack Pointer
NUM	Number of names

<u>Procedures</u>

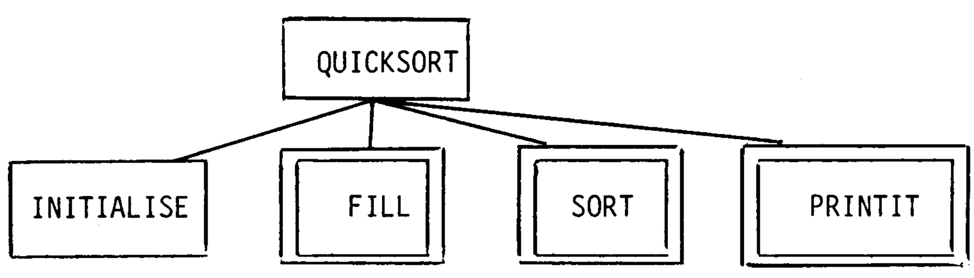
FILL	Fills array
SORT	Sorts array
PRINTIT	Prints sorted array
POP	Pops two numbers off stack
PUSH	Pushes two numbers on to stack
SWAP	Exchanges two items

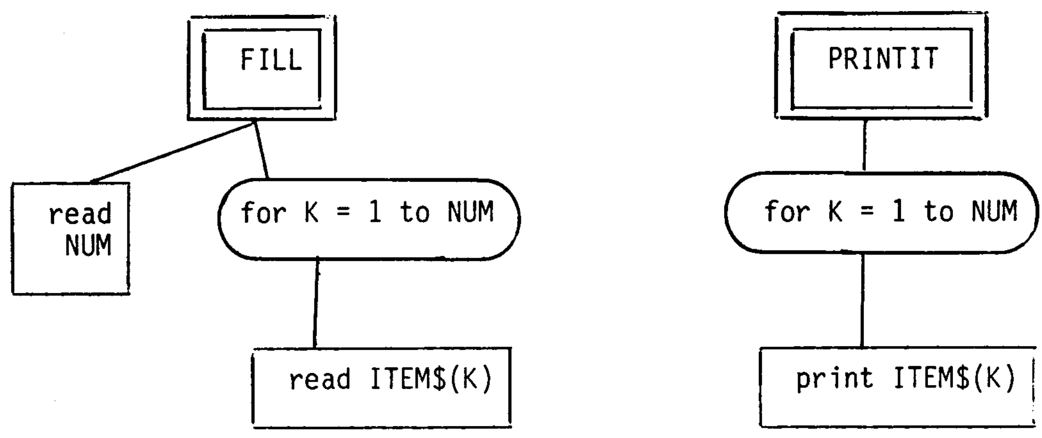


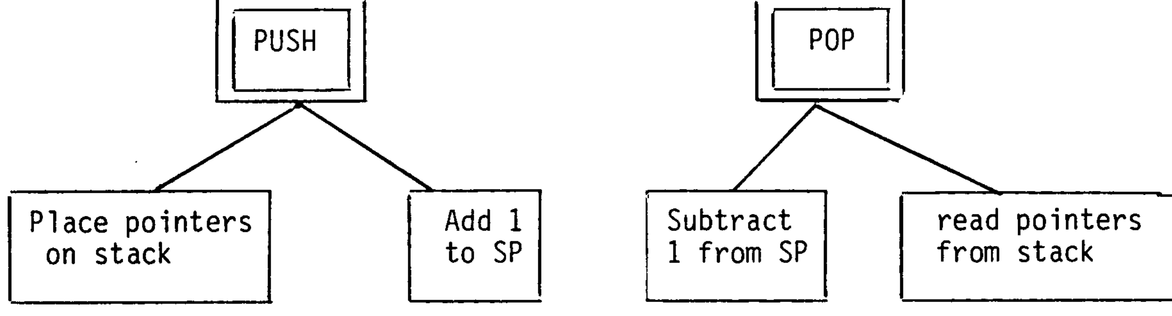
Stack



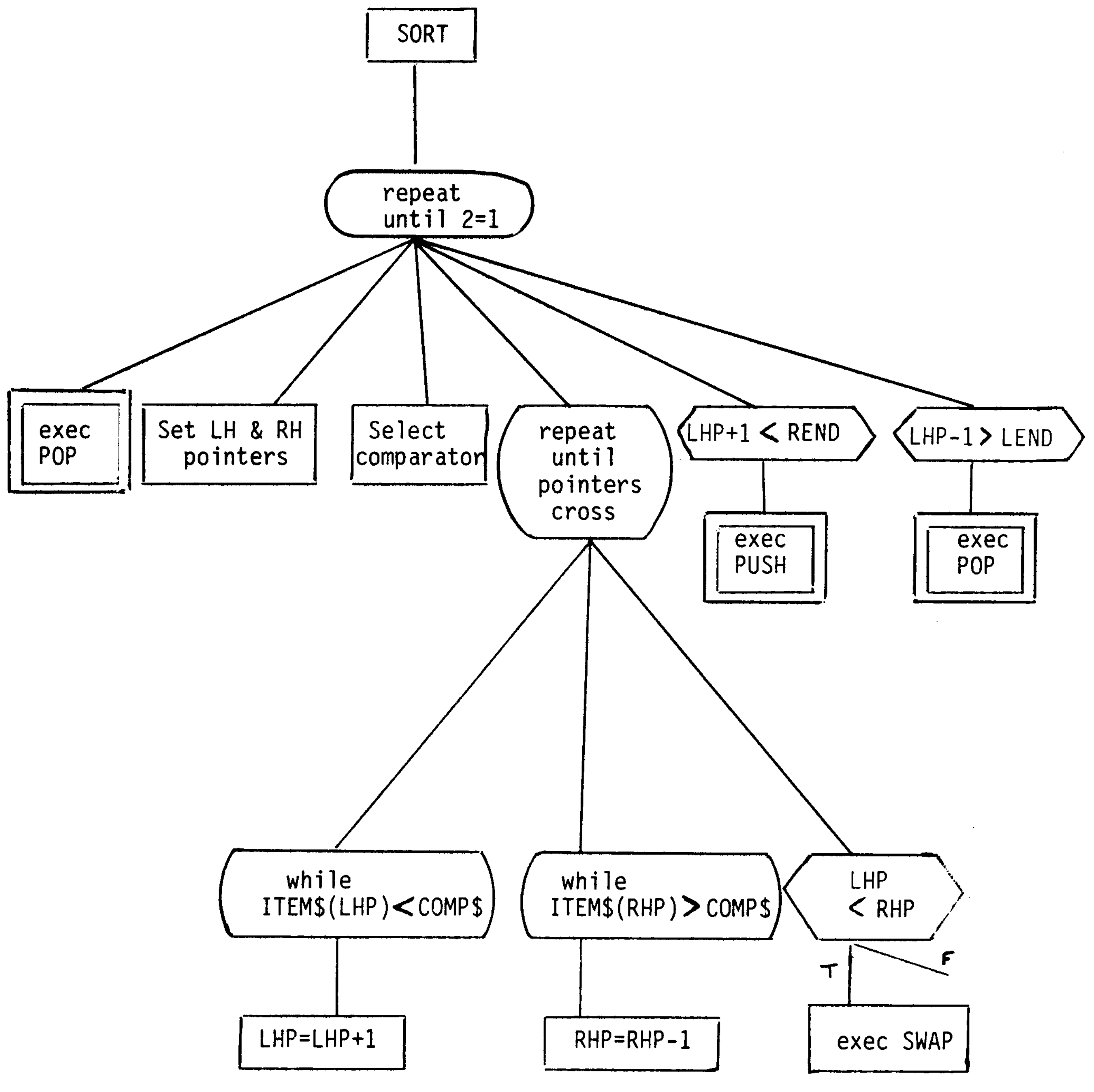
Structure Diagrams







.....



The detail in the above diagrams varies. Some details have been left until the program because it seemed more natural to do so.

/......

Program

```
 10 dim ITEM$(11,3),COMP$(4),STACK(20,2),TEMP$(4)
 20 exec FILL
 30 SP=1; LEND=1; REND=NUM
 40 exec PUSH(1,NUM)
 50 exec SORT
 60 data 11, "JIM","BEN","ZOE","PAT","VAL","KEN"
 65 data "RON","HAL","LEN","ALF","TOM"
 70 stop
 80 proc FILL
 90   read NUM
100   for K=1 to NUM
110     read ITEM$(K)
120   next K
130 endproc
140 proc SORT
150   repeat
160     exec POP
170     LHP=LEND; RHP=REND
180     COMP$=ITEM$((LEND+REND) div 2)
190     repeat
200       while ITEM$(LHP)<COMP$ do
210         LHP=LHP+1
220       endwhile
230       while ITEM$(RHP)>COMP$ do
240         RHP=RHP-1
250       endwhile
260       if LHP < RHP then exec SWAP
270     until LHP= RHP
280     rem *** stack pointers to right sub array ***
290     ifLHP+1<REND then exec PUSH (LHP+1,REND)
300     rem *** stack pointers to left sub array ***
310     if LHP-1> LEND then exec PUSH(LEND,LHP-1)
320   until 2=1
330 endproc
340 proc POP
350   if SP=1 then exec PRINTIT
360   SP=SP-1
370   LEND=STACK(SP,1); REND=STACK(SP,2)
380 endproc
390 proc SWAP
400   TEMP$=ITEM$(LHP); ITEM$(LHP)=ITEM$(RHP); ITEM$(RHP=TEMP$

420 endproc
430 proc PUSH(LEFT,RIGHT)
440   STACK(SP,1)=LEFT; STACK(SP,2)=RIGHT
450   SP=SP+1
460 endproc
470 proc PRINTIT
480   for K=1 to NUM
490     print ITEM$(K); " ";
500   next K
510   stop
520 endproc
```

```
ALF   BEN   HAL   JIM   KEN   LEN   PAT   RON   TOM   VAL   ZOE
```


/....

Comments

While Quicksort is a simple idea, what happens at the end of a run, when the pointers meet, sometaimes causes problems. The above program is written in such a way that the pointers never cross and at the end of a run both LHP and RHP will point at the comparator which divides the data into two subsets. However, the program fails for data which contains a number of repetitions. For example, it would sort A,A,A,A,A,B,A,A,A,A,A but the program would not stop because on the second run the pointers would never meet.

This problem is discussed by Wirth[1] and he gives a solution which involves swapping elements which are equal and moving the pointers on so that they do not always point at the comparator at the end of a run. The writer prefers the program as given with a small amendment to cure the problem of a number of identical elements,

(1) Change line 270 to read

270 until LHP => RHP

(2) Add

265 if ITEM$(LHP) = ITEM$(RHP) then RHP=RHP-1

These amendments force at least one pointer to keep moving when otherwise both might get stuck. However, a full discussion of these processes is beyond the scope of this treatment.

The algorithm with the amendment works for the data sets

ABCDEFGHIJK

KJIHGFEDCBA

AAAAABAAAAA

It has also been tested with 100 randomly generated four-character items. The program given on the next page uses Wirth's algorithm.

Reference

(1) Wirth, N. Algorithms + Data Structures = Programs, Prentice Hall,1976.

```
 10 dim ITEM$(11,3),COMP$(4),STACK(20,2),TEMP$(4)
 20 exec FILL
 30 SP=1; LEND=1; REND=NUM
 40 exec PUSH(1,NUM)
 50 exec SORT
 55 exec PRINTIT
 60 data 11, "JIM","BEN","ALF","HAL","KEN"
 65 data "VAL","RON","PAT","LEN","ZOE","TOM"
 70 stop
 80 proc FILL
 90   read NUM
100   for K=1 to NUM
110     read ITEM$(K)
120   next K
130 endproc FILL
140 proc SORT
150   repeat
160     exec POP
165     repeat
170       LHP=LEND; RHP=REND
180       COMP$=ITEM$((LEND+REND) div2)
190       repeat
200         while ITEM$(LHP)<COMP$ do
210           LHP=LHP+1
220         endwhile
230         while ITEM$(RHP)> COMP$ do
240           RHP=RHP-1
250         endwhile
260         if LHP < =RHP then exec SWAP
270       until LHP> RHP
280       rem *** stack pointers to right sub array ***
290       if LHP<REND then exec PUSH(LHP,REND)
300       REND=RHP
320     until LEND= > REND
325   until SP=1
330 endproc
340 proc POP
350   if SP=1 then exec PRINTIT
360   SP=SP-1
370   LEND=STACK(SP,1); REND=STACK(SP,2)
380 endproc
390 proc SWAP
400   TEMP$=ITEM$(LHP); ITEM$(LHP)=ITEM$(RHP);ITEM$(RHP)=TEMP$
410   LHP=LHP+1; RHP=RHP-1
420 endproc
430 proc PUSH(LEFT,RIGHT)
440   STACK(SP,1)=LEFT; STACK(SP,2)=RIGHT
450   SP=SP+1
460 endproc
470 proc PRINTIT
480   for K=1 to NUM
490     print ITEM$(K); " ";
500   next K
510   stop
520 endproc
```


/.....

Chapter eight

SEQUENTIAL FILES

"And it must follow, as the night the day."

Polonius to Laertes in "Hamlet".

It is usual to distinguish between sequential files and direct access files. An example of the former might be a file on a cassette in which the records can only be accessed in sequence. For example to find the seventeenth record of a cassette file it may be necessary to read the first sixteen as well even if the information is not used.

The records in a Piccolo disc file are automatically numbered and can be accessed directly by using the required number. Clearly a direct access file with numbered records can also be accessed sequentially and the idea is useful. The first example is a simple process of placing "records" in sequence on a file. Each record will consist of the letters :

ABCDEFGHIJ

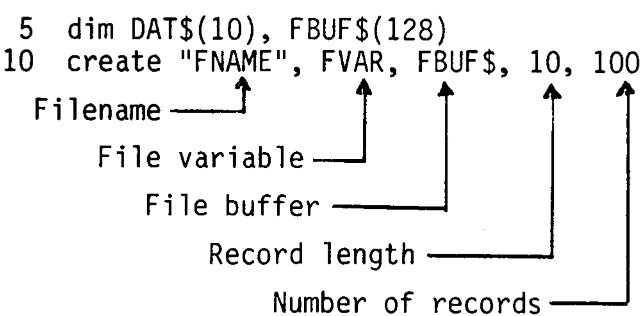
and we will need a string variable DAT$(10) to hold this ten-letter record.

We must create the file by providing the system with the following information.

1. The filename
2. A file variable to record the "status" of the file and to indicate any errors.
3. A file buffer of 128 bytes. This will be used by the system and should not be used by the programmer while the file is "open".
4. The size (in bytes) of a record.
5. The number of records in a file.

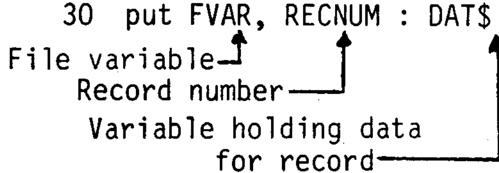
The names of the first three items will be chosen arbitrarily as : FNAME, FVAR and FBUF$ and we will establish a file of 100 records of 10 bytes each. The two program lines will do this :

```
 5  dim DAT$(10), FBUF$(128)
10  create "FNAME", FVAR, FBUF$, 10, 100
```



The next thing the program does is place, say, 90 records on the file. This requires the put construct :

```
30  put FVAR, RECNUM : DAT$
```



Finally we must close the file. The system will ensure that any necessary processing is completed and the variable FBUF$ will be available for other purposes if required. The complete program is now given.

```
 5 dim DAT$(10),FBUF$(128)
10 create "FNAME",FVAR,FBUF$,10,100
15 DAT$="ABCDEFGHIJ"
20 for RECNUM=1 to 90
30   put FVAR, RECNUM : DAT$
40 next RECNUM
50 close FVAR
```

Having established a file it is useful to be able to check that it does exist and is correct. We use another program to open and read (get) records from the file.

The information needed to open a file is given.

1. The filename
2. A file variable
3. A file buffer
4. The record length

This information is used in an open construct.

```
 5 dim DAT$(100,10), FBUF$(128)
10 open "FNAME", FVAR, FBUF$, 10
```

The next requirement is to get records from the file.

```
30 get FVAR, RECNUM: DAT$(RECNUM)
```

File variable (FVAR)
Record number (RECNUM)
Variable receiving data (DAT$(RECNUM))

Finally the file is closed and the data is displayed on the screen or printer.

```
 5 dim DAT$(100,10),FBUF$(128)
10 open "FNAME",FVAR,FBUF$,10
20 for RECNUM=1 to 90
30   get FVAR,RECNUM : DAT$(RECNUM)
40 next RECNUM
50 close FVAR
60 for R=1 to 90
70   print DAT$(R); " " ;
80 next R
```

A more complex record

Suppose we wish to establish a set of records consisting of student names, year, course number.

For example :

Name	Forenames	Year	Course
COUSINS	JANET	2	F5/024
(20 ch)	(20 ch)	(1 ch)	(6 ch)

The total record length is 20 + 20 + 1 + 6 = 47 bytes.

/....

N.B. It is the programmers responsibility to ensure that data in the record is properly matched to any variables used.

```
 5 dim FBUF$(128),NAME$(20),FORENAME$(20),Y$(1),C$(5)
10 create "FNAME",FVAR,FBUF$,47,2
20 for R=1 to 2
25   read NAME$,FORENAME$,Y$,C$
30   put FVAR,R : NAME$,FORENAME$,Y$,C$
40 next R
50 close FVAR
60 data "COUSINS", "JANET", "2", "F5/021"
70 data "SMITH", "JIM", "2", "F5/021"
```

The file can be read and the two records printed.

```
 5 dim FBUF$(128),NAME$(20),FORENAME$(20),Y$(1),C$(6)
10 open "FNAME",FVAR,FBUF$,47
20 for REC=1 to 2
30   get FVAR,REC : NAME$,FORENAME$,Y$,C$
35   print NAME$,FORENAME$,Y$,C$
40 next REC
50 close FVAR

COUSINS           JANET           2           F5/021
SMITH             JIM             2           F5/021
```

Mixing of String and Numeric Data

Numeric data e.g. 27 may be treated as string data in the form "27". It would need to be converted to numeric data for any arithmetic processing. Alternatively a second file can be created for the numeric parts. We will call it NUMFIL, its buffer will be NUMBUF$, and its file variable will be NUMVAR it will contain the records :

```
13, 15, 12, 14
 9, 14, 12, 13
```

```
 5 dim NUMBUF$(128)
10 create "NUMFIL",NUMVAR,NUMBUF$,16,2
20 for R=1 to 2
25   read M1,M2,M3,M4
30   put NUMVAR,R : M1,M2,M3,M4
40 next R
50 close NUMVAR
60 data 13,14,12,14
70 data 9,14,12,13
```

We can now write a program which will read data from both files and produce a complete external record.

```
 5 dim FBUF$(128),NAME$(20),FORENAME$(20),Y$(1),C$(6),NUMBUF$(128)
10 open "FNAME",FVAR,FBUF$,47
15 open "NUMFIL",NUMVAR,NUMBUF$,16
20 for R=1 to 2
30   get FVAR,R : NAME$,FORENAME$,Y$,C$
32   get NUMVAR,R : M1,M2,M3,M4
35   print NAME$,FORENAME$,Y$,C$, "  ";M1;M2;M3;M4
40 next R
50 close FVAR
60 close NUMVAR

COUSINS           JANET           2           F5/021  13   14   12   14
SMITH             JIM             2           F5/021  9    14   12   13
```


/.......

Up to 90 files may be stored on one Piccolo diskette and the complexity of file handling operations may be developed very substantially by the programmer.

Data Control

As mentioned in the previous chapter data must be checked at all stages to ensure that it is correct. A data file should be constructed in such a way that it can be tested for errors in the ways already applied to data statement files. This generally means having extra characters in a record and extra dummy records purely for test purposes.

The file variable also provides information about any errors in connection with file operations. So far we have used the file variable to identify files for reading (getting), writing (putting) and closing but it can also be used for error detection.

After an operation the file variable will take values as follows,

Value	indication
0	correct operation
1	diskette not inserted
2	no room in catalogue
3	too many files open
4,5	diskette is write-protected
6	reading/writing outside file area
7	file already exists
8	file does not exist
9	no room on diskette
> 10	read/write diskette error

This introductory booklet is not the place for a full discussion of files, their design and use but with care and practice modest forms of data processing can be successfully accomplished with a Piccolo. In this area of work the Piccolo's large memory, wide screen and disc system, together with the greater safety and readability of COMAL programs, are considerable assets.

Chapter nine

DIRECT ACCESS FILES

"If circumstances lead me, I will find
Where truth is hid, though it were hid indeed
Within the centre."

Polonious to Hamlet.

A serial file, like pop music on a cassette is fine for many purposes as long as things can be done in sequence, perhaps with some aids like fast forward or backward winding. By contrast, track number 5, say, of a long playing record can be played by moving the needle to it directly. However, though the L.P. may be used as a direct-access system it can also be played sequentially if the listener wishes. Piccolo disc files are similar.

In the previous chapter files were treated in a sequential way because the concept of a sequential file is widely used. However, each record in the disc-files are numbered and may be accessed directly by simply quoting the record number. In the case of medium-sized or large files this can make a considerable difference to the speed of operations. It is also generally more efficient because it reduces wear on the discs and disc drives.

Since all Piccolo disc-files are direct access files there are no new techniques to learn for creating reading or writing to files. What is more interesting are the ways in which direct access files may be used.

Records and keys

We may wish to set up a file of names, addresses, year number, course number etc. and access it according to a given name. For example, suppose we have 1000 records and wish to find the one belonging to

FOOT Michael

This is the key and in some more sophisticated languages such as COBOL it would be possible to find the record by simply quoting this key. In COMAL however we have to devise a way of finding the right record fairly quickly. Perhaps the most obvious way would be to create a separate file of all the names and their associated record numbers. This could be used first, reading it into a large array and then doing, say, a binary search. This would imply about 1000 x 20 bytes, or 20K of data. This could work perfectly well but if the file of keys became too large it could be split into say AKEY, BKEY.....FKEY.....ZKEY - 26 files, each probably less than 2K bytes and fairly easily constructed from the main file.

Simulated Case Study

We will simulate some of the processes of direct access files by the following procedure. It would be unrealistic to use only small sets of test data because the problems of handling small amounts of data are different. Even this modest-scale simulation will avoid some problems associated with handling real data but it will be a reasonable introduction.

1. Generate 100 random four letter words. These will be the keys.

2. Generate 100 random six letter words. These, together with the keys, will constitute 100 10-letter "records".

3. Sort the records into alphabetic order of their keys and place in a file.

4. Create a second file of keys only with record numbers.

5. Write a program which will, for a given "key"

   (a) Search the key file and obtain the record number .

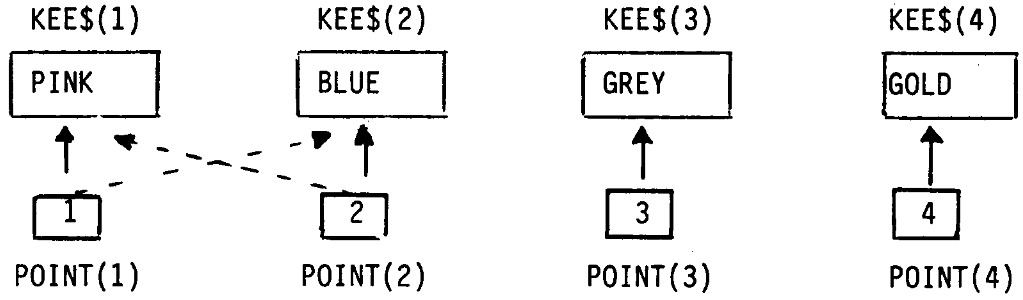
   (b) Get the required record.

Generation of Records

KEE$(100,4) will hold the keys ready for sorting. A simple bubble sort will be used but it will be adapted to make it faster. The sort program of chapter six will be altered in three ways to make it more efficient but it will still be one of the slowest computer sort methods.

(1) Each run length will decrease by one
(2) A flag will indicate if no exchanges have occurred and the sort will be complete.
(3) A set of "pointers" (numbers) will be sorted so that the actual data is not moved.
(4) We will print the value of run each time because a sort process is slow and it is useful to see what is happening.

The idea of sorting pointers can be confusing. But consider a simple case of four items.



We say :

```
        if KEE$(POINT(1))> KEE$(POINT(2)) then
          (exchange contents of POINT(1) and POINT(2))
so that POINT(1) points at BLUE
    and POINT(2) points at PINK.
```

File Creation and Entry of Records

We require a file of 100 records each with 10 bytes. The key will be placed in the first four bytes and the rest of the record in the last six. Thus we require to generate six letter random words and associate them with the four letter words already generated.

The file will be created as follows,

1. Create file, 100 records of 10 bytes.
2. Generate six letter word.
3. Place it with a four letter word on a record in alphabetical order of the four letter word.
4. Repeat (2) and (3) 100 times and close file.

Once this file is established it will be a simple matter to use it to create a second file of keys and record numbers. This will be used to access the first file. The point is that however large the records may be we can still access them directly fairly quickly.

Sections of the program are given below,

Main Program

```
 10 randomize
 20 dim KEE$(100,4),POINT(100),WORD$(6)
 30 dim REC$(10),DBUF$(128)
 40 exec FILKEES
 50 exec SORT
 60 exec PRINTIT
 70 exec FILEM
```

Fill array with 100 four letter keys

```
 80 proc FILKEES
 90   for COUNT=1 to 100
100     WORD$=""
110     POINT(COUNT)=COUNT
120     for CHAR=1 to 4
130       WORD$=WORD$+chr(int(rnd(1)*26)+65)
140     next CHAR
150     KEE$(COUNT)=WORD$
160   next COUNT
170 endproc
```

Sort pointers for alphabetical order

```
180 proc SORT
190   for RUN=1 to 99
200     FLAG=0
210     for PAIR=1 to 100-RUN
220       if KEE$(POINT(PAIR))> KEE$(POINT(PAIR+1)) then
230         TEMP=POINT(PAIR)
240         POINT(PAIR)=POINT(PAIR+1)
250         POINT(PAIR+1)=TEMP
260         FLAG=1
270       endif
280     next PAIR
290     if FLAG=0 then exit
300     print RUN;
310   next RUN
320 endproc
```

Print keys in order

```
330 proc PRINTIT
340   for K=1 to 100
350     print KEE$(POINT(K)); "  ";
360   next K
370 endproc
```

Create and enter 100 words in file "DIRECT"

```
380 proc FILEM
390   create "DIRECT",DVAR,DBUF$,10,100
400   for R=1 to 100
410     WORD$=""
420     for CHA=1 to 6
430       WORD$=WORD$+chr(int(rnd(1)*26+65))
440     next CHA
450     REC$=KEE$(POINT(R))+WORD$
460     put DVAR,R : REC$
470   next R
480   close DVAR
490 endproc
```

Sample of Output from Main Program

The sample shows that there were 84 passes in the SORT operation and some of the four letter keys.

```
    1     2     3     4     5     6     7     8     9    10    11.....
   23    24    25    26    27    28    29    30    31   ...........
   43    44    45    46    47    48    49    50    51    ...........
   63    64    65    66    67    68    69    70    71     ..........
   83    84  AHOV   AIBQ  AULE  BGGC  BOPK         ...................
 EVZK  FCGL  FEUR  FPHH  GAAR  GEIY                ...................
 HXOE  IALC  IBNR  IFUN  ITLX  IZJW                ...................
 KZJR  LBDQ  LDFG  LEVJ  LGGQ  LJPL                ...................
 NMZJ  NQRH  NSGV  NSON  OAKY  OBBC                ...................
 QOFC  QWHD  QXPQ  QZBE  RAKZ  RBEE                ...................
 TQMV  TSWQ  TUHC  TWSD  UFAI  USHO                ...................
 WRLJ  XALR  XHAJ  XZOK  YEHO  YMLQ                ....................
```

Creation of Key File

1. The file "KEYFIL" is created and the file "DIRECT" is opened.
2. The Records from DIRECT are placed in an array and the keys are extracted.
3. The four letter keys are placed in the file "KEYFIL"

```
 10 dim KBUF$(128),DBUF$(128),REC$(100,10),WORD$(4),ITEM$(10)
 20 create "KEYFIL",KVAR,KBUF$,4,100
 30 open "DIRECT",DVAR,DBUF$,10
 40 for RECNUM=1 to 100
 50   get DVAR,RECNUM : REC$(RECNUM)
 55 next RECNUM
 60 for RECNUM=1 to 100
 70   WORD$=REC$(RECNUM,1 : 4)
 80   put KVAR,RECNUM : WORD$
 90 next RECNUM
110 close DVAR
120 close KVAR
```

Search of Main File Using Keys

1. Both files are opened.
2. The required key is entered.
3. KEYFIL is searched to get record number.
4. Record number is used to access "DIRECT".
5. Required record, if it exists, is printed.

```
 10 dim KBUF$(128),DBUF$(128),SEEK$(4),WORD$(4),ITEM$(10)
 20 open "KEYFIL",KVAR,KBUF$,4
 30 open "DIRECT",DVAR,DBUF$,10
 40 input "Enter required key.",SEEK$
 50 put KVAR,100 : SEEK$
 60 RECNUM=0
 70 repeat
 80   RECNUM=RECNUM+1
 90   get KVAR,RECNUM : WORD$
100 until WORD$=SEEK$
110 if RECNUM=100 then
120   print "Not found."
130 else
140   get DVAR,RECNUM : ITEM$
150   print RECNUM,ITEM$
160 endif
170 close DVAR
180 close KVAR

 71       RFWYPICTOU
```

Procedure for illustration of D.A. Files

(1) Run DIRFIL

Generate 100 ten letter words and file in "DIRECT". The words will be sorted in alphabetic order of their keys (first four letters). The sort keys are printed.

(2) Run SETKEYF to set up separate file of 4 letter keys only.

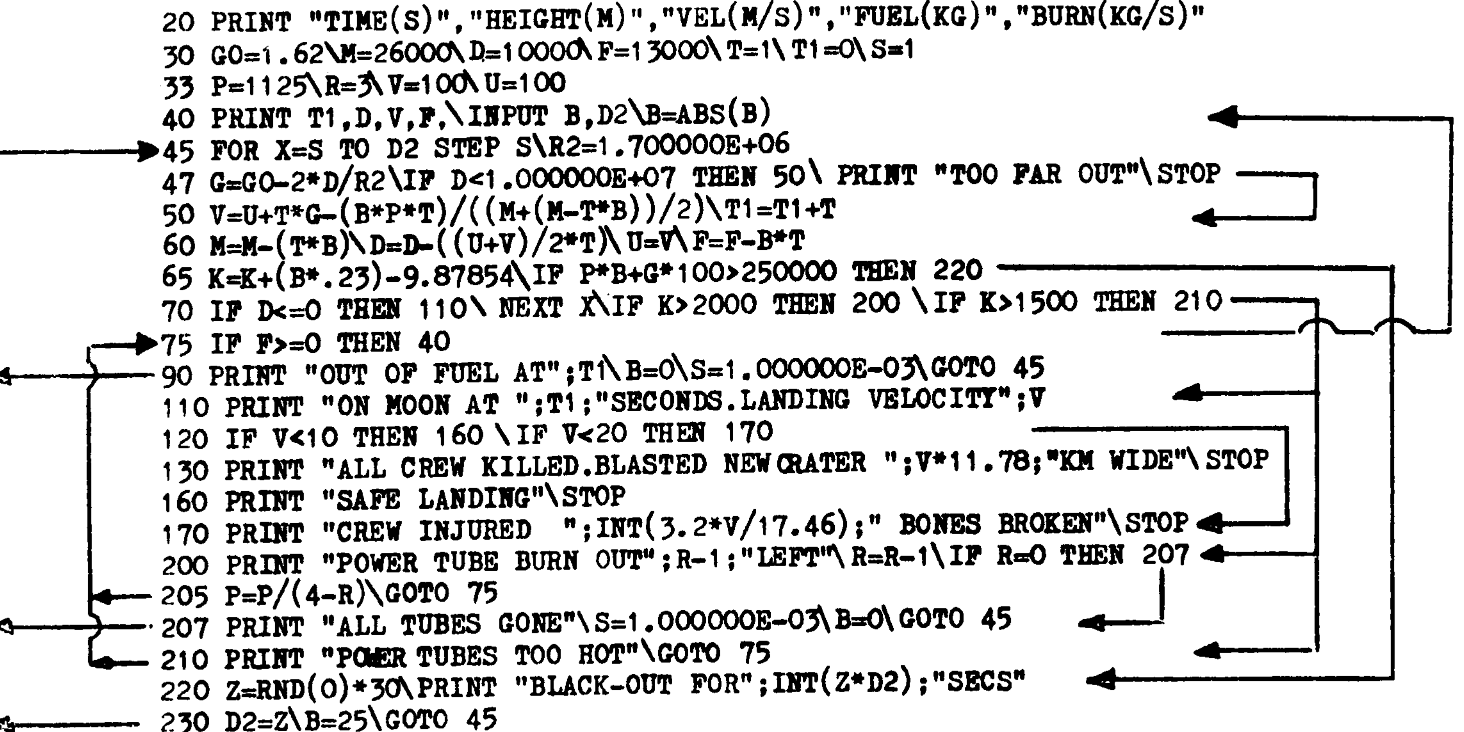
(3) Run SEARCH and use one of the keywords to find corresponding record. The record with record number will be printed if it exists.

(4) A simple program to read and display the file "DIRECT" was also used to check that it was correctly stored on the disc.

The system is a simple outline. Other techniques for access and sorting could make the operation considerably faster. Attention would also need to be paid to error avoidance and testing.

APPENDIX A1.

MOON LANDING PROGRAM



```
20 PRINT "TIME(S)","HEIGHT(M)","VEL(M/S)","FUEL(KG)","BURN(KG/S)"
30 GO=1.62\M=26000\D=10000\F=13000\T=1\T1=0\S=1
33 P=1125\R=3\V=100\U=100
40 PRINT T1,D,V,F,\INPUT B,D2\B=ABS(B)
45 FOR X=S TO D2 STEP S\R2=1.700000E+06
47 G=GO-2*D/R2\IF D<1.000000E+07 THEN 50\ PRINT "TOO FAR OUT"\STOP
50 V=U+T*G-(B*P*T)/((M+(M-T*B))/2)\T1=T1+T
60 M=M-(T*B)\D=D-((U+V)/2*T)\U=V\F=F-B*T
65 K=K+(B*.23)-9.87854\IF P*B+G*100>250000 THEN 220
70 IF D<=0 THEN 110\ NEXT X\IF K>2000 THEN 200 \IF K>1500 THEN 210
75 IF F>=0 THEN 40
90 PRINT "OUT OF FUEL AT";T1\B=0\S=1.000000E-03\GOTO 45
110 PRINT "ON MOON AT ";T1;"SECONDS.LANDING VELOCITY";V
120 IF V<10 THEN 160 \IF V<20 THEN 170
130 PRINT "ALL CREW KILLED.BLASTED NEW CRATER ";V*11.78;"KM WIDE"\STOP
160 PRINT "SAFE LANDING"\STOP
170 PRINT "CREW INJURED  ";INT(3.2*V/17.46);" BONES BROKEN"\STOP
200 PRINT "POWER TUBE BURN OUT";R-1;"LEFT"\R=R-1\IF R=0 THEN 207
205 P=P/(4-R)\GOTO 75
207 PRINT "ALL TUBES GONE"\S=1.000000E-03\B=0\GOTO 45
210 PRINT "POWER TUBES TOO HOT"\GOTO 75
220 Z=RND(0)*30\PRINT "BLACK-OUT FOR";INT(Z*D2);"SECS"
230 D2=Z\B=25\GOTO 45
```

The above program was written by a teacher who does not understand why some people object to the liberal use of GOTO statements.

Try to follow the program. The flow lines on the right are conditional jumps and those on the left simple jumps.

Do you think the program has any discernable structure ? Does it have what Dijkstra has called "Spaghetti-like control paths" ?

How many times do you think RUN was typed in writing and debugging the program ? Do you think the exercise had any educational value ?

APPENDIX A2.

PICCOLO STARTING PROCEDURE

1. Connections

   (a) keyboard to port J4. Screw in screws.

   (b) video to IN socket. Switch to 75Ω unless a slave video is used.

   (c) video power connection. Screw in.

   (d) Computer power.

2. Switch on computer and video.

3. Place system in drive - hold disc by label edge face upwards.

4. Close disc drive shutter.

5. Press re-set switch.

   The computer should load and start COMAL. An asterisk will indicate this. The LOAD or LOOKUP commands can now be used.

   EXIT from COMAL with BYE then use :

   CATALOG to format a disc.

   SYSTEM to save COMAL on disc.

   COMAL to return to COMAL.

Switching off

1. Remove disc and place in envelope.

2. Switch off computer and terminal if further use is not required for 15 minutes.

   It is not advisable to move the video monitor until 15 minutes after switching off.

APPENDIX A3

CHARACTER CODES

The characters generated by the COMAL function CHR(N) are given below.
N is a denary number.

N	CHR(N)	N	CHR(N)	N	CHR(N)	N	CHR(N)
0		32	SPACE	64	@	96	`
1		33	!	65	A	97	a
2		34	"	66	B	98	b
3		35	£	67	C	99	c
4		36	$	68	D	100	d
5	⇤	37	%	69	E	101	e
6	XY CUR	38	&	70	F	102	f
7	BLEEP	39	'	71	G	103	g
8	CUR L1	40	(	72	H	104	h
9	CUR R4	41	)	73	I	105	i
10	CUR D1	42	*	74	J	106	j
11		43	+	75	K	107	k
12	CL SCR	44	,	76	L	108	l
13	CUR ST LN	45	-	77	M	109	m
14	PRINTER SG	46	.	78	N	110	n
15	PRINTER N	47	/	79	O	111	o
16		48	0	80	P	112	p
17		49	1	81	Q	113	q
18		50	2	82	R	114	r
19		51	3	83	S	115	s
20		52	4	84	T	116	t
21		53	5	85	U	117	u
22		54	6	86	V	118	v
23		55	7	87	W	119	w
24	CUR R1	56	8	88	X	120	x
25		57	9	89	Y	121	y
26	CUR U1	58	:	90	Z	122	z
27		59	;	91	[	123	{
28		60	<	92	\	124	¦
29	CUR HOME	61	=	93	]	125	}
30	DEL EO LN	62	>	94	↑	126	~
31	DEL EO SCR	63	?	95	—	127	■

APPENDIX A4.

Graphics

For a full description of graphics the manuals should be consulted. The following procedures will be useful for certain types of application. Each identifier contains a "Z" so that possible conflict with user's identifiers is avoided.

1. CURSOR (ACROSS,DOWN) positions cursor
2. PLOT (ACROSS, DOWN) positions cursor and plots a small square on a 160 x 72 grid.
3. PUTPIC places all screen data on a disc file.
4. GETPIC retrieves screen data from a disc file.
5. DRAW draws a picture fast using the screen data.
6. PRINTIT outputs screen data to printer.
7. SCREEN incorporates all the above procedures.

To use any procedure it should be loaded from the disc. The user may then write a program which calls the procedure. In the case of PLOT the user should replace the sample routine with his own.

All the procedures use the identifiers as indicated. Clearly the user should not use identifiers which conflict with any of those used in a procedure incorporated in his program.

Procedure	Identifier	Description
CURSOR	AZ	across position
	DZ	down position
PLOT	the identifiers of CURSOR plus :	
	SPZ$(80)	80 spaces
	LINEZ$(24,8)	Screen data
	LZ	Working variable
	BLANKZ	Procedure to clear screen LINE$
	ACZ, DNZ	Formal parameters of PLOT
	ACPOZ, DNPOZ	Positions in 160 x 72 array
	BITZ	Bit in semi-graphics character
	CODEZ	Semi-graphics character code
PUTPIC	FNAMZ$(10)	File name
	FVARZ	File variable
	FBUFZ$(128)	File buffer
	LZ	Working variable
	LINEZ$(24,80)	Screen data
GETPIC	same as PUTPIC	
DRAW	the identifiers of CURSOR plus :	
	LZ	Working variable
	LINEZ$(24,80)	Screen data
PRINTIT	LZ	Working variable
	LINEZ$(24,80)	Screen data
SCREEN	All the above procedures and identifiers	