Contents

Preface

Ten to fifteen years ago machines with the memory size and processing capability of the BBC micro would have cost many thousands of pounds and were the exclusive domain of computer professionals. Nowadays powerful computers are in the hands of the home user and this book aims to bring the tools of the trade of the computer scientist to the micro user.

The book is a practical introduction to advanced topics in computer science. Rather than adopt the formal approach found in most computer science texts, we have introduced each technique practically, by using simple program modules that act as building blocks. Each topic is covered at sufficient depth so that for a non-professional the waters are neither fathomless nor so shallow as to be trivial.

The techniques selected for inclusion in this text form the foundation stone of advanced computer graphics, artificial intelligence, automatic musical composition, databases, arcade game programming, board game programming, adventure game programming, computer assisted learning, computer aided design and language processing.

With the exception of Chapter 3 the book contains no difficult mathematics and perseverance with the material will compensate for a lack of knowledge of the higher echelons of mathematics. Even the mathematics in Chapter 3 can be ignored with impunity and a sound understanding of the material derived from using the procedures.

The text is supported by a considerable number of program fragments, procedures and complete programs together with suggestions on projects that you can undertake yourself.

An essential prerequisite is a knowledge of BASIC, either from our companion volume, 'The BBC Micro, BASIC, Sound and Graphics', or front experience of other BASIC dialects. If your experience is on another machine, you will need access to a BBC Micro 'User Guide'. For the sake of completeness some material from our companion volume is repeated in Chapter 2.

Structured programming techniques are used throughout the text and we have attempted to make the programs readable. The programming style adopted is described in Chapter 1 and it makes extensive use of the BBC BASIC control structures and procedure facilities. Apart from a single unavoidable occurrence there is no use of GOTO or GOSUB anywhere in the text.

A mastery of the matorial In this book will make you an expert micro-programmer. If you can creatively expand and develop the ideas herein then ring ACORNSOFT and ask for a job.

How to use this book

This book need not be read sequentially, you can if you prefer dip into the topics in any order.

Some chapters are self contained and others can be read only after earlier material has been understood. The following table is a guide to how the book can be used.

Chapter no.	Prerequisite
1	Some knowledge of standard BASIC
2	A knowledge of the basic graphics facilities of the BBC Micro. (See our companion volume)
3	As for chapter 2
4	Chapter 2
5	A knowledge of the basic sound facilities of the BBC micro. (See our companion volume)
6	Chapter 1
7	Chapter 1
8	Chapters 1 and 7
9	Chapters 1, 7 and 8
10	Chapters 1 and 7

Are you a slow typist?

You can experiment with the programs described in this book without having to type them. All the programs of any length are available on a computer cassette.

Some of the larger programs are useful utilities in their own right and others provide the foundations for fairly elaborate programming projects in sound, graphics, animation and games.

Programs are also available on cassette for the companion volume by the same authors:

The BBC Micro Book: BASIC, Sound and Graphics

Books and cassettes are available from your bookseller or direct from

Addison-Wesley Publishers Ltd.,
53 Bedford Square,
London, WC1B 3DZ.

Chapter 1 Programming style for BBC BASIC

Apart from the powerful graphics and sound facilities, BBC BASIC provides a number of 'control statements' that are not usually available in other dialects of BASIC. A control statement is a statement that is used to control the order in which a program is obeyed and examples of the control statements that are available in standard BASIC are FOR statements, IF-GOTO statements, GOTO statements and GOSUB statements.

It has long been recognised by computer scientists that programs written using combinations of the above statements tend to be difficult to write, difficult to read and difficult to debug. This is because of heavy reliance on the use of GOTO and GOSUB statements. Excessive use of these statements results in programs whose possible execution pathways are so intertwined that the control structure of a program is obscured.

The number of control structures that are needed to cover the majority of programming situations is very small and BBC BASIC provides control statements for implementing most of the commnon constructions without using GOTOs or GOSUBS. If you are used to programming in 'standard' BASIC it needs a certain amount of self-discipline to learn to use these new statements and abandon old programming habits, but the effort is well worthwhile. The result will be more readable programs. The programmer will also have a much clearer idea of the structure of his programs and will find them easier to debug and alter. Programs are not less efficient as some of the 'GOTO diehards' would have us believe. In many cases, the use of the appropriate control statements instead of a messy combination of GOTO statements results in a more efficient program.

1.1 Control statements in BBC BASIC

In this section, we briefly introduce and illustrate the use of the novel control statements in BBC BASIC, and, in subsequent sections, we discuss ways of using these statements to improve our programming style. Note that there are some dialects of BASIC that provide some, but not all, of the facilities described in this chapter.

Loops

As well as the FOR-NEXT construction available in standard BASIC, BBC BASIC provides a REPEAT-UNTIL construction for use in implementing a so-called 'non-deterministic' loop. A non-deterniiinistic or conditional loop is a loop where the computer cannot calculate in advance how many times to obey the loop and a section of program is to be obeyed repeatedly until some condition has been satisfied. For example, a simple ccnnputer-assisted learning program might repeatedly set multiplication questions and input the answers from the keyboard until one of the questions is answered wrongly:

```
10  questions = 0
20  REPEAT
30    questions = questions+1
40    a = RND(11) + 1 : b = RND(11) + 1
50    PRINT a; "x"; b; "=";
60    INPUT answer
70  UNTIL answer <> a*b
80  PRINT "Wrong! "
90  PRINT "You got "; questions-1; " questions right"
```

The REPEAT statement introduces a section of program that is to be obeyed repeatedly and the UNTIL statement indicates the extent of the loop, and specifies the condition for stopping the repetition. The REPEAT statetment and an equivalent construction using a GOTO statement are shown below:

REPEAT loop	Equivalent GOTO loop
20 REPEAT	
.	30 .
.	.
.	.
70 UNTIL condition	70 IF NOT condition GOTO 30

IF statements

The IF statement in BBC BASIC can have the form of the 'standard' logical IF:

```
IF condition THEN one or more statements
```

The statements after THEN are obeyed only if the condition is TRUE. The other form of IF statement is

```
IF condition  THEN   one or more statesnents
              ELSE   one or more statements
```

In this ense, if the condition is TRUE, the statements after THEN are obeyed, otherwise the statements after ELSE are obeyed. A simple example of a program involving an IF-THEN statement is:

```
10  INPUT bankbalance, withdrawal
20  bankbalance = bankbalance - withdrawal
30  IF bankba1ance<0 THEN
      bankbalance=bankbalance-0.20 : PRINT "Overdrawn!":
      PRINT "Send this customer a letter from manager"
40  PRINT "Balance is now "; bankbalance
```

Note that an IF statement constitutes a single numbered line of a BBC BASIC program. (A single numbered line can occupy up to 240 characters and may occupy several screen lines.) You must type the complete IF statement without pressing RETURN. The RETURN key is pressed only when a numbered line is complete. If an IF statesnent does not fit into 240 characters, then it is almost certain that your program would be better structured using procedures (see below).

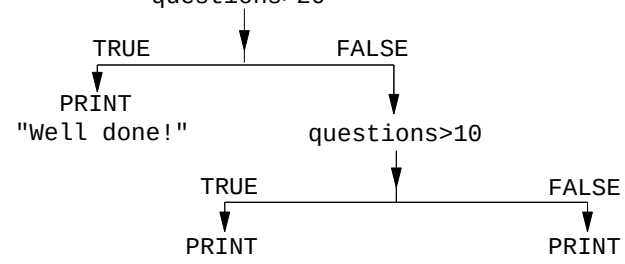
When more than one statement is typed after THEN, the statements must be separated from each other by colons and these statements are either all obeyed or all ignored. The name rule applies to multiple statenents following the ELSE.

To illustrate the use of an IF-THEN-ELSE statement, we could extend our 'multiplication program':

```
100  IF questions>20 THEN
       PRINT "Well done! That was very gwd. "
     ELSE PRINT "You must brush up on your tables."
```

or even

```
100  IF questions>20 THEN
       PRINT "Well done!"
     ELSE IF questions>10 THEN
          PRINT "Room for improvement"
        ELSE PRINT "Learn your tables!"
```



In the second case, the statement after the ELSE is a further IF statement which will be obyed only if the condition 'questions>20' was FALSE. Only one of the three PRINT statenents will be selected and obeyed. The program is selecting one out of three alternative courses of action as illustrated in the tree diagram.

We can compare an IF-THEN statement with an equivalent GOTO construction:

```
10 IF condition THEN 10 IF NOT condition GOTO 20

   statements . statements (numbered)
   .
20 carry on 20
```

Here is an IF-THEN-ELSE statement together with an equivalent GOTO construction:

```
10 IF condition THEN 10 IF NOT condition GOTO 16
   .
   statements . statements (numbered)
   .
   ELSE 15 GOTO 20
   .
   statements . statements (numbered)
   .
20 carry on 20 carry on
```

In one of the examples above, we 'nested' one IF statement inside another. Unfortunately, the extent to which we can nest IF-statements in BBC BASIC is limited in several ways. For example, unexpected effects can be obtained if we use an IF-THEN-ELSE after the THEN of another IF statement. (You will find that the computer can not decide to which IF the ELSE belongs.) We are also limited by the restriction that our complete nested IF statement must fit into 240 characters (6 lines inM0DE 7). We suggest that the use of nested IF statements is limited to the use of IF after ELSE as illustrated above.

The standard way of implenenting more complex IF structures in BASIC is to use the GOTO statement. However, the use of procedures described below will enable us to program complex nested control structures without resorting to the use of GOTO.

Simple procedures

A procedure in BBC BASIC provides a facility for giving a name to a section of program. The programmer can then write the name of the procedure wherever he wants that section of program to be obeyed. This has two main advantages:

Firstly, if the named operation has to be carried out at several different places in a large program we avoid writing out the same section of program in full at each place.

Secondly, and just as important, careful use of procedures can make a large program easier to write and simpler for other people to read.

The first advantage can, of course, be obtained in standard BASIC by using GOSUB statements and the second advantage can be obtained, to a certain extent, by careful annotation of standard BASIC subroutines with REM statements. However, the use of named procedures makes it easier to obtain these advantages and encourages the writing of more readable programs. Here is a short BBC BASIC program that involves a procedure:

```
 10  PRINT "Type first 10 numbers"
 20  PROCaddtenumbers
 30  PRINT "Type next 10 numbers"
 40  PROCaddtennumbers
 50  END

100  DEF PROCaddtennumbers
110  LOCAL i, next, total
120    total = 0
130    FOR i = 1 TO 10
140      INPUT next
150      total = total + next
160    NEXT i
170    PRINT "Total = "; total
180  ENDPROC
```

The section of program from line 100 onwards constitutes a procedure definition and the procedure is referred to or 'called' at line 20, and again at line 40, by writing the name of the procedure. Calling a procedure in this way tells the computer to go and obey the procedure definition and come back when it encounters an ENDPROC statement.

In the program above, we have specified that the variables 'i', 'next', and 'total' are 'local' to the procedure. These variables are available for use only while PROCaddtennumbers is being obeyed. Variables declared at the start of a procedure in this way can <u>not</u> be used after ENDPROC has been obeyed. It is recommended that any variable which is used only within a particular procedure should be declared locally to that procedure. The computer will then

ensure that the programmer does not accidentally use the same variable for conflicting purposes in different parts of a large program. A variable with the same name can be used elsewhere in the program and its value will automatically be held in a different storage location, thus eliminating any possibility of confusion. The same program could have been written in standard BASIC using GOSUB statements:

```
 10  PRINT "Type first 10 umbers"
 20  GOSUB 120
 30  PRINT "Type next 10 numtx;rs"
 40  GOSUB 120
 50  END
120  T = 0
130  FOR I = 1 TO 10
140    INPUT N
150    T = T + N
160  NEXT i
170  PRINT "Total = "; T
180  RETURN
```

In the above BBC BASIC program, a section of program given a name and this name was used (twice) to tell the computer to obey that section of program. As we shall discuss later, it is good programming practice to give a name to any logically separate section of program, even if that section of program is obeyed only once.

Procedures with parameters

A simple procedure can be used to enable a program to carry out the same operation at different parts in a program. A common requirement is for similar, but not necessarily identical, operations to be carried out at different points in a program.

As a somewhat contrived example, the procedure of the last section could have been given a 'parameter' indicating how many numbers were to be added up.

```
 10  PRINT "Type 5 umbers"
 20  PROCaddnumbers(5)
 30  PRINT "Now type 10 numbers"
 40  PROCaddnumbers(10)
 50  END
```

The parameter in brackets after the name of the procedure is a piece of information that is to be tranenitted to the proocedure that is being called. In this case, the intention is that the number in brackets tells the procedure how many

values to add up. The first time the procedure is called it is to add up 5 numbers and the second time it is to dd up 10 numbers.

The procedure must now be defined in terms of a named variable that will be given a value each time the procedure is called.

```
100  DEF PROCaddnumbers(howmany)
110  LOCAL i, next, total
120    total = 0
130    FOR i = 1 TO howmany
140      INPUT next
150      total = total + next
160    NEXT i
170    PRINT "Total = "; total
180  ENDPROC
```

When this procedure is called line line 20, the procedure definition is obeyed with:

```
howmany = 5
```

and when it is called from line 40, the procedure definition is obeyed with:

```
hownany = 10
```

As another example, here is a program that is given a sum of money and which works out how many coins of each available denomination are required to make up that sum of money.

```
 10  INPUT "Change", change
 20  PROChownany(50) : PROChowmany(20)
 30  PROChowmany(10) : PR(Ehownany( 5)
 40  PROChomnany( 2) : PROChowmany( 1)
 50  END

100  DEF PROChowmany(denomination)
110  LOCAL noofcoins
120    noofcoins = change DIV denomination
130    change = change MOD denomination
140    PRINT "No of "; denomnination; "s "; noofcoins
150  ENDPROC
```

The program first works how many 50p pieces can be fitted into the given sum and works out how much change is left then that has been done. It then does the same, with the remaining change, for 20p pieces, and so on. A procedure is

used to work out how many coins of a given denomination fit into the change that is currently loft. PROChowmany is the name of a procedure that is used six times. Each time the procedure is called, it is supplied with a parameter in brackets telling it which denomination of coin to deal with next. The operators DIV and MOD are useful in this context. DIV gives the result of dividing two integers or whole numbers, ignoring any remainder. MOD gives the remainder obtained on dividing two integers.

A procedure can have a parameter that is a string and it can also have more than one parameter. These features will be illustrated as and when we need them.

In other programming languages, it is usually possible to pass information out of a procedure by changing the value of one of its parameters while the procedure is being obeyed. In BBC BASIC, parameters can only be used for passing information into a procedure and these parameters are sometimes known as input parameters. If information calculated in a procedure is to be used outside that procedure, the information must be placed in a global variable - any variable that is not a parameter or a local variable. For example, in the program at the beginning of this section the global variable 'change' is altered by each call of the procedure. In fact this global variable is used to transfer information both into and out of the procedure.

Functions

If the result of some process is a single value then a function is sometimes an elegant alternative to a procedure.

First let us look at the ways in which a function differs from a procedure. Certainly, they are both separate modules of program text referred to by name, but they differ in the way in which they are called. Functions are called by using them in expressions - that is the first difference. The second difference is that the result of obeying the function is a single value which replaces the function call in the originating expression. Let us illustrate this by considering the use of one of the standard functions:

```
y = x + SQR(2)
```

When this statement is being obeyed, the computer obeys the definition of the function SQR, and a number - the result of obeying the function - replaces the subexpression SQR(2). In the case of a standard function like SQR, the definition of the function is already stored as part of the BASIC system, but it is also possible for the programmer to define his own functions. In BBC BASIC, the programmer defines a function in a way that is very similar to the way in which a procedure is defined. A function defined in this way can be used in exactly the same way as the standard functions.

This program reads 3 pairs of numbers and adds the larger

of the first pair, the larger of the second pair and the larger of the third pair. A function is used to find the larger of two numbers.

```
10  INPUT a,b, p,q, x,y
20  PRINT FNmax(a,b) + FNmax(p,q) + FNmax(x,y)
30  END

40  DEF FNmax(first,second)
50    IF first > second THEN  = first
      ELSE  = second
```

The effect of calling a function is the calculation of a single result. Since calling a function produces a single result, we must indicate, somewhere in the function definition, what this result is to be. Instead of ENDPROC, the function terminates when a statement of the form:

```
= expression
```

is obeyed. The value of this expression is returned as the value of the sub-expression used to call the function.

When the above program is obeyed, evaluation of the sub-expression 'FNmax(a,b)' causes the function definition to be obeyed with 'first' set to the value of 'a' and 'second' set to the value of 'b'. If the function is called when we have the situation

```
a = 4.79
b = 5.64
```

then the function definition is obeyed with

```
first = 4.79
second = 5.64
```

and the statement

```
= second
```

is obeyed as a result of obeying the IF-statement. The value of the sub-expression 'FNmax(a,b)' will therefore be 5.64 and this is the value which will be used in subsequent evaluation of the larger expression:

```
FNmax(a,b) + FNmax(p,q) + FNmax(x,y)
```

Apart from the need to return a particular value as its result, the definition of a function in BBC BASIC is very similar to the definition of a procedure. A function

definition can use as many statements as we like in order to calculate the value that is to be the result of the function. More complicated function definitions will beintroduced when they are required.

1.2 Stepwise refinement and program design

In the remainder of this chapter, we demonstrate the well-known programming technique called 'stepwise refinement'.

The first step in writing a complex program should be to sketch an outline of what the program is going to do without getting bogged down in the detailed BASIC instruction required. Using procedures for the logically distinct operations in a program allows us to write our initial outline in BASIC where we invent procedure names to describe operations that we have not yet programmed in detail. Only when we have a clear idea of what each named procedure is to do and how it fits into the overall program do we go on to define each procedure in detail.

In a complicated program, writing one of the procedures may itself be a difficult programming task and a procedure can itself be defined in terms of other procedures.

We shall illustrate this approach to programming by writing two moderately complicated programs.

CAL structures - a multiplication compestition

The next program is an example of a Computer Assisted Learning program. It could be used to encourage children to learn their multiplication tables by organising a multiplication competition. Once the program is running, the children will take turns to sit at the keyboard and do a tables test. The program will keep a league table of the top ten scores obtained during a run of the program and this table will be printed after each test is completed.

We can outline the process that the program will carry out :

```
10  CLS
20  PRINT "Multiplication Competition"
30  PROCinitialise
40  REPEAT
50    PROCnextcompetitor
60    PROCprinttopten
70  INPUT "Anyone else to play (Y/N)", reply$
80  UNTIL reply$="N"
90  END
```

Note that this outline does not involve any tremendously complicated control structure. It contains only a simple REPEAT loop and writing an outline like this should not present any difficult programming problems.

Now that we have the overall structure of the program clear in our minds, we can introduce a little more detail by

defining the procedures used in our outline.

Most programs require variables or arrays to be set to starting values and such 'initialisation' is best tidied away into a special procedure. In this case, PROCinitia1ise will set up a 'top scores' table to contain ten zero scores. The table will consist of two parallel arrays, one array to maintain the names of the top ten players, and the other their scores.

```
100  DEF PROCinitialise
110  LOCAL slot
120    DIM topname$(10), topscore(10)
130    FOR slot = 1 TO 10
140      topscore(slot) =0
150      topname$(slot) = "--------"
160    NEXT slot
170  ENDPROC
```

We can now concentrate on the problem of defining PROCnextcompetitor which will set a single multiplication test and handle the results. Writing this procedure can be viewed as a separate programming problem that is a little easier than the problem with which we started. We use the same approach to writing PROCnextcompetitor as we used in approaching the original problem - we write an outline description of what the procedure will do using further named procedures to describe operations that will be programmed in detail later. This process can be continued and we can have procedures within procedures within procedures etc. Very complex tasks can be implemented in this way.

```
200  DEF PROCnextcompetitor
210    INPUT "What's your name" , name$
220    PRINT "Hello, "; name$
230    PRINT : PRINT "Ready"
240    PRINT : PRINT "Go! " : PRINT
250    PROCgivetest
260    PROCupdatetable(name$, score)
270  ENDPROC
```

We have already written a short program that sets a multiplication test and we use a variation of this program as our definition of PROCgivetest. We introduce a further constraint so that a test is terminated either if a question is answered wrongly or if a time limit is exceeded. The special BBC BASIC variable TIME is automatically increased by 1 every one hundredth of a second and we use this

variable to time the test.

```
300  DEF PROCgivetest
310  LOCAL questions, a, b, answer
320    TIME = 0
330    questions = 0
340    REPEAT
350      questions = questions+1
360      a = RND(11) + 1 : b = RND(11) + 1
370      PRINT a; "x"; b; "=";
380      INPUT answer
390    UNTIL answer<>a*b OR TIME>3000
400    IF answer <>a*b THEN
          PRINT "Wrong!" : score = questions - 1
       ELSE score = questions
410  ENDPROC
```

PROCupdatetable is probably the trickiest procedure to write. We need to check first of all whether the new score should be in the top ten. If it is not, the procedure should terminate immediately. If the new score is in the top ten, then we need to find the position at which it should be inserted, move the other scores and names down to make room and insert the new score and name in the table.

```
500  DEF PROCupdatetable(n$, score)
510  LOCAL slot, position
520    IF score <= topscore(10) THEN ENDPROC

530      REM find slot for new top score
540    slot = 0
550    REPEAT
560      slot = slot + 1
570    UNTIL score>topscore(slot)

580      REM move old scores down
590    FOR position = 9 TO slot STEP -1
600      topscore(position+1) = topscore(position)
610      topname$(position+1) = topname$(position)
620    NEXT position

630    topscore(slot) = score
640    topname$(slot) = n$
650  ENDPROC
```

Finally we define the procedure for displaying the league table on the screen.

```
700  DEF PROCprinttopten
710  LOCAL p
720    CLS
730    PRINT "Last score: "; score : PRINT
740    PRINT "TOP TEN" : PRINT : PRINT
750    FOR p = 1 TO 10
760      PRINT topname$(p); TAB(20); topscore(p)
770    NEXT p
780  ENDPROC
```

Exercises

1 If you are familiar with the BBC BASIC SOUND statement, modify the multiplication contest program so that it plays a short 'tension building' tune before each test.

2 We could have applied a further stage of stepwise refinement to PROCupdatetable by defining it in terms of two further procedures, PROCfindslot and PROCinsert. Do this.

3 If a test is terminated because of a wrong answer, the message indicating that the answer was wrong does not remain on the screen long enough for it to be read. Insert a time delay at the appropriate point in the program.

4 As it stands, the program could ask the same question twice during the course of the same test. Modify the program so that it records the questions asked and ensures that the same question is not asked twice.

5 Change the program so that a test is terminated only when a time limit is exceeded. If a wrong answer is typed, the test should continue, but a message should of course be displayed. Only correct answers should be counted towards the score.

Program structure for playing a board game

In Chapters 8 and 9, we shall be looking at some of the techniques needed to write program that play 'board' games. Examples of the kind of game that we have in mind are NIM, Noughts and Crosses (or Tic-Tac-Toe), Kalah, Go-Moku, Go, Draughts (or Checkers) and Chess.

Programming techniques used for board games are completely different from those required for 'reaction' games like Space Invaders or Pacman where the machine is simply logging the user's reaction speeds and looking for coincidence of objects. In such arcade games, most of the programming effort goes into producing exotic animated displays.

Here we present the outline structure of a program that plays a board game for two players. The obvious possibility is that the computer (or, to be more precise, part of the program) will act as one player and someone seated at the keyboard will act as its opponent. However, as we shall see, this is not the only possibility and the same overall program structure will allow for other useful combinations. The essential requirement for a board game program is that it should repeatedly process one player‹s move, either its own or its opponent‹s. The most convenient way of organising this is outlined in the procedure PROCplaygame.

```
  10  PROCplaygame
  20  END
 100  DEF PROCp1aygame
 110    PROCsetupboard : REM and decide who starts.
 120    PROCdisplayboard
 130    gameover = FALSE
 140    REPEAT
 150      IF turn$ = "A" THEN PROCplayerA
          ELSE PROCplayerB
 160      PROCdisplayboard
 170      PROCtestgameover
 180    UNTIL gameover
 190    PROCannouncewinner
 200  ENDPROC
```

In order to show how versatile this structure is, we describe four situations in which it could be used:

(a) The program will act as player A, using PROCplayerA to choose its move. Someone seated at the keyboard acts as the program's opponent and PROCp1ayerB will organise the input of this player's move.

(b) Two human players seated at the keyboard can use the computer as a board and scorekeeper. PROCplayerA is used to input one player's move and PROCplayerB is used to input the other's.

(c) Two rival programmers want to ccnnpare their game programming skills. One programnger can write PROCplayerA to choose a move and the other can write PROCplayerB to choose a move his way. The computer can then be made to play through one or more games by itself in order to decide whose procedure is better.

(d) The serious student of a game such as chess may want to store records of interesting games (on cassette or disk files) and have the program play through these game

under his control so that he can analyse them. If, as is quite likely, the chess analyst wanted the program to go back to an earlier stage in the game and replay a sequence of moves or if he wanted to experiment with alternatives to the recorded moves, some slight adjustment to our outline program structure might be necessary.

We now convert our outline program into a complete program for a very simple game. The game we use is called 'Last One Wins' and the rules are:

The 'board' is a pile of counters.
Two players take turns at removing at least one and not more than three counters frcun the pile.
The player who removes the last counter is the winner.

The computer will play against an opponent. The opponent is to be allowed to decide how many counters there will be at the start of the game and who makes the first move. The following program plays this game very stupidly (by making completely random moves) but the program will serve to illustrate a number of points. This game and the program developed in this section will be extensively referred to in chapter 8.

```
 10  PROCplaygame
 20  END

100  DEF PROCplaygame
      .
      .
      .
200  ENDPROC

300  DEF PROCsetupboard
310    INPUT "How many counters", counters
320    INPUT "OK. Do you want to start", reply$
330    IF INSTR("Yy" ,LEFT$(reply$,1))
       THEN turn$ = "B" ELSE turn$ = "A"
340  ENDPROC

400  DEF PROCdisplayboard
410    PRINT
420    PRINT "There are "; counters; " counters left."
430  ENDPROC

500  DEF PROCtestgameover
510    gameover = (counters = 0)
520  ENDPROC
```

```
  600  DEF PROCannouncewinner
  610    IF turn$ = "A" THEN PRINT "You win."
         ELSE PRINT "I win."
  620  ENDPROC

  700  DEFPROCplayerA
  710    PRINT "My turn."
  720    PROCcheckmovesavailable
  730    IF movesavailable=1 THEN move=1
         ELSE move= RND(movesavailable)
  740    PRINT "I take "; move; " counters."
  750    counters = counters - move
  760    turn$ = "B"
  770  ENDPROC

  800  DEF PROCcheckmovesavailab1e
  810  IF counters<3 THEN movesavailable = counters
           ELSE movesavailable = 3
  820  ENDPROC

  900  DEF PROCplayerB
  910    PROCinputmove
  920    counters = counters-move
  930    turn$ = "A"
  940  ENDPROC

 1000  DEF PROCinputmove
 1010    PROCcheckmovesavailable
 1020    INPUT "Your turn. How many do you take", move
 1030    IF move>0 AND move<=movesavailable THEN ENDPROC
 1040    REPEAT
 1050      PRINT "At least one and not more than three."
 1060      INPUT "Try again:" move
 1070  UNTIL move >0 AND move <= movesavailable
 1080  ENDPROC
```

As the program stands, some of the procedures contain only one or two instructions and you may wonder why we bother to use so many procedures. The advantage of breaking the program up into procedures in this way is that if we want to change or improve any particular aspect of the program's behaviour, we can concentrate our attention on the procedure that controls that aspect. For example, as it stands the prograni plays rather stupidly, but all the stupidity is confined to one procedure, PR0CplayerA. If we wanted the program to play a better games we would concentrate on reprogramming this procedure. (There is in fact a very simple rule that can be used for choosing a good move in this game - think about it.) As another example of the sort of improvement that could be made, we might want to use graphics facilities to produce a pictorial

representation of the pile of counters, the display being changed after each move. The changes to enable the program to do this could be concentrated in PROCdisplayboard.

Exercises

1 Change the above program so that it displays a pictorial representation of the pile of counters on the screen. The display should be updated after each move.

2 Change the program so that, after a game, it asks the program's opponent if he would like another game and terminates only if he says NO.

3 Write a program that plays Noughts and Crosses by making completely random moves. Use the structure introduced in the last section as a framework on which to build your program.

Experiment with different board representations. Three possibilities are:

(a) A 3x3 two-dimensional array.

(b) A one-dimensional array of 9 locations.

(c) (Rather difficult) - A single number that is handled by the program as a bit-pattern. The nine least significant bits indicate the position of the X's and the next nine indicate the position of the O's. Thus:

```
board = ........000001100001010O00
```

might represent the position

```
  |   | X
--+---+---
  | X | O
--+---+---
O |   |
```

You are free to decide which bits represent which squares. Individual bits can be isolated from a number by using DIV and logical operations. This opens up the possibility of very efficient testing for winning positions. We can test for the presence of the winning pattern by comparing the 'board' with the bit-pattern

```
  |   | X
--+---+---
  | X |
--+---+---
X |   |
```

by comparing the 'board' with the bit-pattern

```
.....000000000000l0l0lO0 (ie. &54 in hex)
```

as follows:

```
IF (board AND &54)=&54 THEN ..,,.
```

Placing an X in the top left-hand corner square involves

```
board = board OR &100
```

The bit-patterns corresponding to each possible move and each possible winning pattern could be, stored in a lookup table or could be calculated as powers of 2.

Binary and hexadecimal notation, and the application of logical operations to bit-patterns are described in Appendix 2. If you get the third version working, you will learn a great deal about binary and hexadecimal representation of numbers.