Gateway to Computing with the Amstrad CPC 464

Ian Stewart

SHIVA

BOOK 1

GATEWAY TO COMPUTING

with the Amstrad CPC464

Ian Stewart
Mathematics Institute, University of Warwick

Series Editor
Eleanor Ball

BOOK 1

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
64 Welsh Row, Nantwich, Cheshire CW5 5ES, England

ISBN 1 85014 016 2 (paperback)

Typeset by MHL Typesetting Limited, Coventry
and printed by Devon Print Group, Exeter

Contents

Introduction v

1 Make Friends with your Computer 1

2 Programs 6

3 Running a Program 13

4 PRINTs Charming 20

5 INPUT the Boot 30

6 The Very Able Variable 36

7 All Right for Sums 43

8 IFs but no buts 58

9 Bugs under the Rug 70

10 FOR, 3, TO, 1, NEXT! 78

11 Inside the Computer's Brain 88

Glossary 99

Commands and Symbols Index 104

Ian Stewart is Reader in Mathematics at the University of Warwick, having been Visiting Professor to a number of overseas universities. He has contributed to several computer magazines, including *Sinclair User, Oric Owner* and *Popular Computing Weekly*, and has written for *The Guardian, Nature, New Scientist* and *Scientific American*. Ian also writes occasional science fiction stories for *Analog* and *Omni*, and is a member of the British Science Fiction Association.

Now in his thirties, Ian has already written more than forty books, about half of these being computer books written jointly with Robin Jones including: *PEEK, POKE, BYTE & RAM!, Machine Code and Better BASIC* and *Easy Programming for the ZX Spectrum*. His books have been translated into ten languages. He is an amateur cartoonist under the pseudonym 'Cosgrove' and has published three cartoon books on advanced mathematics – in French – as well as *Computing: a Bug's Eye View*.

Ian lives in a small Warwickshire village with one wife, two sons, and two cats rejoicing in the names *Star* and *Stripes*. His hobbies include home computing, science fiction, playing the guitar, painting scenery and making wine.

Eleanor Ball's first encounter with computers happened by accident in 1966, whilst analysing the chemical properties of wheat and bread flour.

She soon abandoned the direction suggested by a BSc General Degree from London University in favour of computing. The next five years were spent as a Programmer with British Airways, establishing systems on various mainframe computers and terminal equipment at bases throughout Europe.

In 1973, Eleanor retired from city life and came to settle in Cheshire with her husband and young family. It was to be nine years before Shiva discovered her, in the early years of the company, and her interest in computers was rekindled. Eleanor's freelance editorial work for Shiva has turned into an almost full-time occupation, with even the family being caught up in the *Gateway* series, as her husband and children have all shared her involvement in its production and testing!

Introduction

The first electronic computer to be built and put on the market was the ENIAC in 1946. It cost a third of a million pounds, occupied over a hundred cubic metres of space and used as much electricity as two hundred electric fires. Of course, such computers were restricted by size and cost to important areas of scientific research, business and army intelligence. It would have stretched the imagination too much to suggest that computers as powerful as this could become an everyday household object. But that is exactly what they are today – and almost as common as the TV sets we plug into them.

Large mini- and mainframe computers have become essential to business and industry, which now rely on their speed and accuracy to perform as efficiently as possible. Microcomputers – the sort we have in our homes and schools – are not suitable for large-scale application because they are too small and too slow, but they are a great source of fun and can be very stimulating for the individual.

People make very successful careers these days in computer programming – writing instructions to tell the computer what to do. *You* will probably begin by purchasing the software prepared by a professional, but in time you might like to try writing your own programs. It's the best way to appreciate how a computer works and where its strengths and weaknesses lie.

This book is one of a series intended to introduce the fundamentals of computing to young people (and to adults who are still young at heart!). The series is based on the belief that it is possible to be serious about something without being solemn. Learning can be fun!

The material will apply to almost all home computers, not just the machine displayed on the front cover. However, there are always annoying differences between one machine and another, and even if these are small, they are especially puzzling to anyone just starting computing. To avoid these problems each book in the *Gateway* series will appear in several different versions, directed at specific machines, so that the instructions will always fit the machine you are using.

The series begins with one main objective: to get you used to computers and to prove that you *can* write your own programs. You will first be introduced to BASIC – the user-friendly language that both you *and* the computer can understand easily. Later volumes deal with more advanced ideas, such as programming structure and organizing attractive screen layouts.

As well as BASIC programming techniques, you will find problems, puzzles and projects for you to practise what you have learned (I shouldn't tell you this, but all the answers are included, too). In addition to the meanings of BASIC commands, there are useful methods for stringing them together to *do* things. The notorious sleuths, Sherlock Holmes and Dr Watson, will show you the art of debugging – tinkering with a faulty program to make it work. Programs taken from books and computer magazines often contain mistakes; and commercial software may not always work quite the way you want it to. So it's worth learning a few tricks of the trade for ironing out the snags.

As you work through the series, computers will become less of a mystery to you. You'll understand how they work and realize that there is life beyond BASIC. We *have* to know about these machines because *our* future world is *theirs* too. But for once, something important we have to learn about isn't boring – computers are intriguing, creative and rewarding. Most of all, they're fun!

1 Make Friends with your Computer

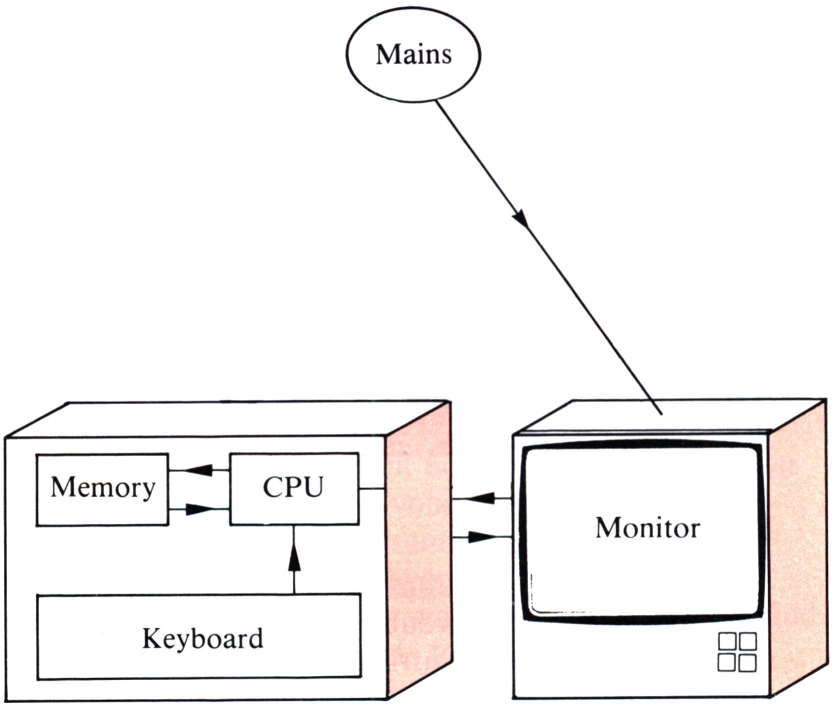
Computers are a bit like people. If you don't take the trouble to get to know them, they can seem very unfriendly. But deep down inside they *can* be very pleasant and helpful. So it pays to make friends with your computer, to learn what it can and cannot do, and to take good care of it.

A computer is a machine that obeys instructions. It works by electricity, and it's built from complicated electronic components. You don't need to know any electronics to be able to use or program a computer. But it helps if you know what the main parts of the machine are and how they connect together.

Programs are lists of instructions to be carried out by the computer. They are stored as electrical signals in its *memory*. The *Central Processing Unit*, or *CPU*, uses more electrical signals to obey these instructions. The *user* (that's you – and an essential component in the whole set-up!) 'talks' to the computer by typing instructions on its *keyboard*.

On home computers the memory, CPU, and keyboard are all inside a single box – the thing you buy from the computer company.

The computer must also be able to 'talk back' to the user. The commonest way it does this is by 'printing' messages on a *TV screen* (also called a *monitor, Visual Display Unit*, or *VDU*) or on a mechanical *printer*. The CPC464 comes with its own monitor as a standard accessory.

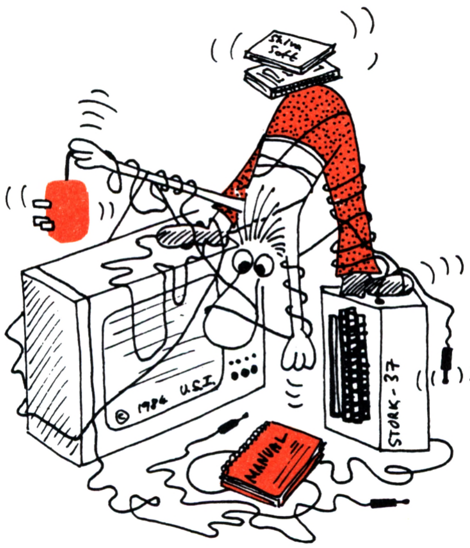


SETTING UP THE COMPUTER

Different computers do things in slightly different ways, but the basic ideas are the same, and I'll run through them quickly. For the full story, consult your *Manual* – the book that came with the machine when you bought it. Starting with the mains switched off at the wall sockets, you must:

- Plug the monitor into the mains.
- Connect the computer to the monitor (using the two leads to the sockets marked 5V DC and MONITOR).
- Switch everything on.

It is important to decide on a sensible and comfortable layout. Ideally, the computer should be on a table or desk with the monitor behind it, so that you can sit in a chair in front of both, with space for the *Manual*, pen and paper, and other equipment like cassettes. DON'T put everything in a heap on the floor. You'll get pins-and-needles from kneeling too long on a hard surface, and there's a danger that the computer will get trodden on.



If you have a green-screen monitor you may wish to use a colour TV sometimes. You'll need to get an MP1 modulator. This comes supplied with special cables which must be plugged into the correct sockets on the computer and TV. The power supply cable is permanently connected to the modulator, and plugs into a socket on the computer. Another cable runs from the modulator to the *aerial* socket on the TV (usually marked UHF on the back of the TV set). Pull out the usual aerial lead and replace it with the cable from the computer.

The TV must be tuned to Channel 36 (either by rotating a dial or by selecting a push-button number you don't normally use, say 6, and tuning that one until it picks up a signal from the computer). When the tuning is correct you will see a message on the TV screen. The monitor gives the same message:

```
  Amstrad 64K Microcomputer (v1)
  © 1984 Amstrad Consumer Electronics plc
             and Locomotive Software Ltd.
  BASIC 1.0
Ready
■
```

Or something like that. You may have to tune contrast, brightness and colour to get the best display. Over-bright displays are bad for the eyes and the TV.

THE KEYBOARD

Next, get the 'feel' of the keyboard. Push keys and see what happens. If the screen gets full of junk, you can always clear it by switching the computer off for a second and then on again.

The keys for the alphabet, and numbers, are arranged in the usual typewriter pattern (which you'll soon get used to):

```
 1 2 3 4 5 6 7 8 9 Ø
Q W E R T Y U I O P
 A S D F G H J K L
  Z X C V B N M
```

and there are various extra keys too, including a second set of number keys to the right of the main keyboard. Some keys have several other symbols printed on them. You get these by holding down the key marked:

SHIFT

and then pressing the key with the symbol on. Holding down CTRL and then pressing another key can also produce some strange symbols.

Notice the way the number *zero* appears:

> The number zero is written
>
> Ø
>
> in computing, to distinguish it
>
> from the letter 'Oh'.

Don't worry too much about SHIFT and CTRL to begin with. Just get used to how the computer reacts when you

press keys at random. (Some keys can have very strange effects, but *nothing you type on the keyboard can damage the computer*. If anything weird happens, just flip the mains off for a few seconds to reset the machine.) Once you've got the feel of the keyboard, try to work out which keys are needed for which symbols, especially those that need SHIFTs.

When you've finished computing, pack everything away tidily. If you've used the family TV, RETUNE IT TO NORMAL SETTINGS. Other members of your household may just possibly want to use the TV to watch *Dr Who*, and they will object if your computer has mucked up the tuning.

From now on I'll use the word 'monitor' to mean either a monitor or a TV.

THINGS TO DO

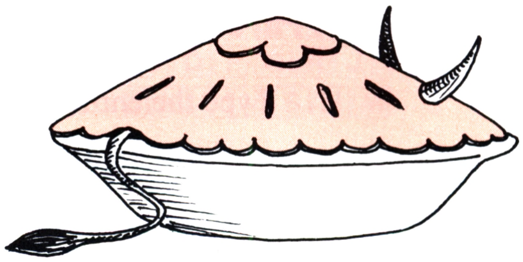
1. Type the entire alphabet on the screen, in order:

   ABCDEFGHIJKLMNOPQRSTUVWXYZ

   You can, of course, use lower case letters if you wish.
2. Find out what happens if you type a sentence that is longer than the width of the monitor screen.
3. Count how many letters wide the display on your computer is. And how many letters high. If you fill the screen with the letter 'L' how many 'L's will you get?
4. *In your Manual*: Read the section on the keyboard and find out exactly how to use the SHIFT keys and CAPS LOCK to get upper and lower case letters.
5. *In your Manual*: Find out where the DEL key is, and how to use it to correct mistakes.

2 Programs

If you want to cook cow pie for lunch, you look up the *recipe* in Despairing Daniel's Cookbook:



First, catch your cow.
Take a large pie-dish.
Line it with pastry.
Add 1 cow.
Add salt and pepper to taste.
Cover with pastry crust.
Bake in oven at 437° (Mark 62) until lightly browned.
(Serves 256 persons.)

This tells you exactly what to do. If you want a computer to do something, you have to tell it the recipe – but a different word is used to describe it:

> A *program* is a list of commands for the computer to carry out.

But what do I mean by commands? Suppose you were working through a recipe that went like this:

First, catch your cow.
Take a large pie-dish.
Line it with pastry.
星期三我们参观了红旗电机厂
Eine Kuh beifügen.
Ajoutez du poivre et du sel à volonté...

Well, you'd be up a Gum Tree without a paddle. Unless you could read Chinese, French, and German. (And if you *could* read Chinese you'd still have problems, because it's not part of a recipe at all: it says 'On Wednesday we visited the Red Flag Electric Machinery Plant'.)

It's the same with computers: you have to use the right language.

A computer language is made up from a lot of standard instructions, together with rules for using them. The computer can only 'understand' the language – that is, carry out the commands successfully – if you obey those rules *to the letter*. The computer can't guess what you mean if you make a tiny mistake. (Maybe that will change over the next few years!)

The computer is supplied with an *interpreter,* held permanently in its memory, which tells it how to obey the commands in that language. The nice thing for the programmer (that's you, again) is that you don't need to know how the interpreter works. All you need to know is the language that it uses.

There are many different computer languages, rejoicing in names like COBOL, FORTRAN, ADA, PROLOG, and C. But for home computers, one language is by far the most popular. It's called BASIC, which stands for:

Beginner's
All-purpose
Symbolic
Instruction
Code

It was invented in America in the 1960s, and looks like a cross between ordinary English and Algebra. Although it has its defects, it is very widely used; and it's the only computer language used in this book.

BASIC BEFORE BREAKFAST

Here is an example of a BASIC program.

```
1  PRINT "GROAN"
2  PRINT "IS IT THAT TIME ALREADY?"
3  PRINT "BOTHER"
4  PRINT "WHO ARE YOU, ANYWAY?"
5  INPUT NAME$
6  PRINT "OH, YOU AGAIN"
7  PRINT "BUZZ OFF,"
8  PRINT NAME$
9  PRINT "I'M HAVING A LOVELY
   SNOOZE"
```

You may well be able to guess what this does. Let's see if you're right! We need to get the program stored in the computer's memory – but first, a word from our sponsor...

LINE NUMBERS

You'll have noticed that, unlike the recipe for cow pie, the commands in the program are numbered.

In BASIC, every program command must start with a number, called its *line number*.

This makes it easy for the computer to refer to a particular command, and to tell which one comes next. Each numbered command is called a *program line* (even if it covers more than one line of printing on the monitor, which is possible).

The numbers do NOT have to go up in ones, as 1,2,3,4,... They can be anything you like (up to 65535).

The computer automatically arranges the lines in numerical order. You *could* number the lines:

13, 42, 43, 255, 777, 779, 3001, 4000, ...

if you wanted to. A favourite way is to count in tens:

10,20,30,40,...

This leaves enough space to insert extra lines at a later stage if you need them. From the next chapter on, I will follow this fashion. But *you don't have to count in tens if you don't want to.*

ENTERING A PROGRAM LINE

Programs are entered into the computer one line at a time. We'll start with line 1.

- Press the 1 key.
- Now press the key for a SPACE
  (This is the long bar at the front.)

The next step is to get the word **PRINT** into the machine. To do this,

- Press keys P R I N T in turn.

(The word **PRINT** may not come out in capitals yet, but the computer will change it to capitals later—see page 12.) Now for the rest of line 1:

- Press the " key.

  (To get quotation marks press the SHIFT key and the 2 key together.)

- Press keys G R O A N in turn.

The word 'groan' will appear on the screen in *lower case* letters. This doesn't matter a bit. In fact, it is easier if you use lower case letters because you don't have to remember to use the SHIFT key as well.

- Press " again (that's SHIFT and 2 as before).

Now you have to press a special key to tell the computer to store all of the above in memory. This key is called:

ENTER

Find it (it's a big blue key on the right of the keyboard) and:

- Press it.
- Check that you have the correct command:

```
1   PRINT "groan"
```

on the monitor.

CORRECTING MISTAKES

Well, gosh. I checked it *ever* so carefully ... but I missed something. It actually reads:

```
1   PLINT "gloan"
```

(That's what you get for buying a Japanese computer, Neddy!) And now the silly thing is in the memory. WHAT DO I DO?

No sweat. Just type the whole line again, and then press:

ENTER

of course. (There are other ways to correct the line, called *editing*: they are completely described in your *Manual*.)

You'll see both versions of line 1 on the screen:

```
1   PLINT "gloan "
1   PRINT "groan"
```

That's OK. Your CPC464 will use only the newest version.

If you've made a mistake in the line number, though, you need to do a little more. Suppose you'd typed:

```
11  PRINT "groan"
```

by mistake. If you just type:

```
1   PRINT "groan"
```

you'll get line *one* right ... but you'll still have a wrong line *eleven*. To get rid of it, type the wrong line number again:

1 1

and then:

ENTER

> To correct a line, just type it again and press:
>
> ENTER
>
> To get rid of a line, type its line number and then press:
>
> ENTER

TYPING IN A WHOLE PROGRAM

To type in a whole program, you just repeat this process for every line. So line 2 is entered by typing the keys:

2 SPACE P R I N T " I S SPACE I T SPACE T H A T

SHIFT required

SPACE T I M E SPACE A L R E A D Y ? "

SHIFT required

and then:

ENTER

Similarly line 5 is entered like this: —

5 SPACE I N P U T SPACE N A M E $ ENTER

dollar sign needs SHIFT

Before we can all move on to the next chapter, you've got a job to do: type in the whole of the program above. Just copy each line.

To *enter* a program (get it into memory) just type each line in turn, and press:

ENTER

after each line.

LISTING A PROGRAM

At any time while you are entering a program (but not in the middle of a program line!) you can check what's in memory by entering the command:

LIST

Type:

L I S T

The computer will then display the complete program *listing*, up to that point, on the screen.

To find out what's stored in program memory, type **LIST** and ENTER

Try it now. You'll notice that **PRINT** and **INPUT** are now in capitals: this always happens when you **LIST** a program.

THINGS TO DO

1. Type that program in, buster!
2. *After* you've worked through the next chapter, get out your *Manual* and read up about *editing* programs.

3 Running a Program

Once the program is safely stored away in memory, and you're satisfied it's correctly typed, you can make the computer obey (or, in the jargon, *execute*) it. The command for this is:

RUN

which you type directly from the keyboard:

R U N ENTER

Assuming you've done what I asked you to, and typed the program in the previous chapter into the computer, you're ready to try it out. Do so.

Unless something is not quite right, you should get this on the monitor screen:

```
groan
is it that time already?
bother
who are you, anyway?
?
```

The computer is now waiting for a response from *you*. The   sign means it's waiting for an *input* (see program line 5, page 8). I'll have a lot more to say about inputs in Chapter 5, but for now, what you must do is type in your name and then ENTER . For example, suppose your name is Hortense Mousebender. Then you type:

H O R T E N S E SPACE M O U S E B E N D E R ENTER

use SHIFT if you like

Now the computer continues:

```
oh, you again
buzz off,
hortense mousebender
I'm having a lovely snooze
```

Then it stops (because it's run out of program to do). There will be a final message—Ready—but you can ignore it.

CRASHES

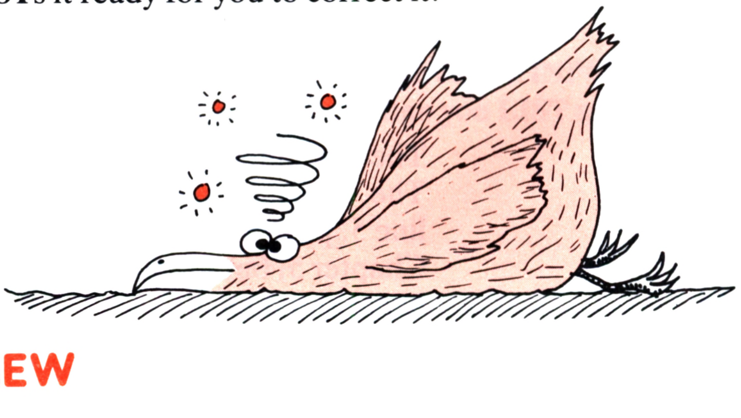
If you make a mistake in a program, it may *crash*. That is, the computer may stop and refuse to continue obeying the program. It only does this when you have asked it to do something that it can't actually do. If a program crashes then the computer will print an *error message* on the monitor screen.

The error must be corrected before the program will run properly. Find the line responsible for the crash, and retype it (I'll say a little more about this in Chapter 9 on Debugging).

In particular if you get a:

```
Syntax error
```

message it means you have broken the grammatical rules of BASIC. The computer tells you which line is wrong, and **LIST**s it ready for you to correct it.



NEW

When you've finished with a program, you must get rid of it *before* typing in another one. Otherwise the two just get mixed up.

One way to do this is to switch off for a second. A better way is to type the command:

NEW

That is, press:

N E W ENTER

Always type **NEW** before entering a new program.

NO NEWS IS BAD NEWS

You probably didn't know that the BBC and ITV both produce scripts for their newsreaders by computer. On the nurth of Octember last year, the ITV news started this way.

```
 20  PRINT "GOOD MORNING. THIS IS THE"
 30  PRINT "ITV"
 40  PRINT "NEWS. THERE IS NO"
 70  PRINT "CROSS-CHANNEL FERRY TO"
100  PRINT "DIEPPE. PASSENGERS SHOULD"
110  PRINT "BE PREPARED FOR"
120  PRINT "DELAYS. AUSTRALIA"
130  PRINT "WON THE FIFTH"
140  PRINT "TEST MATCH WITH ONE"
150  PRINT "RUN OVER"
170  PRINT "A"
180  PRINT "SPOKESMAN FOR MCC SENDS"
190  PRINT "HEARTY CONGRATULATIONS"
200  PRINT "TO THE AUSTRALIAN TEAM."
```

Then the BBC ran their script through the same computer, but *forgot to type NEW first*. The BBC News should have been this:

```
 10  PRINT "BBC NEWS AT TEN"
 30  PRINT "CUP FINAL"
 50  PRINT "RESULT. THE WINNER"
 60  PRINT "IS THE FAVOURITE"
 80  PRINT "NOTTINGHAM FOREST"
 90  PRINT "AND THE LOSER IS"
120  PRINT "WOLVES."
130  PRINT "THE PRIME MINISTER"
140  PRINT "HAS BEEN"
160  PRINT "BY"
180  PRINT "BUS"
200  PRINT "TO BRITISH COACHBUILDERS."
```

What, in fact, did the BBC News come out as? See the end of the chapter for the answer.

UPPER OR LOWER CASE?

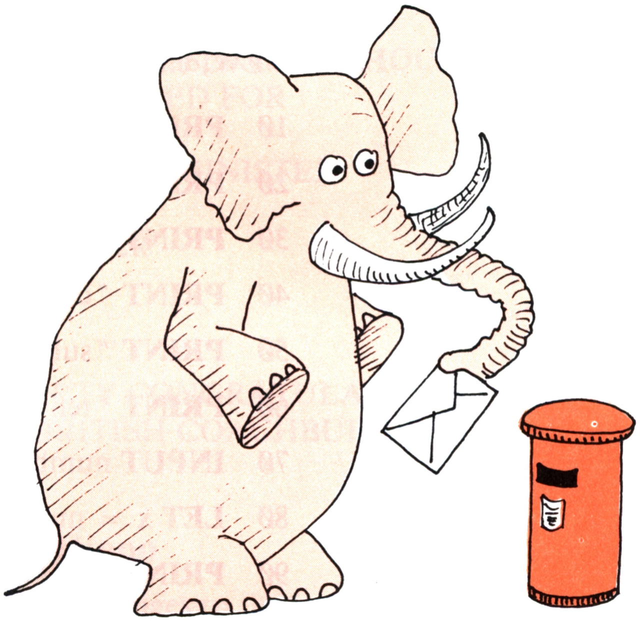
From now on I'm going to list programs in lower case in places where they would normally appear in lower case on the screen, because that makes it easier for you to check that you have entered the lines correctly. (Of course, if you wish to use capital letters throughout then that's OK.)

THE FIRST ELEPHANT JOKE EVER TOLD

Here's another program to practice on. Type it in and **RUN** it. Type **NEW** first, dummy! Don't you listen to *anything* I tell you?

```
10  PRINT "extremely ancient joke"
20  PRINT "do you know the
    difference"
30  PRINT "between an elephant and a"
```

```
40  PRINT "letterbox?"
50  INPUT answer$
60  IF answer$ = "yes" THEN PRINT
    "bother, you've heard it before"
70  IF answer$ = "no" THEN PRINT" I
    won't ask you to post a letter,
    then!"
```



Remember – **NEW** first, then type it in line by line, with a ENTER after each. **LIST** to check it's OK. Then **RUN** it. Input yes or no after the question has been asked.

There are some new commands:

IF and **THEN**

which I'll say more about in Chapter 8.

FRUITFUL HIPPO

Change lines 3∅ and 4∅ above so that the question becomes:

do you know the difference between

a hippopotamus and a fruit-bowl?

Then change line 7Ø so that the computer's answer becomes:

> so that's why my pet hippo has a
> banana stuffed in its ear!

NUMBER MAGIC

This program was devised by the famous conjuror Daniel Pauls. **NEW**, enter, **LIST, RUN.**

```
10   PRINT "think of a number"
20   PRINT "double it"
30   PRINT "add 24"
40   PRINT "divide by 2"
50   PRINT "subtract 9"
60   PRINT "tell me what you get"
70   INPUT number
80   LET x = number - 3
90   PRINT "you chose ";x
```

For instance, suppose you chose 17. Then:

```
number        =   17
double it     =   34
add 24        =   58
divide by 2   =   29
subtract 9    =   2Ø
```

So you input 2Ø at this stage. The computer will tell you:

```
you chose 17
```

which is right.

Try some other numbers. Does it always work? WHY?

ANSWERS

No News is Bad News

If you type in both programs (preferably using CAPS LOCK for everything *inside* "....."), *without* using **NEW**, you will find the BBC News print-out was:

```
BBC NEWS AT TEN
GOOD MORNING. THIS IS THE
CUP FINAL
NEWS. THERE IS NO
RESULT. THE WINNER
IS THE FAVOURITE
CROSS-CHANNEL FERRY TO
NOTTINGHAM FOREST
AND THE LOSER IS
DIEPPE. PASSENGERS SHOULD
BE PREPARED FOR
WOLVES.
THE PRIME MINISTER
HAS BEEN
RUN OVER
BY
A
BUS
HEARTY CONGRATULATIONS
TO BRITISH COACHBUILDERS.
```

Fruitful Hippo

Make these changes to the program. (If it's still in memory, just type the new lines.)

```
30  PRINT "between a hippopotamus
    and"
40  PRINT "a fruit-bowl?"

70  IF answer$ = "no" THEN PRINT "so that's
    why my pet hippo has a banana stuffed in its ear!"
```

Number Magic

Yes, it always works. Take any number x. Double it, you get 2x. Add 24, you get 2x + 24. Divide by 2, you get x + 12. Subtract 9, you get x + 3. This is the input number, and so x is number – 3 (line 80).

4 PRINTs Charming

Now that you've seen a few programs, you're ready to start writing simple ones of your own. To do this, you need to know the commands allowed in BASIC, what they do, and how to put them to good use. As the old song has it:

'It ain't what you do,
It's the way that you do it.
That's what gets results.'

The main commands in BASIC are single words, much like ordinary English. They are called *keywords,* and in this book are always printed in **boldface.** So far we've met the keywords:

PRINT INPUT RUN LIST LET IF THEN NEW

which isn't bad for three chapters. Though, to be fair, we haven't explained all of them in detail yet. The keywords are shown in capitals, because that's what a **LIST** command does to them.

You'll certainly have worked out what the first keyword in that list does:

PRINT causes symbols to be displayed on the monitor.

However, **PRINT** is more versatile, and different kinds of **PRINT** commands can be combined to give different effects.

CHARACTERS

What sort of things can be **PRINT**ed? They have to be symbols that the computer 'knows'. Every computer has a standard list of such symbols including:

Letters	A B C ... Z
Numbers	Ø 1 2 ... 9
Punctuation marks	. : ; , " ? !
Arithmetic	+ – * /
Others	$ # ( ) [ ]

Most computers also have lots of special symbols. On the CPC464 you get these by using CTRL instead of SHIFT when you type. There is a special name given to all these symbols:

A *character* is a single symbol taken from the standard list that the computer can **PRINT**.

PLAIN PRINTS

To **PRINT** a whole sequence of characters, you use the BASIC command in the form:

PRINT "sequence of characters"

quotes

keyword

For example:

PRINT "James Bond ØØ7"

use lower case if you wish

Note that *quotes* " " must be placed at both ends of the sequence of characters. (If the second quote is at the *end* of a program line it can be omitted.)

PRINT PUZZLES

1. Fill in the blanks in the following four short programs, so that they will **PRINT** the messages:

   (a) good morning
   (b) I've got a headache
   (c) that's funny, so is mine
   (d) how dare you!

   Here are the programs:

   (a) 1Ø **PRINT** ″go_ _ _ _ _ _ing″
   (b) 1Ø _ _ _ _ _ ″I've got a headache″
   (c) 1Ø **PRINT** _that's funny, so is mine_
   (d) 1Ø _ _ _ _ _ _how _ _ _ _ _ _
   _ _u!_

2. What is wrong with these commands?

   (a) **PRINT** ″hi there gorgeous!
   (b) **PRIMT** ″woops″
   (c) **PRINT** various things
   (d) **PRINT** ″I karnt spel 2 good″

3. Write a 1-line program to **PRINT** your name backwards.

SILLY SPLITS

Sometimes you may need to add SPACE s to **PRINT** commands to avoid BREAK
ING words like this. For example:

```
PRINT ″my hippopotamus is extremely
hard of hearing″
```

would come out as:

```
my hippopotamus is extremely hard of hea
ring
```

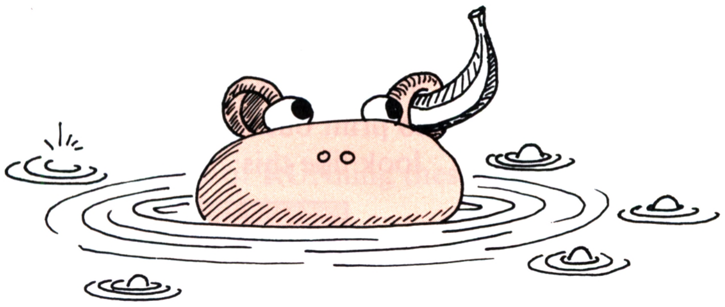
But you can pad it out with spaces like this:

```
PRINT "my hippopotamus is extremely
hard of SPACE SPACE SPACE SPACE
hearing"
```

with the effect:

```
my hippopotamus is extremely hard of
hearing
```

Why is my hippopotamus extremely hard of hearing? All together now: BECAUSE IT'S GOT A BANANA IN ITS EAR!



If a sentence is too long to get into a single **PRINT** command, break it into pieces.

PRINTING ON SEVERAL LINES

At the end of a **PRINT** command, the computer automatically moves on to the next line of the monitor screen, ready for the *next* **PRINT**. (I'll explain later how to prevent this.) See what happens with this program:

```
10  PRINT "the"
20  PRINT "quick"
30  PRINT "brain's"
40  PRINT "facts"
50  PRINT "jump"
60  PRINT "over"
70  PRINT "the"
```

```
80  PRINT "lazy"
90  PRINT "dogma"
```

You can also leave a line of the screen blank (to separate things) by using the command **PRINT** on its own:

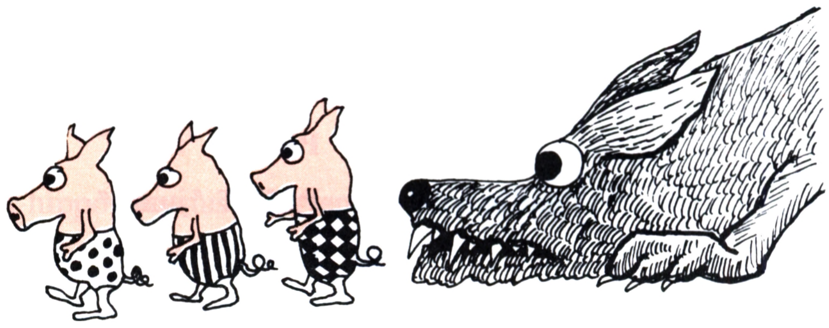
```
130  PRINT
```

PROGRAMMED PIGS

The Lower Standards Repertory Company is putting on an Easter pantomime and the Director, Nigel Podlington-Wally, is a computer buff, so he wants to write a program to print out the programme. (Got that?) It will have to look like this:

```
THE THREE LITTLE PIGS

CAST IN ORDER OF APPEARANCE:

PERKY PIG
PORKY PIG
PUNKY PIG
SNAGGLETOOTH BALDPATCH, WOLF

ACT ONE: WHO'S AFRAID OF THE
    BIG BAD WOLF?

ACT TWO: WOLF'S AFRAID OF THE
    PIGS' HOT BATH
```

Write him a suitable program.



SEMICOLON

Now it's not always helpful to move on to the next line, after a **PRINT**. So you want to be able to avoid this if you choose. The answer is to put a *semicolon*

$$\boxed{;}$$

after the final quote in the **PRINT** commands, like this:

```
5Ø  PRINT "something";
                     ↑
                 semicolon
```

Compare the effect of **RUN**ning these two programs:

```
(a)  1Ø PRINT "thunder"   ←——— no semicolon
     2Ø PRINT "bird"

(b)  1Ø PRINT "thunder";  ←——— semicolon
     2Ø PRINT "bird"
```

You should have found that program (a) gave:

```
thunder
bird
```

whereas (b) gave:

```
thunderbird
```

all on one line. A tiny semicolon can have an enormous effect – check listings carefully for punctuation! (It is said that early in the American space programme a rocket crashed for lack of a semicolon in the computer guidance program. You might call that a Blunderbird!) Notice that the semicolon not only leaves you on the same line – it doesn't even put in a space! How would you get:

```
thunder  bird
```

with two **PRINT** commands?

The only way is to *tell* the computer to **PRINT** the SPACE too. SPACE is considered to be a character, just like all the rest. Here are two different ways to do it:

```
10  PRINT "thunder";
20  PRINT " SPACE  bird"

10  PRINT "thunder SPACE  ";
20  PRINT "bird"
```

You can also do it by:

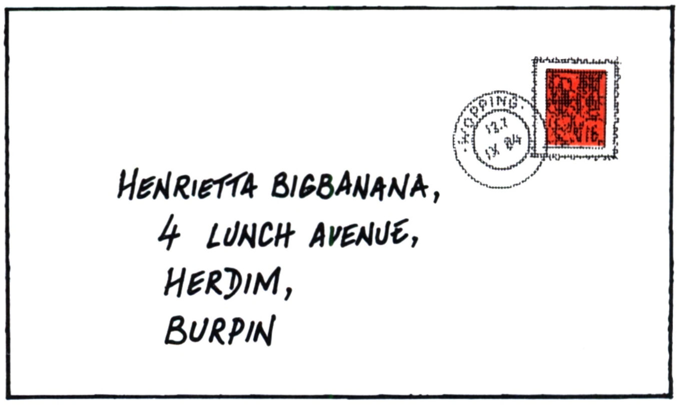
```
10  PRINT "thunder";
20  PRINT " SPACE  ";
30  PRINT "bird"
```

or you can string a lot of **PRINT** commands together like this:

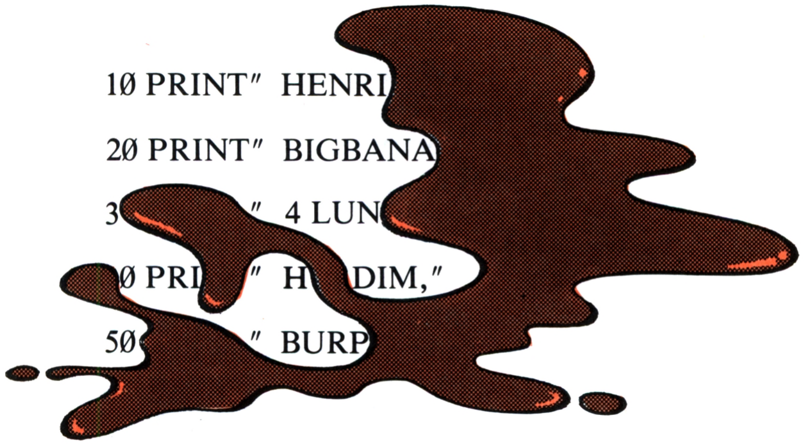
```
10  PRINT "thunder"; " SPACE  ";
    "bird"
```

PUZZLE POSTCARD

Poking Pete the Peerless Programmer has written a program to address a postcard:



Unfortunately his pet armadillo tipped a bottle of HP sauce over it. Can you fill in the missing parts?



```
1Ø PRINT"  HENRI
2Ø PRINT"  BIGBANA
3        "  4 LUN
 Ø PRI   "  H   DIM,"
5Ø       "  BURP
```

COMMA

The comma $\boxed{,}$

is used in a similar way to the semicolon, but with a different effect. It works like this. The screen is divided up into three columns. After a comma, the **PRINT** position moves to the next column in the line – or skips to the next line if all columns have been used. I don't want to make a big song and dance about this ... but you can easily check it out on your computer by running the following test program.

```
1Ø   PRINT "tiger","hare","dragon",
2Ø   PRINT "snake","horse","goat",
           "monkey",
3Ø   PRINT "cockerel", "dog",
4Ø   PRINT "pig","rat","ox"
```

CLEARING THE SCREEN

After you've used the computer, there's a lot of junk floating around on the screen. It's often a good idea to get rid of it.

There is a BASIC keyword to clear the screen. On most computers, including the CPC464, it is:

```
CLS
```

I've left you to do the tidying up in this book, so to produce neat displays on the screen, you should feel free to add program lines like:

```
175  CLS
```

to the programs you copy, or use.

ANSWERS

PRINT Puzzles

```
1.  (a)  1Ø  PRINT "good morning!"
    (b)  1Ø  PRINT "I've got a headache"
    (c)  1Ø  PRINT "that's funny, so is mine"
    (d)  1Ø  PRINT "how dare you!"
```

2. (a) No final quote.
   (b) **PRINT** misspelled.
   (c) No quotes.
   (d) As far as the computer is concerned, there's nothing wrong with it at all!

3. This is Hortense Mousebender's answer:

```
1Ø  PRINT "rednebesuoM esnetroH"
```

Programmed Pigs

(You need CAPS LOCK for this!)

```
1Ø  PRINT "THE THREE LITTLE PIGS"
2Ø  PRINT
```

```
      PRINT  "CAST IN ORDER OF
      APPEARANCE:"
40  PRINT
50  PRINT "PERKY PIG"
60  PRINT "PORKY PIG"
70  PRINT "PUNKY PIG"
80  PRINT "SNAGGLETOOTH BALDPATCH,
      WOLF"
90  PRINT
100 PRINT  "ACT ONE: WHO'S AFRAID OF
      THE"
110 PRINT " SPACE SPACE SPACE BIG
      BAD WOLF?"
120 PRINT
130 PRINT  "ACT TWO: WOLF'S AFRAID OF
      THE"
140 PRINT " SPACE SPACE SPACE PIG'S
      HOT BATH"
```

Postcard Puzzle

(You need CAPS LOCK for this one too!)

```
10  PRINT "HENRIETTA  SPACE  ";
20  PRINT "BIGBANANA,"
30  PRINT "4 LUNCH AVENUE,"
40  PRINT "HERDIM,"
50  PRINT "BURPIN"
```

5 INPUT the Boot

When you use **PRINT** to send a message from the computer to the monitor, that's an example of *output* – communication from the computer to the outside world. The opposite – sending messages from the outside world to the computer – is called *input*.

OUTPUT is what goes out,
INPUT is what goes in.

When you type a program from the keyboard, that's input. And there's even a BASIC keyword:

INPUT

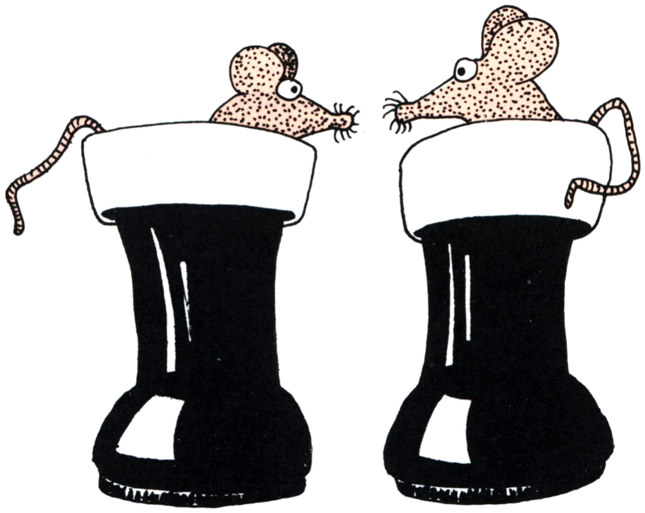
that tells the computer to fetch a message from the keyboard. You've used it already, but I haven't explained what it's all about.

NAMED INPUTS

When you use **INPUT** you must include a 'code name' so that the computer can tell which input is which. So the command takes the form:

INPUT codename

For instance, suppose you want to write a program to work out the price of a pair of Wellington boots, given the prices of the left and right boots separately. (Look, I'm *trying* to be realistic, but it's not always possible.) The computer needs to know the two prices, and it needs to know which



is which (in case you also wanted to buy three left boots and seven right boots for an extraterrestrial acquaintance). So you invent two codenames:

    leftboot   rightboot

to distinguish them. Then the program is a doddle:

```
10  PRINT "price of left boot?"
20  INPUT leftboot
30  PRINT "price of right boot?"
40  INPUT rightboot
50  PRINT "Total price is ";
60  PRINT leftboot + rightboot
```

RUN this, and try the inputs:

    2 ENTER

for the left boot and:

    3 ENTER

for the right. You should get the answer:

```
Total price is 5
```

I haven't bothered to say what sort of money: pounds, dollars, roubles, corporals—that's up to you.

You'll have noticed that the computer gives you a ? sign to show it wants an input.

? asks for an input—that is, a message or a number. Type your input and then ENTER.

Try the program again with:

leftboot input 1234567
rightboot input 7654321

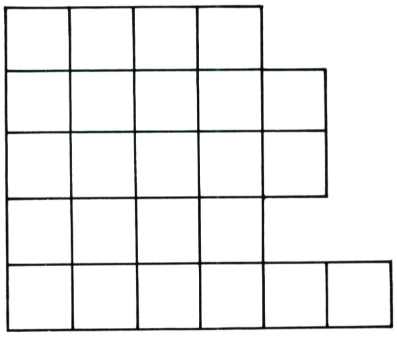
And if you're a decimal freak, try something more plausible, like

leftboot input 3.42
rightboot input 3.69

What answers do you get?

COMPUCROSTIC

Solve the *across* clues, and spot the BASIC keyword among the *down* words.



Clever thought

Opposite of day

Mini-dog

A single piece

Common to fishing and football

THE ALMIGHTY DOLLAR

That's how to input *numbers*. To input *words* there is one minor change. The codename must end in a dollar sign:

$

For example:

```
10   PRINT "good morning"
20   PRINT "this is your friendly"
30   PRINT "neighbourhood computer."
40   PRINT "you're new, aren't you?"
50   PRINT "what's your name?"
60   INPUT name$ ◄── dollar sign on codename
70   PRINT "hi, SPACE ";name$;
80   PRINT " SPACE welcome to the
     swamp!"
```

The $ sign tells the computer to expect a sequence of characters (especially letters, but you can use any other symbols too) rather than a number. Such a sequence of characters is called a *string*. Here are some strings:

FRED
OMO
R2D2
**!%#@!

A *string* is just a load of characters *strung* together.

The codename for a $tring **INPUT** mu$t fini$h with a dollar $ign ($).

PRACTICE PROBLEMS

Write a program that lets you **INPUT** a name, and prints it out five times one beneath another.

PROMPTS

In most of the inputs above, I have **PRINT**ed a message to remind you what it is that has to *be* input. This is a very useful trick: a bare ? isn't always very comprehensible.

A message to remind you what the input is for is called a *prompt*.

On the CPC464 you can build a prompt into an **INPUT** command. Instead of something like:

```
5Ø   PRINT "what's your name, honey?"
6Ø   INPUT name$
```

you can write just:

```
                                          codename
                                              ↓
5Ø   INPUT "what's your name, honey";name$
           └─────────────┬────────────┘↑
                      prompt       semicolon
```

I haven't used this short-cut in this book, but don't let that stop you!

ANSWERS

Named Inputs

```
Total price is 8888888
Total price is 7.11
```

Compucrostic

```
 I D E A
N I GH T
P U P P Y
U N I T
T A C K L E
↑───INPUT
```

Practice Problem

```
10  PRINT "input a name"
20  INPUT name$
30  PRINT name$
40  PRINT name$
50  PRINT name$
60  PRINT name$
70  PRINT name$
```

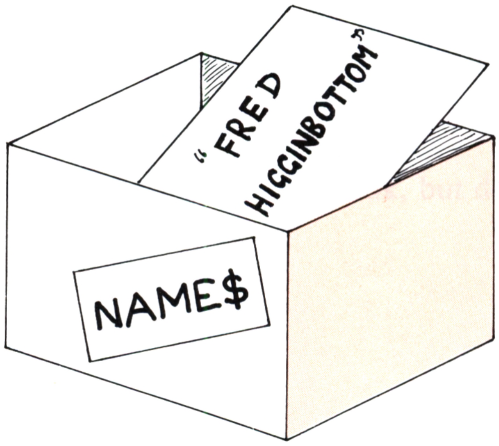
for a better way see Chapter 10

6 The Very Able Variable

In the last chapter I said we use 'codenames' for inputs. Now let me tell the whole truth. Well ... *some* of the whole truth!

You can think of the computer's memory as a lot of boxes. Each box contains one piece of information.

When you give a computer an **INPUT,** it stores it away in a box. It uses the 'codename' as a kind of *label* on the box, so that it can tell which box is which.



(Actually, it usually needs several boxes in a row, but let's not go into fine details at this stage, huh?)

A memory box with a label on it is called a *variable.* I'll tell you why in a moment.

There are two main types of variable in BASIC:

1. A *numeric* variable – which contains a number.
2. A *string* variable – which contains a string of characters.

The codename for a variable consists of letters and numbers. If it's a string variable, the codename must end in a $ sign. The codename, that is, the label on the box, is referred to as the *name* of the variable. Unlike a BASIC keyword, which must come from a standard set of commands, variable names may be chosen by the programmer. There are a few rules governing that choice: see page 39.

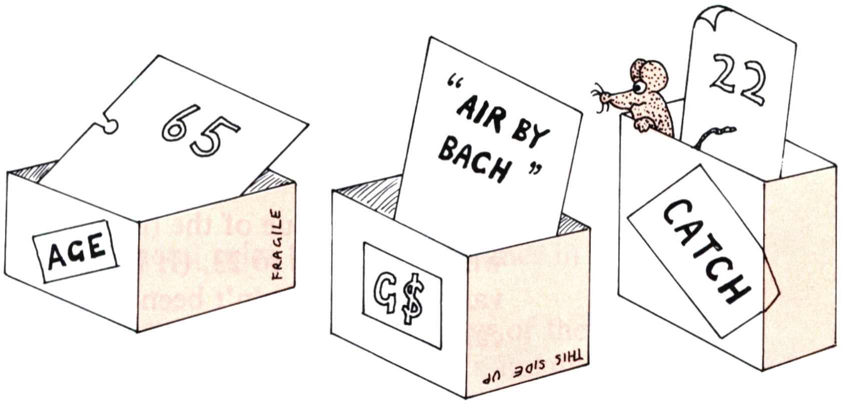
A *variable* is a memory box with a codename that acts like a label.

The label is the *name* of the variable.

The contents of the memory box is the *value* of the variable.

PETE'S PUZZLE

Peripheral Pete the Perfidious Programmer has set up some variables. What are their names? What are their values? Are they numeric or string variables?



THE SEMICOLON REVISITED

As you may remember, using a semicolon between two **PRINT** strings (or string variables) causes the strings to be joined together, with no extra SPACE s added:

PRINT "thunder"; "bird"

gives thunderbird. However, if you use a semicolon when **PRINT**ing *numeric* variables, each item is **PRINT**ed with a SPACE on either side. For example:

```
10  PRINT "what is your age?"
20  INPUT age
30  PRINT "I think you SPACE ";
```

SPACE required for strings

```
40  PRINT "will be"; age + 1;"next year"
```

no SPACE for numeric items

VARYING A VARIABLE

Once you have set up a labelled memory-box, you can change its contents *without* changing the label. (This idea is widely used by the smuggling fraternity.) That is, you can alter the *value* of a variable without changing its *name*.

The way to do this is to use the keyword:

LET

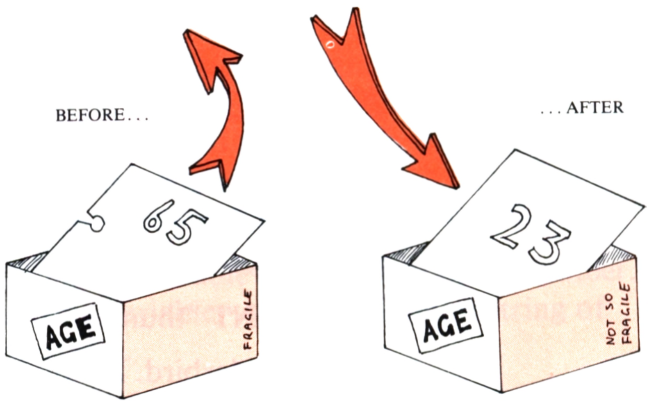
in the form:

LET variablename = value

For example:

LET AGE = 23

changes the value of the (numeric) variable AGE from whatever it was to 23. (If it wasn't anything – that is, the variable AGE hadn't been set up – it sets up a new variable called AGE and puts the value 23 in it.)



The next program uses this ability of variables. It works out the change (from 100 pence) for three different payments – 17, 32 and 81 pence.

```
1Ø   LET amount = 17
2Ø   PRINT 1ØØ – amount
3Ø   LET amount = 32
4Ø   PRINT 1ØØ – amount
5Ø   LET amount = 81
6Ø   PRINT 1ØØ – amount
```

Notice that the *same* box holds the three amounts in turn.

> Variables let you write *general* programs to handle a range of different items in the same way.

On the CPC464, the word **LET** is optional and may be omitted. For example:

```
AGE = 23
```

will give the variable AGE the value 23.

On some computers the **LET** is *compulsory*. I will always use **LET** in this book.

RULES FOR VARIABLE NAMES

The general rules for variable names in BASIC are:

1. They must start with a *letter* of the alphabet.
2. You can use as many letters and numbers as you like, up to a reasonable limit (4Ø) BUT don't use any SPACE s.
3. The name may not be a BASIC keyword.

These rules are to help the computer understand commands. If you break them, you will get a Syntax error message.

Rule 1 means that you can't use a name like:

```
2K        *ABC        (FRED)
```

not a letter

Rule 2 means that you can't use:

pocket money

as a variable name

Rule 3 means you can't use names like:

input run list

but who'd want to, anyway!

PROGRAM PROBLEM

Write a program that uses two string variables f$ and s$ to input a person's first name and surname separately – and then **PRINT** them out *with the surname first,* as in:

WHITTINGTON , DICK

(Well, he *was* told to 'turn again', wasn't he?)

THE HOUND OF THE BASKETBALLS

Mr Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. An irregular piece of parchment lay before him, at which I glanced in curiosity. It appeared to be a list of names—outlandish, wild names: £5-NOTE, TOMATO SAUCE, SNAGGLEPUS$, $NIGGLEPUSS, $NUGGLEPU$$, 99-BONK!, ØØ7, BANANA SKIN, 7-UP AND GEORGE WASHINGTON.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of my occupation.

"How did you know what I was doing? I believe you have eyes in the back of your head!"

"Simple logic, Watson. I know you're a nosey old coot. You wouldn't be able to resist such an opportunity."

I sustained a dignified silence for a few moments. Then, "But what *is* it, Holmes?" I burst out.

"It is a communication from Sir Tobias Basketball of Basketball Hall, Watson. Precisely, it is a list of his foxhounds."

"What strange names," I mused.

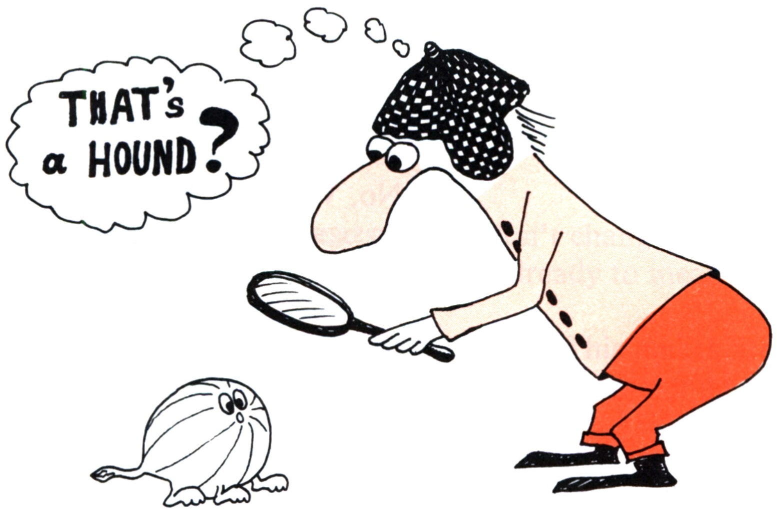
"Indeed, Watson, indeed." Holmes drew deeply upon his pipe and transfixed me with his steely eye. "And one of them – just one, mark you – is an *imposter*.

'The question is, Watson: which?"

"I have no idea," I replied.

"But Watson, it is utterly elementary. Can you not see that there is a BASIC distinction which singles out one name?"

I must confess, I could not. Perhaps my gentle reader will fare better than I.



ANSWERS

Pete's Puzzle

Name	Value	Type
AGE	65	Numeric
G$	"AIR BY BACH"	String
CATCH	22	Numeric

Program Problem

```
10  PRINT "what is your first name?"
20  INPUT f$
30  PRINT "what is your surname?"
40  INPUT s$
50  PRINT "your name, surname first,
    is"
60  PRINT s$;",";f$
```

The Hound of the Basketballs

"When I tell you, Watson, you will be overcome by the irony of it all."

"I doubt it, Holmes. But what *is* the answer?"

"The first clue is the presence of dollar signs, Watson."

A light dawned. "American! The imposter is an American foxhound!" I scanned the list rapidly. "GEORGE WASHINGTON, right Holmes?"

"No, Watson," he said slowly. "It is *not* American. The dollar signs suggest that the names are BASIC variables."

"Oh."

"In addition, hardly any of the names could actually be used on a computer. Most are illegal. Thus, TOMATO SAUCE, BANANA SKIN, and GEORGE WASHINGTON all contain *spaces*; and £5-note, $NIGGLEPUSS, $NUGGLEPU$$, 99-BONK!, ØØ7, AND 7-up do not start with a *letter*."

"Aha!"

"Which leaves, as the odd man out, the name SNAGGLEPUS$. The impostor – "

"Is the only *legal* name, Holmes!" A thought struck me. "The *illegal* imposter signalled by a *legal* name!" I paused. "How ironic!"

Holmes's eyeballs rolled skywards once more. I must persuade him to see a doctor.

I've just remembered something.

I'm a doctor.

He should definitely see a vet.

7 All Right for Sums

Like it or not, if you're going to tangle with computers you'll need to use arithmetic every so often. Fortunately the computer knows how to do all the hard stuff. In fact, sums can be fun – if you make somebody else do the work!

The fundamental operations of arithmetic are:

+	Add
–	Subtract
×	Multiply
÷	Divide

In BASIC, two of these symbols don't change, namely + and –. Indeed, we've used them already to mean 'add' and 'subtract'. (Where?)

But × and ÷ are changed to * and /. This makes life easier for the computer and stops the programmer getting confused. On a monitor × looks much like X and ÷ is easily muddled with +. The easiest thing is not to use them.

ADD (+)

This is just as in arithmetic. To add the values of two variables A and B we pick a variable C to hold the result, and write:

```
70   LET C = A + B
```

(assuming it *is* line 70 that this occurs in). Here's an example that adds a whole lot of numbers at once...

"Can you do Addition?" the White Queen asked.

"What's one and one and one and one and one and one and one and one and one and one?"
"I don't know," said Alice. "I lost count."
"She can't do Addition," the Red Queen interrupted.

But the computer…

```
10   PRINT "answer to the white
     queen's problem"

20   PRINT "1+1+1+1+1+1+1+1+1+1
     SPACE  = ";

30   PRINT 1+1+1+1+1+1+1+1+1+1
```

Notice the difference the quotes make in lines 2Ø and 3Ø. If they are present, then 1+1+1+1+1+1+1+1+1+1 is treated as a *string* and just *copied* out. But if they are missing it is treated as a *number* and *worked* out.
Similarly, if you write:

```
100   PRINT ANSWER
```

then the computer looks for a variable named ANSWER, and **PRINT**s out what its *value* is. But:

```
100   PRINT "ANSWER"
```

just produces the *word* ANSWER. *This is true even if there is a variable named ANSWER.*

DRILL PROBLEMS (+)

to remind you this is arithmetic – serious stuff!

Write three computer programs to work out:

(a) 1+2+3+4+5+6+7+8+9+1Ø
(b) 99999+1
(c) 1ØØØØØØ+1ØØØØ+1ØØ+1

Don't write *commas* to separate thousands or millions (as in 1,ØØØ,ØØØ for a million). The computer won't understand. Write 1ØØØØØØ with no commas, and be careful to count that you have the right number of Øs.

SUBTRACT (–)

Again, this is as in ordinary arithmetic:

```
530  LET C = A - B
```

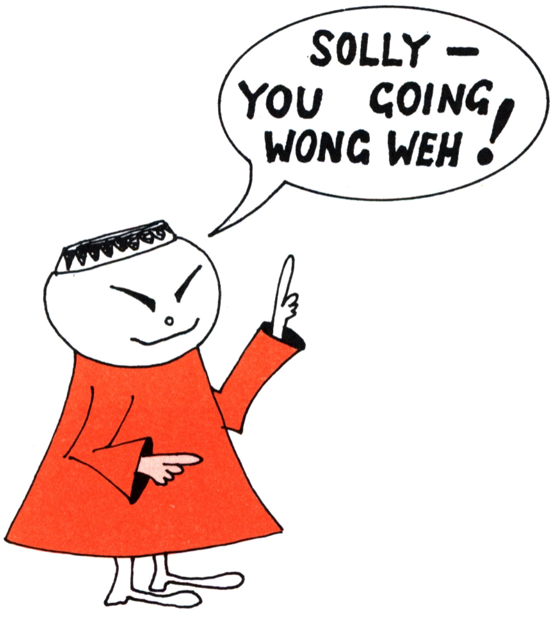
assigns to the variable C the value obtained by subtracting the value of B from the value of A.

The Wong Weh Chinese Restuarant has an arrangement with its regular customers. The customer deposits a sum of £1ØØ with the restaurant. He or she may then order food by telephone. The price is subtracted from the amount deposited and the remainder is left in the customer's account.

(This is called a Chinese Take-Away.)

Here is a program to input the price of an order, and tell the customer how much is left.

```
1Ø  LET deposit = 1ØØ
2Ø  INPUT price
3Ø  LET deposit = deposit – price
4Ø  PRINT "you now have ";
      deposit; " left"
```



Three customers purchase the following meals:

(a) Hortense Mousebender: 1 goldfish fin soup, price: 17.23
(b) Sue Choppy: 1 crispy jellyfish with noodles, price: 2.74
(c) Tommy Upset: Flied Lice in black bean sauce with One-tonne Won-Tons; Noodle noodle soup; Mangrove shoots in water lilies; Wong Weh special hippopotamus Chow Mein; and chips. Total price: 98.72

RUN the program three times, to find out how much each had left.

ALGEBRA IT AIN'T

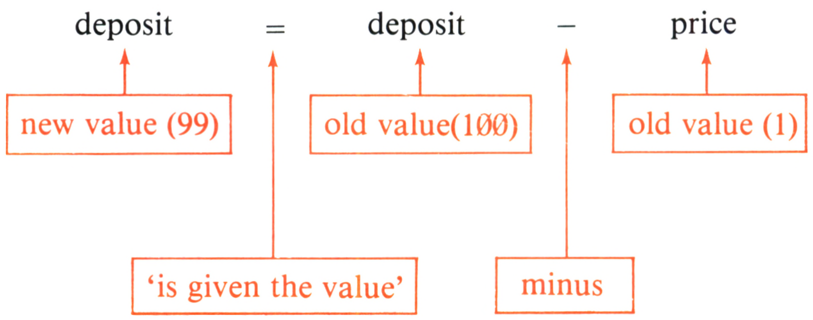
There is something a little peculiar about line 3Ø. Suppose the value of price is (say) 1. Then deposit 1ØØ, price = 1, so deposit – price = 99; and line 3Ø seems to be saying:

LET 1ØØ = 99

which is hard, to say the least!

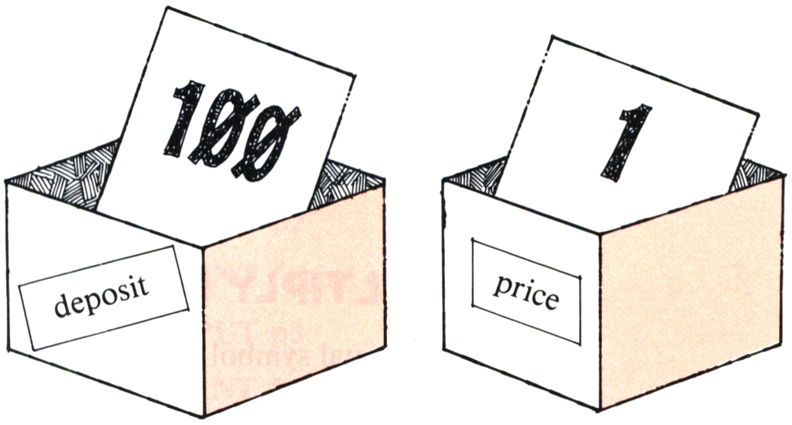
There's no real problem, however. The *equals* sign = in BASIC doesn't really *mean* 'equals'. It means 'is given the value'. The part *before* the = sign is the new value of a variable, to be found by working out the part *after* the = sign. And while that's being worked out, the variables involved retain their old values.

Got that? It's like this:



So inside the computer, what happens is this:

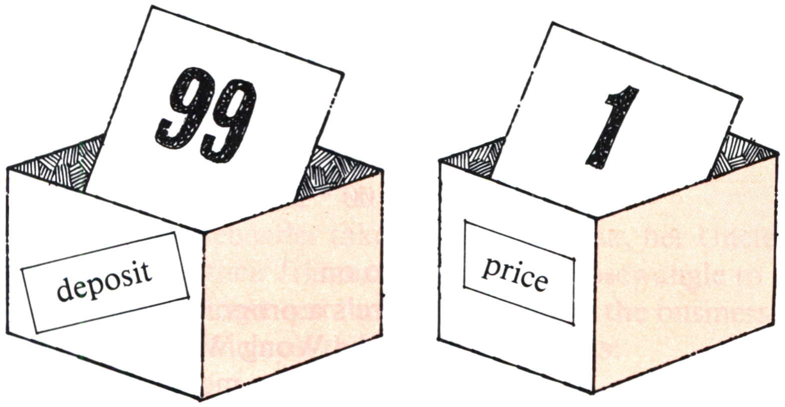
1. Start with these values:



2. Work out deposit – price,

   which is 100 – 1,
   which is 99.

3. Change the value held in deposit to this new value:



Note that the value of price (1) is unchanged.

> Once a variable has been given a value, that value stays the same unless you tell the computer to change it.

DRILL PROBLEMS (–)

Write programs to work out:

(a) 3Ø1 – 6Ø
(b) 1ØØ – 1 – 2 – 3 – 4 – 5
(c) 7Ø7Ø7Ø – 181818
(d) 1 – 2 + 3 – 4 + 5 – 6 + 7

MULTIPLY (*)

The usual symbol for multiplication is:

×

But programmers could easily confuse this with:

X

To avoid that danger, BASIC uses an *asterisk* (or star):

$\boxed{*}$

(That way yer doesn't 'as ter risk confusing it with X!)
For instance, in BASIC:

3 * 5 = 15
7 * 7 = 49
1ØØ * 22 = 22ØØ

and so on.
Here's a program that uses . A party of several people visits the Wong Weh Chinese Restaurant for lunch. All order the same menu, the 3-course businessman's lunch. The prices of the courses must be input as three variables:

p1 p2 p3

and the number of people in the party as:

number

The total cost is obtained by adding the first three inputs together to get the price of one meal,

meal = p1 + p2 + p3

and then multiplying this by the fourth of the input variables, number (hence obeying the Biblical injunction, 'Go ye fourth and multiply').

```
 10  PRINT "first course?"
 20  INPUT p1
 30  PRINT "second course?"
 40  INPUT p2
 50  PRINT "third course?"
 60  INPUT p3
 70  PRINT "number of people?"
 80  INPUT number
 90  LET meal = p1 + p2 + p3
100  LET price = meal * number


110  PRINT "total cost is "; price
```

HORTENSE PIGS OUT

Hortense Mousebender takes her Aunt Hilda, her Uncle Jeremiah, and their friends Mr and Mrs Nosewangle to the Wong Weh Chinese restaurant. They order the business-man's lunch, which on that particular day is:

1st course:	Porridge soup and sauerkraut	2.73
2nd course:	Sweet and sour spaghetti	3.29
3rd course:	Chili ice cream	1.95

RUN the program to find out what it cost them.

DRILL PROBLEMS

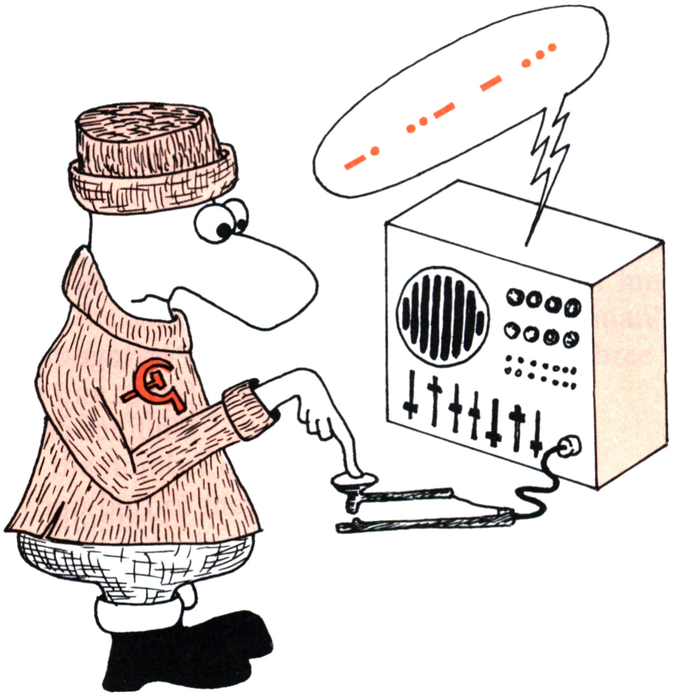
Write computer programs to work out:

(a) 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9
(b) 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
(c) 7 * 7 * 7 * 7
(d) 1 + 2 * 3 + 4 * 5 + 6 * 7 + 8 * 9

AYE-AYE, IT'S MY LITTLE SPY

The Russian Master-Spy Ivan Nokyablokov is sending a computer program in code to his controller, Major Igor Biva of the KGB. He had coded it by changing each letter of the alphabet to a different one, in a systematic way (so that, for example, every 'P' has been replaced by 'X').

Can you, Cheerful Charlie, the Cunning Cryptographer, break the code and work out the program? Here is the coded version. (You'll need CAPS LOCK for this!)



```
10  XLVJS "STX MXHWO MOWLOS"
20  XLVJS "KOOGNZ MXHWO
      LHVJYHNN"
30  XLVJS "VJXDS MXHWO BHVNZ
     MXHWO LHVJYHNN"
40  VJXDS LHVJ
50  NOS LHVJ = 7 * LHVJ
60  XLVJS "LHVJYHNN MXHWO VM
     MXHWO"; LHVJ; " MXHWO OCOLZ
     MXHWO KOOG"
70  XLVJS "BOMSLTZ MXHWO HYSOL
     MXHWO LDJJVJA"
```

DIVIDE (/)

Ordinary arithmetic uses:

÷

for division. But BASIC uses a *slash*:

/

For instance, in BASIC, you must write:

```
6 / 2 = 3
20 / 5 = 4
49 / 7 = 7
```

and so on.

You'll also find that:

```
1 / 3
```

comes out as:

```
.33333333
```

because BASIC uses a decimal point when it's asked to work out fractions.

I've tried not to use decimal notation in this book, except for pounds and pence, such as £4.23.

DRILL PROBLEMS

Write computer programs to work out:

(a) 365 / 5
(b) 1ØØØØØØ / 1ØØ
(c) 727272 / 9

DIVIDE AND CONKER

Hortense's youngest son, Mickey Mousebender, and his friends have been throwing sticks into a conker tree. Suppose that there are n boys and they get k conkers altogether. (k, because they karnt spel 2 good.) If the conkers are shared equally between them, how many will each boy get?

```
1Ø   PRINT "number of boys?"
2Ø   INPUT n
3Ø   PRINT "number of conkers?"
4Ø   INPUT k                      slash for division
5Ø   LET share = k / n
6Ø   PRINT "there are "; share;
     " conkers per boy"
```

This program works well if the number of conkers divides exactly. It doesn't give particularly sensible results otherwise. Try using it to divide 47 conkers among 7 boys! Division *with remainder* is possible in BASIC (and would give an answer of 6 per boy with 5 left over) but is beyond the scope of this book.

BRACKETS ( )

These are used, as in arithmetic, to specify the order in which calculations are carried out.

The standard rule (called a *priority rule* because it tells you what to do first) is that:

* and /

are worked out *before*:

+ and −

So something like:

12 + 4 / 2

is worked out like this:

First do the 4 / 2 = 2

Then do 12 + 2 = 14

It is *NOT* worked out doing the '+' first, like this:

First do 12 + 4 = 16

Then do 16 / 2 = 8

See? It matters!

Now sometimes you may *want* to do the addition first. If so, you must tell the computer this, by using brackets:

(12 + 4) / 12

Anything inside *brackets* is worked out completely before anything else is done to it.

I'm going to use brackets to simplify the program about the Chinese Restaurant in the MULTIPLY section above. It can be shortened to this:

```
10  PRINT "first course?"
20  INPUT p1
30  PRINT "second course?"
40  INPUT p2
50  PRINT "third course?"
60  INPUT p3
70  PRINT "number of people?"
```

```
                                          brackets!
 8Ø  INPUT number
 9Ø  LET price = (p1 + p2 + p3) * number
1ØØ  PRINT "total cost is "; price
```

Brackets must always be used in pairs: '(' at the front and ')' at the end.

Notice that you can't just write:

```
p1 + p2 + p3 * number
```

in line 9Ø. For example, suppose:

```
p1 = 5     p2 = 7     p3 = 1Ø     number = 4
```

Then:

```
p1 + p2 + p3 * number = 5 + 7 + 1Ø * 4
                      = 5 + 7 + 4Ø
                      = 52
```

whereas:

```
(p1 + p2 + p3) * number = (5 + 7 + 1Ø) * 4
                        = (22) * 4
                        = 88
```

MOUSEBENDER'S MUDDLE

Marmaduke Mousebender the Mad Mathematician has found four different answers to the same arithmetic problem:

```
(a)  2 * 3 + 4 * 5 = 26
(b)  2 * 3 + 4 * 5 = 7Ø
(c)  2 * 3 + 4 * 5 = 5Ø
(d)  2 * 3 + 4 * 5 = 46
```

Can you put in some brackets to make all of these correct?

The arithmetic symbols in BASIC are:

+	add
−	subtract
*	multiply
/	divide
( )	brackets

The symbols × and ÷ are NEVER used.

ANSWERS

Drill Problems (+)

(a) 1Ø **PRINT** 1+2+3+4+5+6+7+8+9+1Ø
Answer 55.
(b) 1Ø **PRINT** 99999+1
Answer 1ØØØØØ.
(c) 1Ø **PRINT** 1ØØØØØØ+1ØØØØ+1ØØ+1
Answer 1Ø1Ø1Ø1.

Subtract (−)

Hortense Mousebender had 82.77 left.
Sue Choppy had 97.26 left.
Tommy Upset had 1.28 left.

Drill Problems (−)

(a) 1Ø **PRINT** 3Ø1−6Ø
Answer 241.
(b) 1Ø **PRINT** 1ØØ−1−2−3−4−5
Answer 85.
(c) 1Ø **PRINT** 7Ø7Ø7Ø−181818
Answer 525252.
(d) 1Ø **PRINT** 1−2+3−4+5−6+7
Answer 4.

Hortense Pigs Out

There are five people in the party. The meal cost 39.85.

Drill Problems (*)

(a) 1Ø **PRINT** 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9
Answer 36288Ø.
(b) 1Ø **PRINT** 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
Answer 1Ø24.
(c) 1Ø **PRINT** 7 * 7 * 7 * 7
Answer 24Ø1.
(d) 1Ø **PRINT** 1 + 2 * 3 + 4 * 5 + 6 * 7 + 8 * 9
Answer 141.

Aye-aye, it's my Little Spy

The code is:

```
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ H   W B O Y     V   G N   J T X   L M S D C K   Z
```

(The other letters aren't used, so you can't decide what they are.)
The program was:

```
1Ø  PRINT "TOP [SPACE] SECRET"
2Ø  PRINT "WEEKLY [SPACE] RAINFALL"
3Ø  PRINT "INPUT [SPACE] DAILY [SPACE]
    RAINFALL"
4Ø  INPUT RAIN
5Ø  LET RAIN = 7 * RAIN
6Ø  PRINT "RAINFALL [SPACE] IS [SPACE]"; RAIN;
    " [SPACE] EVERY [SPACE] WEEK"
7Ø  PRINT "DESTROY [SPACE] AFTER [SPACE]
    RUNNING"
```

Drill Problems (/)

(a) 1Ø **PRINT** 365 / 5
Answer 73.

(b) 1Ø **PRINT** 1ØØØØØØ / 1ØØ
Answer 1ØØØØ.

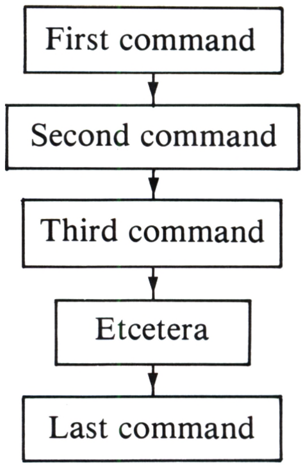
(c) 1Ø **PRINT** 727272 / 9
Answer 8Ø8Ø8.

Mousebender's Muddle

(a) (2 * 3) + (4 * 5) = 26
(b) 2 * (3 + 4) * 5 = 7Ø
(c) (2 * 3 + 4) * 5 = 5Ø
(d) 2 * (3 + 4 * 5) = 46

8 IFs but no buts

So far, the power of the computer has not really been exploited, except in doing arithmetic. This is because every program has just run through a single list of commands in order, and stopped:



This is kind of boring. Also, if you wanted a computer to carry out a million commands (which is not unusual) you would need to write a million lines of program. Not only would this be ridiculous—there wouldn't be room in memory to hold the program anyway. So there has to be a better way.

Two important ways to make a program more flexible are called *branching* and *looping*. In this chapter, we'll take a look at branching. For looping, see Chapter 10.

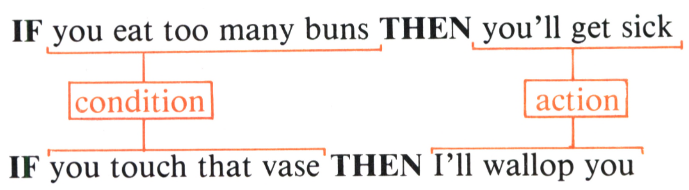
A program is said to *branch* if the computer follows different instructions, depending on whether some *condition* is:

TRUE

or

FALSE

We meet many such conditions in daily life:



IF you eat too many buns **THEN** you'll get sick

IF you touch that vase **THEN** I'll wallop you

In each, the specified *action* occurs only if the condition is TRUE.

It's much the same in BASIC. In fact, if you look back to Chapter 3, you'll find a program using the keywords:

IF **THEN**

to tell the computer what to do.

The next program asks you to toss a coin, input whether the result was heads or tails, and then prints different responses depending on what you tossed.

```
1Ø   PRINT "please toss a coin"
2Ø   PRINT "was it heads or tails?"
3Ø   INPUT side$
4Ø   IF side$ = "heads" THEN PRINT "I win!"
5Ø   IF side$ = "tails" THEN PRINT "you
     lose!"
```

In general, **IF** . . . **THEN** is used in the form:

IF condition **THEN** action

The *conditions* above are:

(in line 4Ø) side$ = "heads"

(in line 5Ø) side$ = "tails"

The *actions* are:

(in line 4Ø) **PRINT** "I win!"

(in line 5Ø) **PRINT** "you lose!"

Note that while the condition is a *statement*, to be tested for truth or falsehood, the *action* is a *command*, to be carried out.

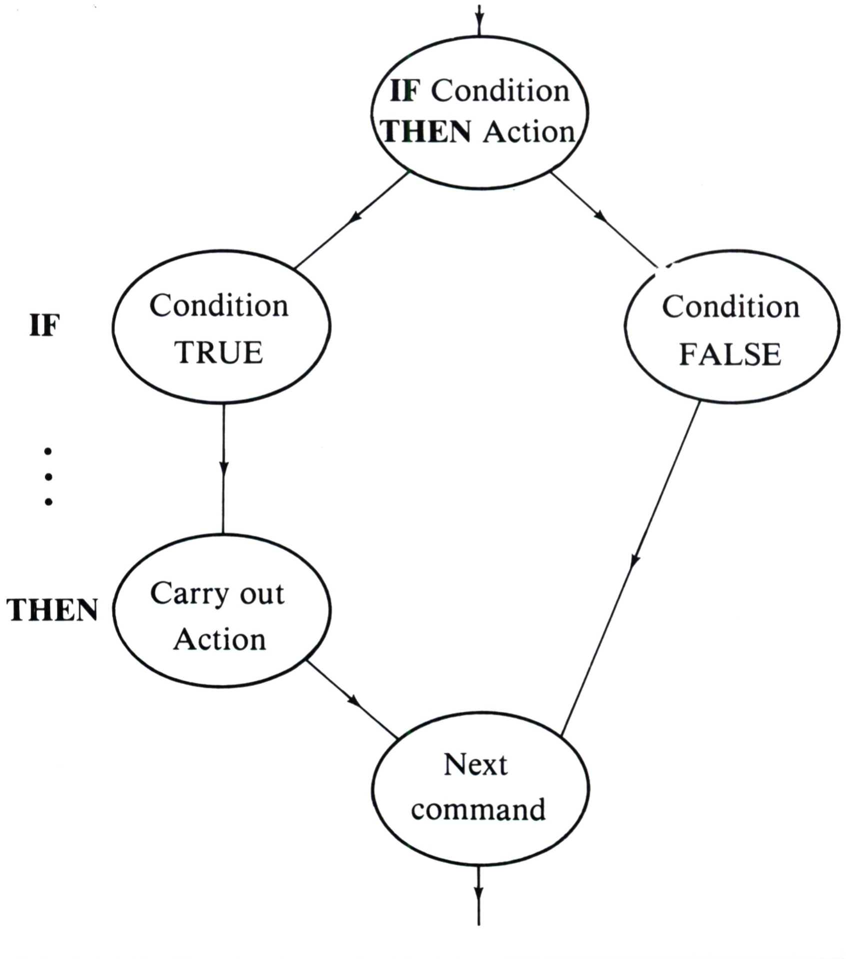
Here's how the computer treats an **IF** . . . **THEN** command.

IF the condition is TRUE
THEN the computer carries out the action.

IF the condition is FALSE,
THEN it doesn't!

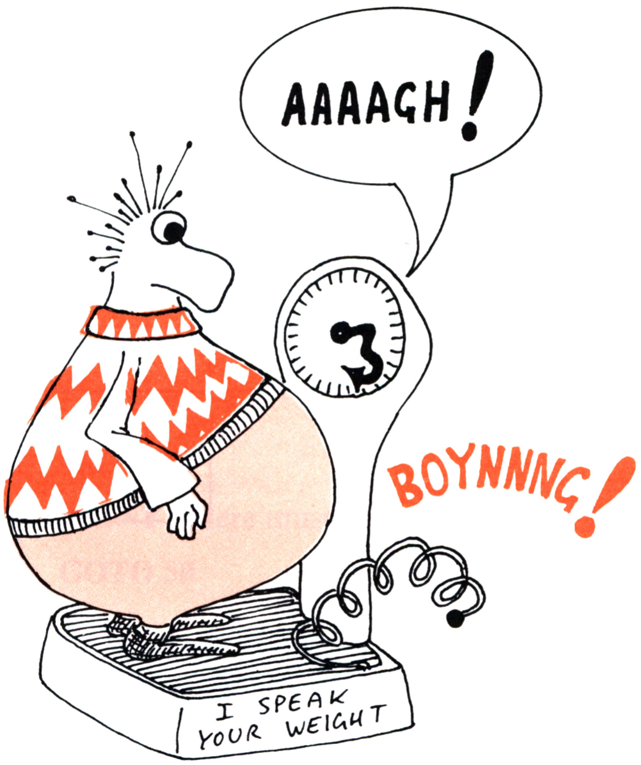
Either way, it goes on to the next program line.

Here's a diagram to show how the computer decides what to do. It is called a *flowchart*, because it lets you see the 'flow of command' through the program.



THE SPEAK-YOUR-WEIGHT MACHINE

Portly Pete the Plump Programmer needs to go on a diet. To help him get up the courage, he had decided to write a computer program. It will ask him to **INPUT** his weight. If this is less than 15Ø pounds, it will congratulate him on his success in becoming thin. If it is 15Ø pounds or more, it will say something rude.



Can you write a program to help Pete stay slim? You will need to know two new symbols:

<	means 'less than'
> =	means 'greater than or equal to'

So the *conditions* you'll need are:

IF weight < 15Ø

IF weight > = 15Ø

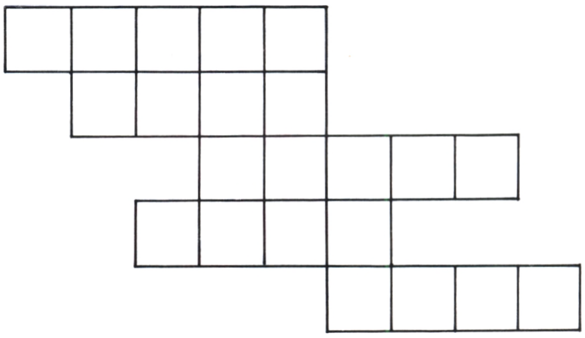
(You get '> =' by pressing keys > and = in turn.)

Size comparison symbols

=	Equal to
<	Less than
>	Greater than
< =	Less than or equal to
> =	Greater than or equal to
< >	Not equal to

COMPUCROSTIC

Fill in the *across* words and find the BASIC keywords in the *down* direction.



Place to stop

Sea creature

American State

Track down

Nasty bird!

JUMPS

I lied to you.

There is one circumstance in which an **IF** . . . **THEN** does *not* proceed to the next command at all.

First you need to know a new keyword,

GOTO

A command:

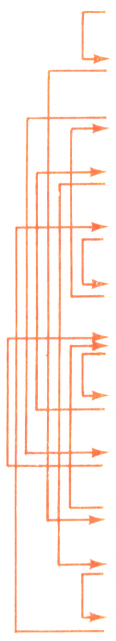
GOTO linenumber

moves the program to the command on that line. For example,

GOTO 71Ø

moves the program to line 71Ø, *no matter where it was before.*

Programming purists profess not to like **GOTO**. This is because it's very easy to get into bad habits, use too many **GOTO**s, and produce 'spaghetti' like this:



```
 1Ø   PRINT ″where do I goto?″
 2Ø   GOTO 1ØØ
 3Ø   GOTO 9Ø
 4Ø   GOTO 11Ø
 5Ø   PRINT ″easier way″
 6Ø   GOTO 3Ø
 7Ø   PRINT ″I'm lost!″
 8Ø   GOTO 4Ø
 9Ø   GOTO 7Ø
1ØØ   GOTO 7Ø
11Ø   PRINT ″there must be an″
12Ø   GOTO 5Ø
```

The lines at the side show where the jumps go: the reason for the word 'spaghetti' should be perfectly clear!

If you **RUN** this awful piece of programming you'll find that among other things it goes on forever. The lines are carried out in the order you'll find if you disentangle the spaghetti—that is:

$$\begin{tikzcd}
1Ø \arrow[r] & 2Ø \arrow[r] & 1ØØ \arrow[dr] & & & \\
 & & & 7Ø \arrow[dr] & & \\
 & & 9Ø \arrow[ur] & & 8Ø \arrow[dr] & \\
 & 3Ø \arrow[ur] & & & & 4Ø \arrow[d] \\
 & 6Ø \arrow[u] & & & & 11Ø \arrow[dl] \\
 & & 5Ø \arrow[ul] & & 12Ø \arrow[ll] &
\end{tikzcd}$$

and so it gets trapped in an *endless loop*, printing out the message:

```
I'm lost!
there must be an
easier way
```

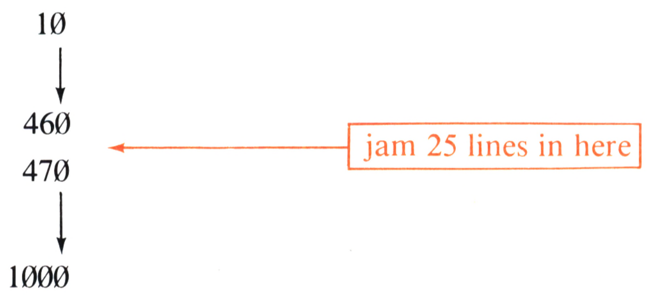
over and over again. Turn the computer off and then on to stop it, or (better!) press:

ESC

twice.

Let's agree right now that the purists have a point: there *is* a danger of getting in an awful muddle if you use too many **GOTO**s. And it would indeed be nice to improve BASIC so that **GOTO** was hardly ever needed. Granted that, it's still true that there are occasions when **GOTO** is very useful, if only to give you enough space to write a series of commands that you've forgotten to include somewhere.

For example, suppose you've written a hundred lines of program, from line 1Ø to line 1ØØØ. Suddenly you realize that you have missed out 25 lines, which *should* come in between lines 46Ø and 47Ø.



Well, there isn't enough room. You could change a few numbers to *make* room (for example change line 46Ø to 451 and 47Ø to 479). Or you could add:

```
465   GOTO 2ØØØ
```

to jump to somewhere with plenty of elbow-room; then write your 25 lines:

$$\begin{tikzcd}
2ØØØ \arrow[d] \\
224Ø
\end{tikzcd}$$

and then finish off with:

```
2250   GOTO 470
```

to link back to the correct place.

Effectively the program takes a big detour and then rejoins the main route, like this:

$$\begin{tikzcd}
1Ø \arrow[d] & \\
46Ø \arrow[d] & 2ØØØ \arrow[d] \\
465 \arrow[ur] & 224Ø \arrow[d] \\
47Ø \arrow[d] & 225Ø \arrow[l] \\
1ØØØ &
\end{tikzcd}$$

This technique is called a *patch*. It belongs to the general class of methods known as 'quick and dirty' in the Trade. That's fine: 'quick and dirty' may be dirty, but it's also *quick*, and there's no great virtue in spending hours and hours making something *look* pretty when it works just as well without.

The main use of **GOTO**, however, is in conjunction with **IF** . . . **THEN**. Suppose that the action to be performed if the **IF** condition is true, is a complicated one, taking many lines of program. Then the usual:

IF condition **THEN** action

just doesn't leave room to specify the action. Instead, we use:

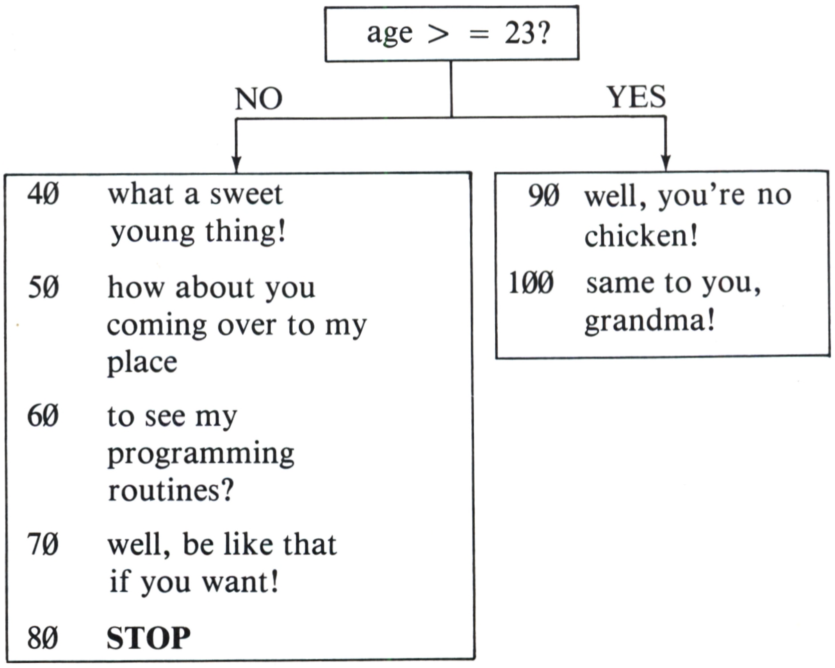
IF condition **THEN** $\overset{\boxed{\text{optional}}}{\textbf{GOTO}}$ linenumber

which jumps the program to *linenumber* if the condition is true. That gives us plenty of room to specify the action required; and then we can always jump back to wherever we want to be, using another **GOTO**.

Here is a program used by Licentious Lionel, the Lothario of Luton. This asks a lady her age. It responds differently, depending whether her age is 23 and over, or not. (Unlike the author of this book, Licentious Lionel is a Male Chauvinist Pig.)

```
 10   PRINT "how old are you, cutie-pie?"
 20   INPUT age      greater than or equal to—remember?
 30   IF age  > =   23 THEN GOTO 90
 40   PRINT "what a sweet young thing!"
 50   PRINT "how about you coming
      over to my place"
 60   PRINT "to see my programming
      routines?"
 70   PRINT "well, be like that if you
      want!"
 80   STOP
 90   PRINT "well, you're no chicken!"
100   PRINT "same to you, grandma!"
```

Here's how the 'flow of command' goes in the above program.


age > = 23?
NO
YES
40 what a sweet young thing!
50 how about you coming over to my place
60 to see my programming routines?
70 well, be like that if you want!
80 STOP
90 well, you're no chicken!
100 same to you, grandma!

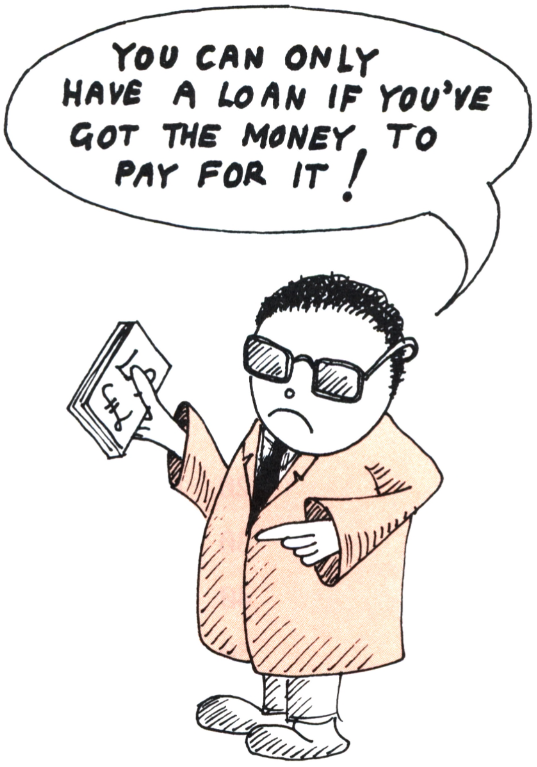
QUICKIE

There's a new command buried in Lionel's program. Find it and guess (it's not hard, I promise) what it does. Get rid of it, and find out why it was put there in the first place.

BRANCH BANKING

Carlton Q. Cashsnitcher, manager of the Lower Standards branch of the Gnatwest Bank, has just had an automatic cash-dispenser installed. It is computer-controlled. When a customer uses it, he is asked to input the password. The correct answer is 'toothpaste'. If the customer gets this right, the program then asks him how much cash he wants to take out. If he gets it wrong, the program informs him that he is in deep trouble.

Write a program to do this for Mr Cashsnitcher, so that all the shareholders in Gnatwest can rest easy in their beds at night.

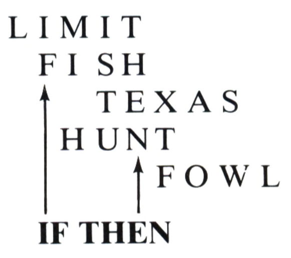

YOU CAN ONLY HAVE A LOAN IF YOU'VE GOT THE MONEY TO PAY FOR IT!

ANSWERS

The Speak-your-weight Machine

```
1Ø   PRINT "good morning, Pete"
2Ø   PRINT "and how heavy are we today?"
3Ø   INPUT weight
4Ø   IF weight < 15Ø THEN PRINT "careful
     walking over gratings you slim devil!"
5Ø   IF weight > = 15Ø THEN PRINT "watch
     out for harpoons!"
```

Compucrostic



Quickie

The new command is **STOP**. It stops the program going past that line. If you miss it out, then when age < 23, *both* messages get printed out. This could cramp Lionel's style . . .

Branch Banking

```
1Ø   PRINT "gnatwest autocash"
2Ø   PRINT "input password"
3Ø   INPUT pw$
4Ø   IF pw$ = "toothpaste" THEN GOTO 7Ø
5Ø   PRINT "if you don't leave this
     instant, I'll call the police!"
6Ø   STOP
```

```
 70  PRINT "how much cash do you want?"
 80  INPUT cash
 90  PRINT "take it, it's yours!"
100  PRINT "thank you for making a humble
     autocash machine very happy"
```

9 Bugs under the Rug

Programs seldom work properly the first time you run them. So don't be surprised! When the World was young and computers were the size of a room, insects used to crawl inside and cause havoc with program runs. This led to a term that is still in use:

> A *bug* is an error in a program. When you fix it, that's called *debugging*.

Program bugs, like their real-life counterparts, are hard to avoid. The best you can do is to find good methods for swatting them.

USER, HIM NO TALK GOOD

The grammatical structure of a computer language is called *syntax*. This may sound as if it refers to payments to the Inland Revenue for bad habits, but it comes from a Greek word and means 'systematic arrangement of parts'. For example, the syntax will be *wrong* if you:

- Spell a keyword incorrectly (**GNU** instead of **NEW**)
- Use non-existent commands (**PUTOUT** CAT. A variable named CAT is legal enough—but there's no command **PUTOUT** in BASIC)
- Combine statements incorrectly (**IF** X = 5 **THEN** "FRED" instead of **IF** X = 5 **THEN PRINT** "FRED")

- Use an illegal variable name (such as ØØ7 or list)
- Fail to match brackets in arithmetic (such as (1 – 3 * (5 + 2) where the final bracket has got lost)
- Write general nonsense (**LET** Z = 42//6 + 17..4)
- Invent your own—you soon will!

As mentioned already in Chapter 3, if you get the grammar wrong, the computer will stop and send you a message to tell you there is a syntax error, and what line it is on:

```
Syntax error in 666
```

It also **LIST**s the line ready for you to correct it.

It's a good idea to RUN the program after typing every 5 or 6 lines. That way you can pick up a lot of the syntax errors from the error messages, and fix them straight away. (There's nothing worse than having finished typing a 1ØØ-line program, and then spending the next hour dealing with Syntax error messages, one line at a time!) *Ignore any error messages other than syntax: these are probably due to the program being incomplete.* Deal with them only after all syntax is correct.

DEBUGS IS COMING!

Portable Pete the Perfunctory Programmer has been overwhelmed by syntax errors. Can you fix them for him?

(a) 1Ø **PRIMT** "HELLO"

(b) 2Ø **IMPUT** WEIGHT

```
(c)   3Ø   LET X = 2 * Y +
(d)   4Ø   LET X = 3 * (1 + 3 * (4 + 5)))
(e)   5Ø   LET 2 * Y = X
(f)   6Ø   IF PRINT "FRED" THEN LET B = 4
(g)   7Ø   IF AGE >3 THEM GOTO 7
(h)   8Ø   LET 2MUCH = 55
(i)   9Ø   ADD 6 TO 35
(j)  1ØØ   THE COMPUTER FACILITY WILL CLOSE
           IN TEN MINUTES
```

THE CASE OF THE CRASHED COMPUTER

Other errors pass the syntax test, but show up as soon as you run the program. These are called *runtime* errors. For example,

```
1Ø   INPUT a
2Ø   INPUT b
3Ø   LET c = 1ØØ / (2 * a - b)
4Ø   PRINT c
```

If you **RUN** this with the following inputs:

a = 3   b = 1

then it works fine, and prints out the answer 2Ø. Now try again using:

a = 2   b = 4

Yuk! What's 1.7Ø141E + 38 got to do with it? It's crashed! Well, almost...

A *crash* is when a program stops where it isn't supposed to.

The CPC464 doesn't actually *stop*...but it's clearly not happy.

Well, it worked OK the first time. So what's gone wrong?

* * * * * * *

"A small problem, Watson?"

"Holmes, thank God you've arrived. I'm in a terrible tangle. The computer simply *refuses* to obey my commands!"

Holmes cast his eye languidly over the monitor. "You have a bug, Watson."

"I am aware of that, Holmes. Regrettably, I have been unable to locate its exact whereabouts."

"But Watson, it's perfectly simple. Look at the *clues*, my dear fellow."

My mouth dropped open. "Clues? Clues? What clues?" I burbled hopelessly on like this for a considerable time. Holmes regarded me in silence until I ran out of breath.

"*First*, Watson, there is the remarkable incident of the first program-run."

"But Holmes, nothing went wrong on the first program-run!"

"*That* was the remarkable incident. And what do we deduce from it, Watson?"

I racked my brains, and a tiny light dawned. "My God, Holmes! It *couldn't have been a syntax error*!"

"Very good Watson." Holmes tactfully refrained from pointing out that there was no syntax error message. "In which case, Watson, we must ask ourselves what was *different* about the second run."

"Nothing, Holmes. The program was exactly the same both times."

"I know that, Watson. But . . ."

"My God! The *inputs*, Holmes! I *changed* the *inputs*!"

"We progress. Excellent, Watson. And where might the input value cause trouble?"

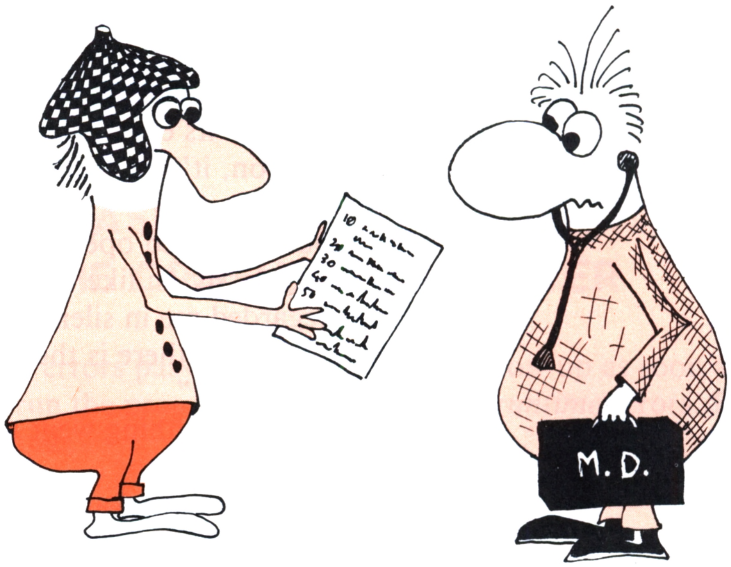
I cast my eyes over the program. The inputs were in lines 1Ø and 2Ø. I failed to see any opportunity for error there. But line 3Ø . . . arithmetic, I confess, is not my strongest point. "I suspect line 3Ø may contain the culprit, Holmes," I said slowly.

The great man nodded his assent. "If you care to study the error message, Watson, you will find that the cause will rapidly become apparent. For a medical man like yourself, a diagnosis of the fatal condition should not prove difficult."

My charitable and perspicacious reader may fare better than I. I confess, even this information did little for my addled brain. I stared blankly at the screen:

```
Division by zero
   1.7Ø141E+38
```

Division by zero? What Ø? And what on Earth is 1.7Ø141E+38?



DRY RUNNING

But Holmes's well of ideas had not yet run dry. He lit his pipe, leaned back in his chair, and puffed solemnly. I half expected to see him pick up his violin, but that was not to be.

"Has it occurred to you, Watson, to *dry-run* the program?"

"I beg your pardon, Holmes?"

"Work through the commands, using pencil and paper. Very instructive, Watson. Very instructive." Holmes took a leaf from his notebook and drew up a small tabulation:

Line no.	a	b	c

"Now, Watson: see that I have provided columns not only for the line number being executed, but also for the values of the three variables a, b and c that occur in the program. I now proceed to trace through the lines in order, paying due regard to the values assumed by those variables." And he wrote:

Line no.	a	b	c
1Ø	2	–	–
2Ø	2	4	–
3Ø	2	4	?

Holmes paused, and rubbed his chin in thought. "The question being: what value does the mysterious '?' take?" His pencil moved on to perform the calculation:

```
c = 1ØØ / (2 * a – b)
  = 1ØØ / (2 * 2 – 4)
  = 1ØØ / Ø
```

"As I imagined," he sighed. "There it is, Watson. Do you see?"

"Well, Holmes . . . arithmetic has never been one of my—"

"Suppose I gave you a bag of apples, Watson."

"Holmes, that would be very kind of you, but I utterly fail to see how—"

"And suppose there were exactly zero apples in it."

"*Zero*? Whatever—"

"How many such bags, Watson, must I give you for you to receive a hundred apples?"

I counted desperately on my fingers. A hundred bags? No, that would still give me zero apples. A thousand? A million? A googolplex? "It seems, Holmes, that no quantity of bags would ever yield even a single apple. Let alone a hundred."

"Precisely, Watson! So the arithmetical statement 1ØØ/Ø has no meaning!"

And then I saw. "So the computer cannot produce a sensible answer for the value of c, Holmes!"

"Exactly. It does the best it can. The largest number it can store is 1.7Ø141E + 38, in Scientific notation."

"And what is *that*, Holmes?"

"About 170 million, million, million, million, million, million."

"That's a lot of apples, Holmes! I'd get stomach ache if I ate all of *them*!" Holmes sighed, and reached for his violin.

The divide-by-zero gremlin can strike at any moment. Beware!

CRICKET BUG

The Inevitable Duck Cricket Club is the proud owner of a war surplus IBM computer. Every year this is used to work out the team's bowling averages. The program uses four variables:

rp	=	total runs scored off bowler at the end of previous season
rt	=	total runs scored off bowler to date
wp	=	total wickets taken by bowler at the end of previous season
wt	=	total wickets taken by bowler to date

It subtracts the totals to find the number of wickets taken in the current season (wt – wp) and the number of runs scored off the bowler in the current season (rt – rp), and then divides to get the number of runs per wicket. Like this:

```
10  INPUT rp
20  INPUT rt
30  INPUT wp
40  INPUT wt
50  LET average = (rt – rp)/(wt – wp)
60  PRINT "average is  ; average
```

At this stage of the season the bowlers' figures are:

Bowler's name	rp	rt	wp	wt
Will Bashitt	2001	2041	72	76
Tom Bowler	9	97	1	12
Dick Laird	403	625	48	49
L.B.Doubleyou	962	1531	17	17
Kit Bagge	122	365	26	35
Wally Pitt	743	967	73	81

One of the bowlers will make the program go wrong. Who, and why? Try the inputs in turn. Write a dry-run table to find the bug.

ANSWERS

Debugs is Coming!

(a) **PRINT**, not **PRIMT**.

(b) **INPUT**, not **IMPUT**.

(c) Delete final + .

(d) One final bracket too many.

(e) **LET** X = 2 * Y is the only legal syntax.

(f) **PRINT** "FRED" isn't a condition.

(g) **THEN**, not **THEM**.

(h) 2MUCH starts with a number and is illegal on the CPC464.

(i) **ADD** is not a BASIC keyword.

(j) Nonsense in BASIC.

Cricket Bug

L.B.Doubleyou will cause a division-by-zero error because wp – wt is Ø for his figures.

10 FOR, 3, TO, 1, NEXT!

We noticed in Chapter 8 that the **GOTO** command can produce an endless loop, where the program does the same things over and over again forever. For example,

```
10  PRINT "help!"
20  PRINT "get me out of here!"
30  PRINT "I'm stuck!"
40  GOTO 10
```

This shows that the computer can be made to carry out a large number of commands with a very short program. It would be far more useful, though, if we could make the computer go round a loop a definite number of times, and then *exit* the loop, ready to obey other commands. This would let us tell the computer to carry out some repetitive task, and then do things with the result. People find repetitive tasks boring; computers like nothing better!

The keywords:

FOR **TO** **NEXT**

are designed to do just that. They are used in the general form:

```
FOR variable = startnumber TO finishnumber
. . .
. . .   program lines
. . .
NEXT variable
```

which is known as a **FOR** . . . **NEXT** *loop*.

Suppose you want the computer to work out a seven times table. It has to find:

1 * 7   2 * 7   3 * 7   . . .   12 * 7

in turn.

Here's a *bad* answer. 'Slow and dirty' instead of 'quick and dirty'.

```
 1Ø   PRINT "1 * 7 = "; 1 * 7
 2Ø   PRINT "2 * 7 = "; 2 * 7
 3Ø   PRINT "3 * 7 = "; 3 * 7
 4Ø   PRINT "4 * 7 = "; 4 * 7
 5Ø   PRINT "5 * 7 = "; 5 * 7
 6Ø   PRINT "6 * 7 = "; 6 * 7
 7Ø   PRINT "7 * 7 = "; 7 * 7
 8Ø   PRINT "8 * 7 = "; 8 * 7
 9Ø   PRINT "9 * 7 = "; 9 * 7
1ØØ   PRINT "1Ø * 7 = "; 1Ø * 7
11Ø   PRINT "11 * 7 = "; 11 * 7
12Ø   PRINT "12 * 7 = "; 12 * 7
```

Pheeeeeewwwww!

It's bad, because it has lots of lines that do *almost* the same thing. And suppose you wanted the table to continue up to 1ØØØ*7? Or 1ØØØØØØ*7? Yuk!

Fortunately, there's a much more efficient way to proceed. The idea is to set up a variable, say n, running from 1 to 12. We work out

n * 7

for n = 1, 2, 3, . . ., 12, by using a loop. Then we *stop* the loop.

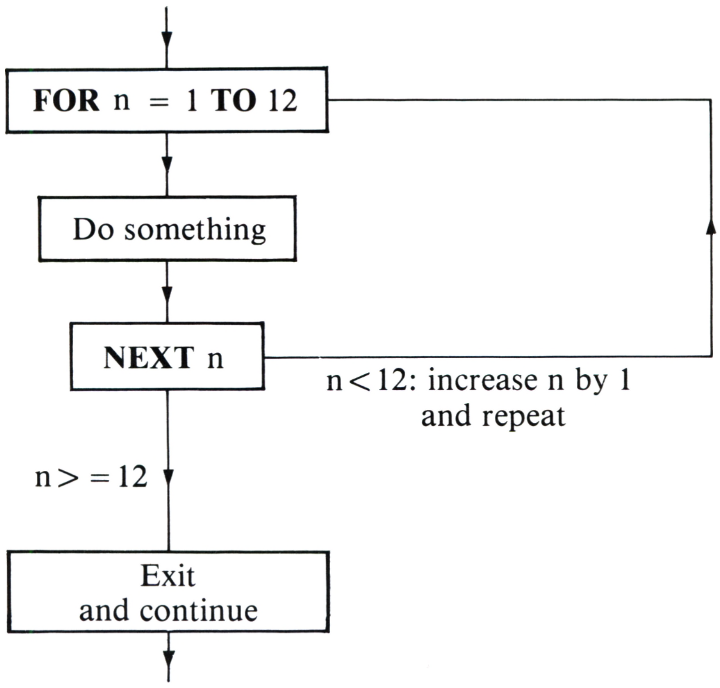
This is the *whole* program:

```
               start  finish
                 |      |
                 v      v
1Ø   FOR n  =    1  TO  12
2Ø   PRINT n;" * 7 = ";n * 7
3Ø   NEXT n
```

Let's see how this works. I'll follow through the line numbers being carried out, with comments on what's happening.

Line number	Commentary	Value of n
1Ø	The **FOR** n = 1 **TO** 12 command starts by setting n to the value 1 (the startnumber).	1
2Ø	Work out 7 * n and print.	1
3Ø	The **NEXT** tells us to test the value of n to see whether the loop has finished. If n is greater than or equal to the finishnumber (here 12) we exit the loop and go on to the command immediately after the **NEXT** command (if there is one; if not, stop). This is not the case, so we go back to the start of the loop, the **FOR** command.	
1Ø	Now increase n to 2	2
2Ø	Work out 2 * 7 and print.	2
3Ø	Test for end: no, so back to the **FOR**.	2
1Ø	Increase n to 3	3
2Ø	Work out 3 * 7 and print.	3
	. . .	
	. . .	
	. . .	
2Ø	Work out 12 * 7 and print.	12
3Ø	Test for finish. This time n = 12, which is the finishnumber. So we exit the loop. That is, instead of going back yet again to the **FOR** command, the program carries on to the line after **NEXT**. Here there isn't one, so it stops.	12

We can draw the loop as a flowchart:



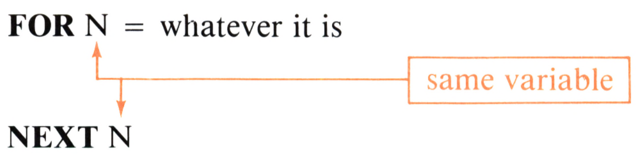
Now, I admit that a seven times table isn't very exciting. But a table up to 1000000*7 would be more impressive, and just as easy:

```
10   FOR n = 1 TO 1000000
20   PRINT n;" * 7 = ";n * 7
30   NEXT n
```

You may care to try this. It *will* take about a day or so for the computer to carry it out! Switch off when you get fed up.

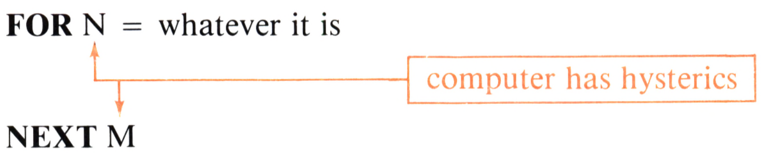
The variable n in the **FOR** and **NEXT** commands is called the *loop counter*. It is important to use the same variable in both commands:

> The loop-counting variable in the **FOR** and **NEXT** commands must match up.



```
FOR N = whatever it is

NEXT N
```

You may *not* use something like this:



```
FOR N = whatever it is

NEXT M
```

The CPC464 can't cope with **NEXT** M if it hasn't already found **FOR** M, so gives:

Unexpected NEXT...

On many computers, loops with non-matching variables will *not* lead to error messages or crashes. BUT the computer will not do anything remotely like what you intended.

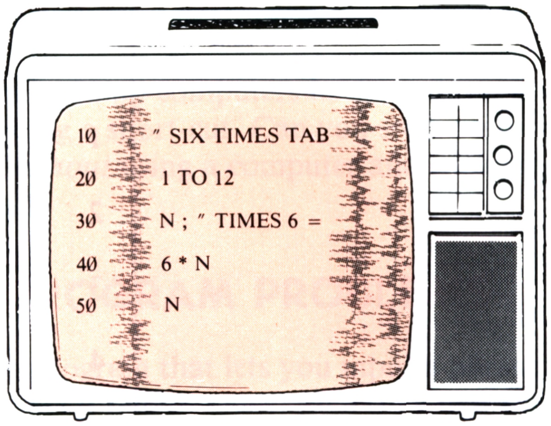
On the CPC464 the variable in the **NEXT** command may be omitted altogether. The computer will automatically match it to its **FOR** command.

```
FOR N = 1 TO 12
NEXT
```

But you *must* put it in the **FOR** command at the start of the loop.

TELLYGRAM

Procedural Pete the Pedestrian Programmer is trying to download software (receive a program) from the Micronit 84 network system, but his telly's on the blink and a lot of the program has been obliterated. Can you work out what it should be?



CUMULATIVE ADDITION

If you want to add up a whole lot of numbers that vary regularly, you can use a loop. You will need a variable to act as a 'running total'. For example, we can work out:

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 1Ø

using this loop:

```
1Ø   LET sum = Ø
2Ø   FOR n = 1 TO 1Ø
3Ø   LET sum = sum + n
4Ø   NEXT n
5Ø   PRINT sum
```

Note that we set up sum, the running total, to an initial value *zero*. And this is done *before* the loop starts. This is important: it has to be given a starting value somewhere; and if you do it inside the loop, you keep resetting it to the start, which is silly. Setting up a start value for a variable is called *initialization*.

Initialize before you loop.

Let's just see how the program does the addition. Here are the variables involved, and how they change.

n	sum takes new value
	Ø at start
1	Ø + 1 (= 1)
2	1 + 2 (= 3)
3	3 + 3 (= 6)
4	6 + 4 (= 1Ø)
5	1Ø + 5 (= 15)
6	15 + 6 (= 21)
7	21 + 7 (= 28)
8	28 + 8 (= 36)
9	36 + 9 (= 45)
1Ø	45 + 1Ø (= 55)

YOU TOO CAN BE A GENIUS!

The great mathematician Carl Gauss showed his genius at a very early age. When he first went to school, his teacher set the class the problem of adding all the numbers between 1 and 1ØØ, that is,

1 + 2 + 3 + 4 + 5 + . . . + 99 + 1ØØ

Gauss took one look at this, and immediately wrote a number on his slate. 'There it is', he said, putting it on the teacher's desk. The teacher didn't believe he could have got the answer so quickly. But at the end of the lesson, when the other children had handed in their answers . . . only Gauss got it right!

Write a program that uses a **FOR** . . . **NEXT** loop to find the total that Gauss worked out in his head.

STING IN THE TAIL

In fact, there were no computers in those days, and Gauss did it by finding a short-cut. Can you see how to work out the answer *without* using a computer and *without* doing lots of sums?

THREE PROGRAM PROBLEMS

(a) Write a program that lets you input a phrase, and then prints it out 2Ø times, each on a new line. Try it with the phrase:

I MUST NOT CHEW BUBBLEGUM IN CLASS

See? Computers can be useful in practical situations! 'Computer, write a thousand lines . . .'

(b) *Without* running this program, work out what it does. Then run it and see if you're right.

```
1Ø   FOR n = 1 TO 1Ø
2Ø   PRINT n
3Ø   IF n * n = 25 THEN PRINT "bingo!"
4Ø   NEXT n
```

(c) Write a program that uses a loop to find all whole numbers n between 1 and 1ØØ for which:

n * n * n + 6875 * n = 15Ø * n * n + 9375Ø

What numbers n solve this?

ANSWERS

Tellygram

```
1Ø   PRINT "SIX TIMES TABLE"
2Ø   FOR N = 1 TO 12
3Ø   PRINT N; " SPACE TIMES SPACE 6 =";
4Ø   PRINT 6 * N
5Ø   NEXT N
```

You too can be a Genius!

```
10  LET sum = 0
20  FOR n = 1 TO 100
30  LET sum = sum + n
40  NEXT n
50  PRINT "total is ";sum
```

The answer is 5050.

Sting in the Tail

Gauss paired off the numbers from both ends, like this:

```
1 + 100 = 101
2 +  99 = 101
3 +  98 = 101
.     .
.     .
.     .
50 + 51 = 101
```

So there are 50 pairs, each adding up to 101, and the total is 50 * 101 = 5050.

Moral: Think first, compute later!

Three Program Problems

(a)

```
10  PRINT "input phrase"
20  INPUT phrase$
30  FOR n = 1 TO 20
40  PRINT phrase$
50  NEXT n
```

(b) It prints out the numbers from 1 to 10 in order. *Further*, when it finds a number n such that n * n = 25, it shouts "bingo!". This occurs at n = 5. (In other words, the computer is solving the equation n * n = 25 by systematic trial-and-error.)

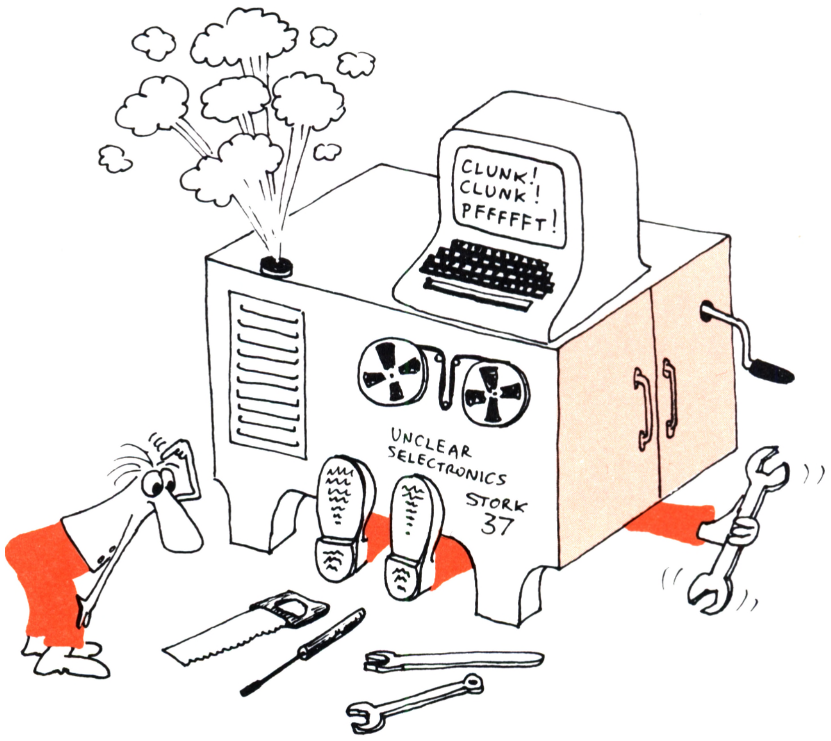
(c)

```
10  FOR n = 1 TO 100
20  IF n * n * n + 6875 * n = 150 * n * n + 93750
    THEN PRINT n
30  NEXT n
```

The numbers n that get printed are 25, 50, 75.

11 Inside the Computer's Brain

This chapter isn't about programming. It's about the way your computer actually *works*. You don't *have* to read this to be a good BASIC programmer, but I thought you might find it useful anyway. A lot of things in computing make more sense if you have some idea of what is really going on inside the machine.

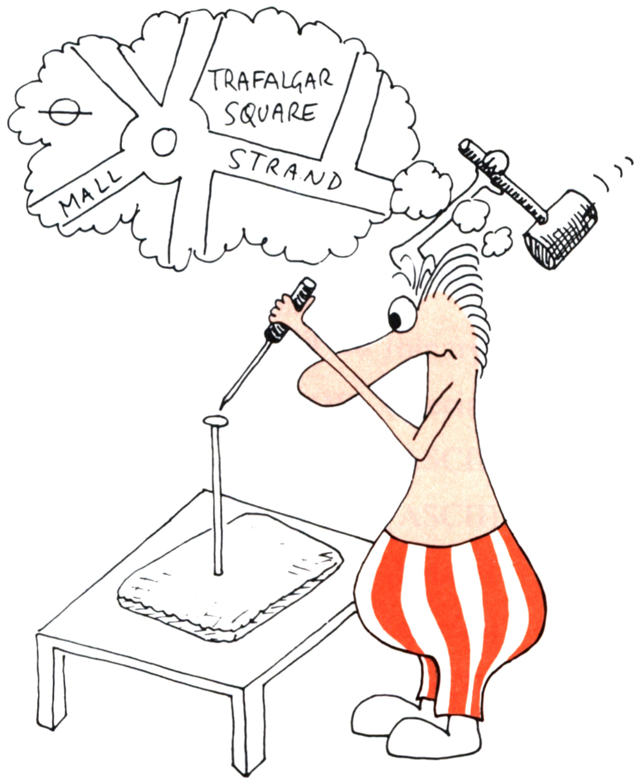


ZEROS AND ONES

Deep down inside, computers work by shuffling pulses of electricity around. The timing of these pulses is controlled by a *clock*—a circuit that 'ticks' electronically at a very high speed. Four million ticks per second is typical of home computers. By contrast a big computer may work at a thousand million ticks per second.

Each tick, the CPU (Central Processing Unit, remember? If not, read Chapter 1 again) looks to see whether there is a pulse of electricity or not. The pulses form code signals, telling it what to do. It then sends similar code signals to other parts of the computer.

In the early days, the circuits that controlled these pulses were very slow and clumsy, and used big vacuum tubes. Nowadays they are constructed photographically on the surface of a tiny chip of silicon. A chip the size of a penny can contain a quarter of a million circuit components! The complexity of a chip is similar to drawing a map of London on the head of a pin. Usually there are a number of chips, each intended to perform a specific task; and these chips shuttle messages around to tell each other what to do next.



The computer makes only a very coarse distinction between:

- No electricity
- Lots of electricity

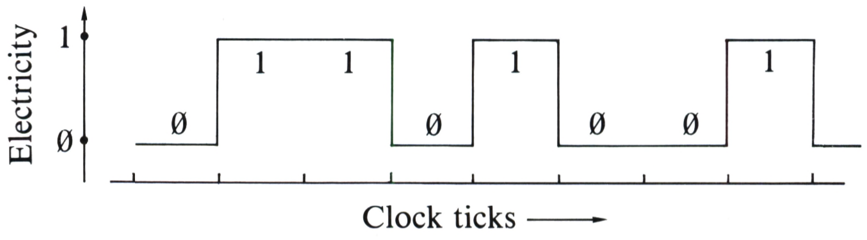
This is done to minimize errors caused by 'noise' in the circuits.

It is convenient to use the numbers Ø and 1 to represent the absence or presence of a pulse.

Ø means 'no electrical pulse'
1 means 'one electrical pulse'

Each Ø or 1 is called a *bit* of information.

Here is a typical message.



It corresponds to the sequence of bits:

Ø 1 1 Ø 1 Ø Ø 1

and has eight bits in it.

Computers use certain special sequences of bits to represent numbers, letters, or instructions. For example, letters are usually represented in ASCII code (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) which starts off like this:

A	Ø 1 Ø Ø Ø Ø Ø 1
B	Ø 1 Ø Ø Ø Ø 1 Ø
C	Ø 1 Ø Ø Ø Ø 1 1
D	Ø 1 Ø Ø Ø 1 Ø Ø
E	Ø 1 Ø Ø Ø 1 Ø 1
. . .	
Z	Ø 1 Ø 1 1 Ø 1 Ø

Numbers are dealt with in a different code, called *binary*. In its simplest form this goes:

Ø	Ø Ø Ø Ø Ø Ø Ø Ø
1	Ø Ø Ø Ø Ø Ø Ø 1
2	Ø Ø Ø Ø Ø Ø 1 Ø
3	Ø Ø Ø Ø Ø Ø 1 1
4	Ø Ø Ø Ø Ø 1 Ø Ø
. . .	
255	1 1 1 1 1 1 1 1

And commands are stored in yet another code, for example:

THEN	1 1 1 Ø 1 Ø 1 1
TO	1 1 1 Ø 1 1 Ø Ø
STEP	1 1 1 Ø Ø 1 1 Ø
FOR	1 Ø Ø 1 1 1 1 Ø
LIST	1 Ø 1 Ø Ø 1 1 1

etc.

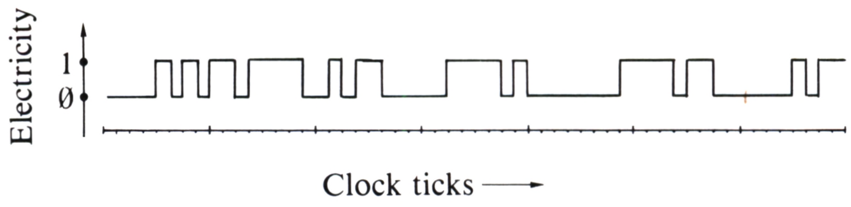
When programming in BASIC you don't need to *know* any of these codes. I'll explain why in a minute. But it's worth realizing that a simple line of BASIC such as:

```
1Ø  FOR X = 1 TO 5
```

is handled by the computer in a form more like this:

Ø Ø Ø Ø 1 Ø 1 Ø	1Ø	(binary)
1 Ø Ø 1 1 1 1 Ø	**FOR**	(special)
Ø 1 Ø 1 1 Ø Ø Ø	X	(ASCII)
Ø Ø 1 1 1 1 Ø 1	=	(ASCII)
Ø Ø Ø Ø Ø Ø Ø 1	1	(binary)
1 1 1 Ø 1 1 Ø Ø	**TO**	(special)
Ø Ø Ø Ø Ø 1 Ø 1	5	(ASCII)

(In fact it's even more complicated, but the general idea is right.) So the BASIC line becomes a long series of electrical pulses, like this:



Notice that the codes all come in 8-bit blocks. This is because almost all home computers use chips that handle bits in groups of 8 at a time: they are *8-bit machines*. (Commercial computers are often 16- or 32-bit machines; scientific ones often more.) A series of 8 bits is called a *byte*.

PULSE CODE PUZZLE

What would the phrase BAD CAT look like:

(a) In ASCII code?
(b) In electrical pulses?

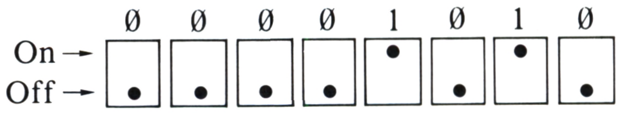
(ASCII for 'T' is Ø1Ø1Ø1ØØ, and ASCII for SPACE is ØØ1ØØØØØ.)

THE MEMORY

Pulses are used to shuttle information around. A similar system is used to *store* information. The way to think of this is to pretend that the memory consists of row upon row of little switches, with:

1 representing an ON switch
Ø representing an OFF switch

like this:



The internal memory of the computer comes in two types:

1. RAM (**R**andom **A**ccess **M**emory). This can be changed by the programmer, and its contents are lost when the power is turned off.
2. ROM (**R**ead **O**nly **M**emory). This holds in permanent form all the instructions that tell the CPU what to do. This does *not* get lost when the power is turned off; and can't be altered by the programmer.

MEMORY SIZE

The amount of memory is usually measured in *kilobytes*. One kilobyte = 1024 bytes; that is 8192 bits. This may sound a lot, but since it takes one byte to hold a single character (ASCII code uses 8 bits per letter) you could only get about half a page of this book into one kilobyte of RAM.

The word 'kilobyte' is abbreviated to the letter K, so for instance 7K would mean 7 kilobytes. Typical memories in home computers nowadays are:

16K or 32K of ROM
16K, 32K, 48K, or 64K of RAM

These would hold about 8, 16, 24, or 32 pages of this book. This is more than enough for most programs.
The Amstrad CPC464 has 64K of memory, which is more than you'll be likely to need... *unless* you want to handle lots of data. And then you really need an outside memory, preferably as *disc drive*, which is quick to get at and stores an awful lot of information cheaply.

For example, suppose you want to catalogue a library of 1000 books, by author and title. You'd need, typically, about 60 bytes per book:

20 bytes for author	40 bytes for title
....	....
A A MILNE	THE HOUSE AT POOH CORNER
DAVID HARWOOD	GAMES TO PLAY ON YOUR AMSTRAD CPC464
LEWIS CARROLL	ALICE'S ADVENTURES IN WONDERLAND AND THR

(Woops! OUGH THE LOOKING-GLASS has got the chop, even with 40 bytes.) That gives a total of 60,000 bytes, so you might *just* squeeze it into 64K. And there would be precious little room for any programs to handle the data—for instance, to arrange items in alphabetical order.

I recently read in an American newspaper an excellent rule for deciding how much computer memory to buy. Add 14 to your salary. That is, if you earn 50K ($50,000) then buy 64K. If you earn 34K then buy 48K. The point it was making was that making a fuss about needing a large memory is often just snobbery.

THE OPERATING SYSTEM

The instructions in ROM are known as the *Operating System* of the computer. They tell the CPU how to go about doing everyday tasks like:

- Read the keyboard to see which key has been pressed
- Print a character to the TV screen
- Decide what a given line of BASIC means
- Send signals to a cassette recorder or disc drive to save a program in permanent form

and a thousand and one other jobs.

The CPU is capable of handling all of these, but only if they are broken down into a series of *very* simple steps.

MACHINE CODE

Each such step has its own representation in yet *another* code (oh, yes: there are lots and lots of codes, depending on what's most effective for the job being done!) called *Machine Code*.

For example a simple line of BASIC like:

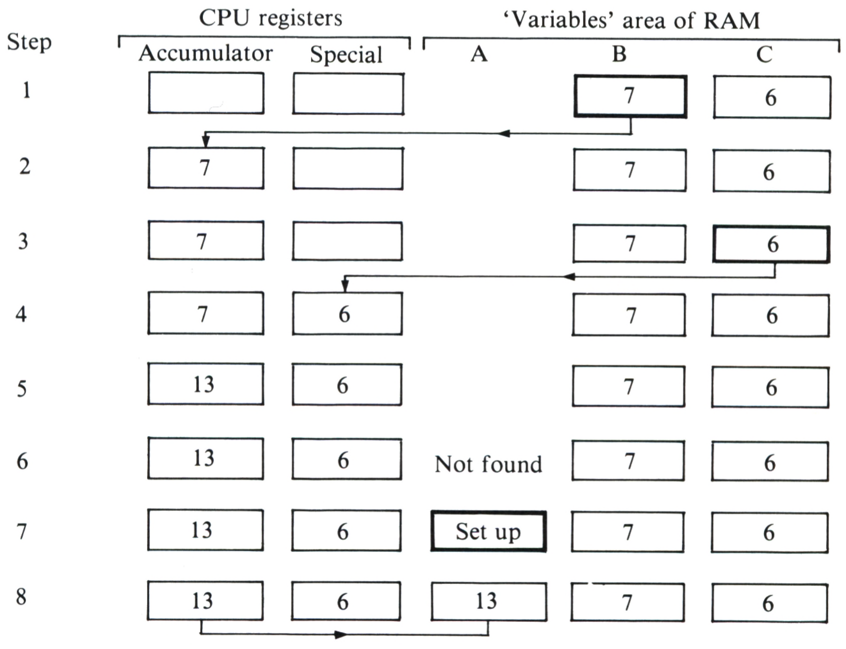
```
20  LET A = B + C
```

is handled by the CPU like this:

1. Search through the 'variables' area of RAM to find a variable labelled 'B'.
2. Take the number stored in B and put it into a special part of the CPU called an *accumulator register*.
3. Search through the 'variables' area of RAM for a variable labelled 'C'.
4. Take the number stored in C and transfer it to another special register in the CPU.
5. Add the number in the special register to the one in the accumulator register and store the result in the accumulator register.
6. Search through the 'variables' area of RAM to find a variable labelled 'A'.
7. If there isn't one, set one up at the end of the 'variables' area.
8. Take the contents of the accumulator register of the CPU and place that in the variable 'A'.

And in fact, each of these steps would be broken down into even simpler tasks. And the whole lot would then be coded into lots of Øs and 1s.

For instance, suppose 'B' holds 7 (which is ØØØØØ111 in binary, but to keep it simple let me use decimal) and 'C' holds 6. Then the sequence of steps has this effect:



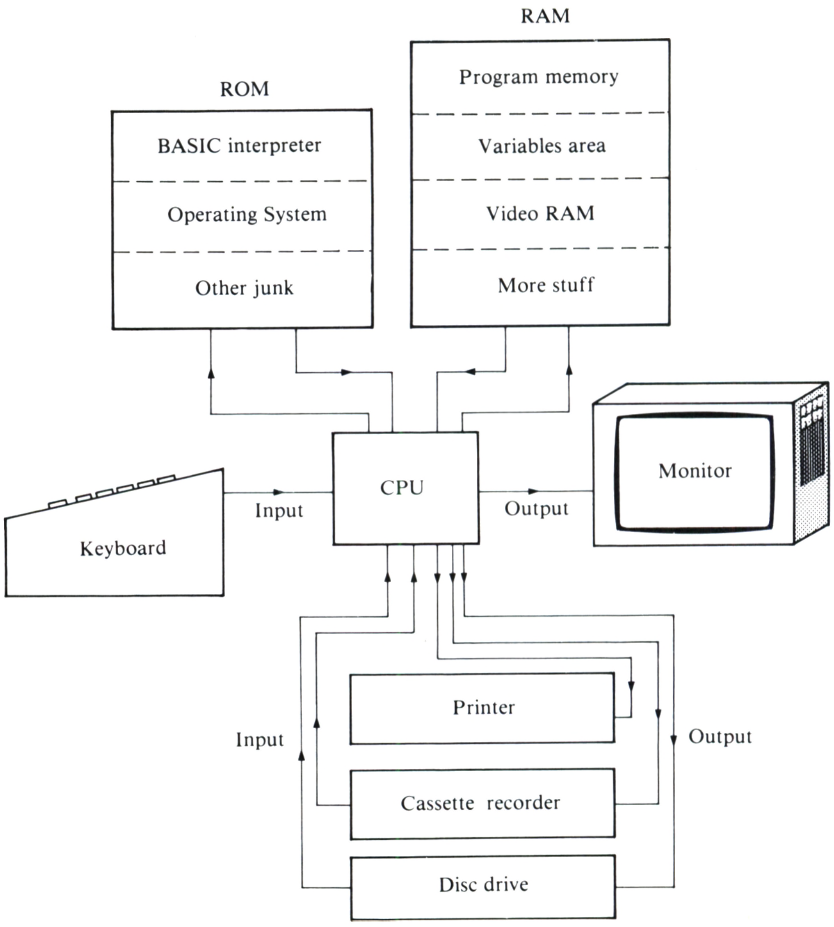
STRUCTURE OF THE COMPUTER

Much of the ROM consists of a BASIC *interpreter* which works out what the computer must do to carry out any particular BASIC command. The rest handles cassettes and discs, generates character shapes, keeps an eye on the keyboard, and so forth.

The RAM is divided into definite areas with particular tasks. For example the:

Program area	stores the program
Variables area	stores values of variables
Video RAM	stores information controlling the screen display

So now our picture of the computer's structure has got more detailed and of course there's more to it even than that.



HOW A PROGRAM IS RUN

Here, in simplified form, is what goes on when you type in and **RUN** a program.

1. The Operating System tells the CPU to scan the keyboard. As each key is pressed, the result is sent to the Program Area of RAM for storage. Then the CPU looks for the next key.
2. When you give the command **RUN**, the Operating System tells the CPU to execute the program.
3. The CPU looks through RAM to find the first program line.
4. It looks at the ROM to find the BASIC interpreter, in order to find out two things:

   (a) Which line will come *next*. (Note that **IF**, **FOR**. . .**NEXT**, **GOTO** and so on can make this different from the next line in numerical order.)
   (b) How to convert the line into Machine Code, to carry it out.

5. It then carries out the series of Machine Code instructions for that line.
6. It uses the result of 4(a) to look up in RAM the next line of the BASIC program.
7. And so on until it gets to the end.

The point to realize is that the CPU spends an awful lot of its time looking either through RAM or ROM to find the information it needs. It's like a person who can only do a job by looking up each step in the handbook.

Note also that each line of BASIC is converted to Machine Code by the interpreter *every time it is executed*. It may have occurred before, for instance in a loop. *The CPU does not realize this*: it asks the interpreter to deal with it *all over again*.

This problem with an 'unintelligent' interpreter is the major reason why BASIC is far slower than Machine Code. But BASIC is so much simpler to write! One solution is a more sophisticated technique called a *compiler*: this changes the *whole program* into Machine Code *first*, before it is carried out. But compilers are harder to produce.

You *can* learn to program directly in Machine Code, to improve the speed, but it's tough going. (See the Shiva catalogue for Machine Code books for ZX81, Spectrum, Dragon, BBC, Commodore, Electron, VIC 20, Atmos and Oric-1, etc.)

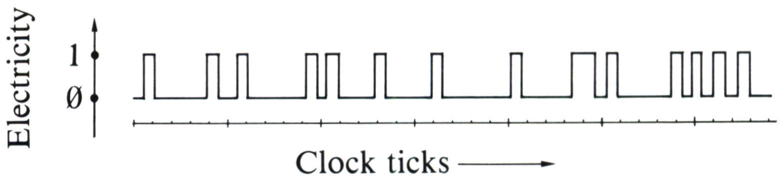
No, that's not the end of the story. Computers are very complicated machines indeed, and we've barely scratched the surface. There are dozens and dozens of BASIC commands I haven't had room to explain yet; and that's just the tip of a gigantic iceberg. And the whole subject changes dramatically every few years. But at least we've got started. And this chapter has gone on too long already, so I'll stop here.

ANSWERS

Pulse Code Puzzle

(a) Ø1ØØØØ1Ø Ø1ØØØØØ1 Ø1ØØØ1ØØ ØØ1ØØØØØ
Ø1ØØØØ11 Ø1ØØØØØ1 Ø1Ø1Ø1ØØ

(b)



Glossary

Accumulator register The place where numbers are added up.

ASCII code American Standard Code for Information Interchange. Represents symbols as sequences of Øs and 1s in a standard way.

BASIC Beginner's All-purpose Symbolic Instruction Code. Computer language widely used on home computers.

Binary A method of writing numbers using only the digits Ø or 1.

Bit A code digit Ø or 1.

Branch A place where a program can carry out one of several different options.

Bug A mistake in a program.

Byte A sequence of eight bits, such as 1Ø11ØØ1Ø. Most home computers use one byte to store each item of information.

Central Processing Unit The 'brains' of the computer: the part that controls what it does. Abbreviated to CPU. Usually resides on a single chip.

Character A symbol that the computer can print on the TV screen, such as X, 7, %, and so on.

Chip An electronic circuit built on the surface of a tiny piece of silicon by photographic etching methods and vapour-deposition. Lots of components crammed into a very small space.

Clock   An electronic circuit that sets the timing of all electrical signals inside the computer.

Command   A single BASIC instruction, such as **PRINT** "FRED".

Compiler   A fancy program that takes any program written in BASIC and converts it into instructions that the computer's CPU can understand.

Condition   A statement that is either *true* or *false*, used to cause a branch in a program.

Crash   When a program stops, or gives an error message, where it isn't intended to. Sign of bugs!

Data   Items of information.

Debugging   Making a program work properly.

Decimal point   Written as a full stop '.' in BASIC.

Disc drive   A device that stores lots of information on a magnetic disc.

Division sign   In BASIC, this is '/' not '÷'.

Dollar sign ($)   Used in BASIC to tell the computer to work with strings, that is, sequences of characters.

Dry-running   A method for debugging a program by working through parts of it by hand.

Endless loop   A state the computer sometimes gets into, where it carries out the same commands over again forever. Due to faulty programming.

Enter   To type a program from the keyboard.

Error message   Message printed by the computer if it spots a mistake.

Execute   To carry out the instructions in a program.

Exit   To leave a program loop.

Flowchart   A diagram that uses boxes linked by arrows to show what the program will do. Overrated as a teaching aid, but sometimes useful.

Function   A rule that associates with each variable some particular value.

Information   Facts, figures, anything that can be put into symbols.

Initialize   Set up values of variables at the start of a program or a part of a program.

Input   To feed information into the computer.

Interpreter   Program permanently stored in memory to convert BASIC into instructions that the CPU can understand, *one line at a time*.

Jump   Instruction to the program to move to a line that is not the next in numerical order.

Keyboard   Used by the operator to type instructions into the computer.

Keyword   Special BASIC word such as **PRINT**, **NEW**, **RUN**, **LIST**.

Kilobyte   Unit of memory size equal to 1024 bytes or 8192 bits. Roughly equivalent to half a page of text.

Line number   Number written in front of each BASIC command, used by the interpreter for reference purposes.

Loop   Part of a program that works through the same sequence of commands several times, usually changing some of the variables as it does so.

Loop counter   Special variable used to ensure that the loop stops after the required number of stages.

Machine Code   Computer language that the CPU is designed to work with.

Manual   Instruction book for a computer.

Memory   Circuitry that lets the computer store data.

Monitor   High-quality TV set used for display.

Multiplication sign   In BASIC this is '*', not '×'.

Name (of a variable)   Codename given to a variable so that the computer can tell which one is required. Examples are B, STRENGTH, C$.

Numeric variable   A variable whose values are numbers.

Operating System Program permanently stored in memory which tells the CPU how to carry out all of the routine tasks needed to make the computer work.

Output To get information out of the computer.

Patch Addition to a program to cure an error, usually as an afterthought, by making the program do a detour.

Power supply Device that supplies electricity to the computer at the correct voltage, etc. On the CPC464 it's built into the monitor or MP1 modulator.

Printer Mechanical device that produces typed output on sheets of paper.

Program List of instructions for the computer to carry out.

Program line In BASIC, the program is divided into numbered lines.

Program memory The part of memory reserved for storing programs.

Prompt Message accompanying an input command to remind the user what is required.

Pulse Electrical signal in the computer's circuitry, used to convey information from one part to another.

Quotes Punctuation " " used to define the start and end of a string.

RAM Random Access Memory. The part of memory that can be changed by the programmer.

Register Special memory area in the CPU.

ROM Read Only Memory. Used by the computer manufacturer to hold permanent instructions, such as the Operating System. Cannot be changed by the user, except by physical removal of the chips.

Runtime error Bug in a program that only shows up when it is executed.

Spaghetti Bad programming style caused by over-use of **GOTO** commands.

String Any sequence of characters—including none at all!

String variable A variable whose value is a string.

Syntax error A mistake in the 'grammar' of BASIC.

User You.

Value The specific item, string or number, currently stored in a given variable. Can be changed by the programmer.

Variable A labelled section of memory that can hold a number or string, which can be found by looking for that label. The label is the *name* of the variable. It can be set up by the programmer for use in a program.

Variables area The section of memory that stores the variables and their values.

Video RAM The area of memory that stores information needed for the monitor display.

Visual Display Unit Fancy name for monitor or TV screen. Abbreviated to VDU.

Wordprocessor Intelligent typewriter that can process text—e.g. correct spelling mistakes, rearrange sentence order, change words.

Zero On computers this is written Ø to distinguish from the letter 'Oh'.

Commands and Symbols Index

Keywords

CLS	28
FOR	78
GOTO	62
IF	59
INPUT	30
LET	38
LIST	12
NEXT	78
NEW	14
PRINT	20
RUN	13
STOP	66, 68
THEN	59
TO	78

Punctuation

" "	21
;	25
,	27
$	32
?	13,31

Arithmetic

+	43
−	45
*	48
/	51
( )	52
.	51
Ø	4
>	62
<	61, 62
> =	61, 62
< =	62
=	46
< >	62

Special Keys and Symbols

CAPS LOCK	5
CTRL	4,21
DEL	5
ENTER	10
ESC	64
SHIFT	4
SPACE	9

Gateway to Computing

with the

Amstrad CPC464

Step through the gateway and enter a computer wonderland. It's a world of intrigue, novelty and enormous fun.

Come and meet the creatures aboard the silicon ship:

- Carlton Q. Cashsnitcher, the Gnatwest Bank Manager
- Ivan Nokyablokov, Spy with a Mission
- Sherlock Holmes and Dr Watson, Anti-bug Squad

They, and others, are involved in a curious entanglement of problems, puzzles and projects.

What's more, they're pioneers of a new 'fun' approach to BASIC learning. You'll find them very friendly towards newcomers to computing.

This is the first book in a series, specifically designed to suit *your* computer and help you to get the most out of your machine.

It is your freedom ticket to fly past the Guardians of the Gateway and seek the key to Computing.

UK £ NET +004.95

ISBN 1-85014-016-2

00495

9 781850 140160

SHIVA

Shiva Publishing Limited

Gateway to Computing with the Amstrad CPC464 BOOK 1 Stewart SHIVA