GETTING STARTED WITH B·A·S·I·C

THE BEGINNERS GUIDE TO COMPUTING

JOHN PARRY

GETTING STARTED WITH BASIC

The beginners guide to computing

John Parry

Phoenix Publishing Associates Ltd
Bushey, Herts

First published in Great Britain
PHOENIX PUBLISHING ASSOCIATES LTD
14 Vernon Road, Bushey, Herts. WD2 2JL

ISBN 0 9465 7615 7

Printed in Great Britain by
Billing and Sons Ltd
Cover design by
Ivor Claydon Graphics
Typesetting by
Sunsetters, Rickmansworth

CONTENTS

INTRODUCTION

This is a book for the *absolute beginner.* It assumes that your knowledge of computers is utterly negligible.

It starts at the very beginning and explains carefully the meaning of all technical terms used. It is not intended to turn the reader into a "computer whizz-kid" but it will enable the ordinary person to use and understand a micro-computer. There is a great deal of satisfaction to be obtained from learning how to control a computer - much more than from playing games sold on luridly labelled cassettes.

The handbooks for new motor cars used to explain such details as how to depress the clutch pedal before moving the gear lever to engage each successive gear. Nowadays the makers can assume that how to drive a car is common knowledge and they don't have to tell you that the pedal in the middle is the brake and the one on the right is the accelerator. The same situation does not exist in computers. There are striking differences between the machines in the high street shops which, while presenting no problem if you already know what it's all about, can cause hours of frustration to the newcomer struggling with a manual. You will have to consult your machine's manual for some points but since it was written by someone who had to cover every facet of the machine's operation you may well find that the answer to your problem is buried in a wealth of incomprehensible detail.

Because of the need to make sure that nobody gets left behind we have had, particularly in the first few chapters, to discuss some elementary ideas at length. We do not apologise if this is tedious - it is the only way to make sure that owners of all machines will find what they need to know.

A very large number of people feel that an increasingly important field of technological advance is passing them by because they “don’t understand computers”. I am often asked “What do you do with it?” by those in this position. The answer in my case is that I use a computer for fun -because I like them. However I hope this book will help in meeting two objectives: to allow everyone to become familiar with how a computer works and to enable as many people as possible to enjoy finding out about them.

1
Why Call It a Program ?

"When I use a word", said Humpty-Dumpty. "It means exactly what I choose it to mean, no more and no less".

- Carroll, "Alice in Wonderland."

WHY PROGRAM?

I have an automatic washing machine. On the front there is a knob known as the 'programmer'. You can turn this knob to a surprisingly large number of settings to select one of the machine's 'programs'. (Sorry about the spelling. The British Computer Society says we must leave off the 'me'.) These programs amount to a number of lists of sequences of operations. The machine is capable of several different activities, for example, 'fill cold', 'fill hot', 'normal action tumble', 'gentle action tumble', 'pump out', 'spin'. Each program is a succession of such operations in a logical order to produce the desired result.

A computer, like the washing machine, is also capable of a number of operations but since it works not on washing but on numbers and text it is useful for number and text processing rather than clothes processing. Such operations as **PRINT** and + are appropriate for the things the computer works on rather than 'spin' or 'rinse'. A more significant difference between the two machines is that in the case of the washing machine the manufacturer has provided the programs, and has provided every program I am likely to need, but the computer manufacturer sends his product out into the world unprogrammed. The users are expected to supply the programs for themselves. There are two ways to do this, you can either buy them in a shop or, far more satisfying, make up your own. It is with this second option that this book is concerned.

To summarise, a program is a list of instructions stored inside a machine which can be used to co-ordinate and organise the machine's activities to produce some useful result. Computers are machines capable of working on numbers and letters but are sold without programs inside them.

LANGUAGES

You may have gained the impression from the previous section that manufacturers have been less than fair to the public in selling machines in such a form that they are incapable of doing anything. This is in fact far from true. To revert to the washing machine consider the operation 'fill hot'. It is not as simple as it seems. To do it the machine must turn on the hot water valve and keep testing the depth of water until some pre-assigned level is reached, then turn the valve off. The other operations also consist of several such elementary actions. The designer of a program to wash for example 'fast coloureds' has to string together a list of items like 'fill hot'. He can assume that the machine contains the necessary wires and relays to make the various parts of 'fill hot' happen. The program is easy to compose because if he wants to fill with hot water he can tell the machine to get on with it and not worry about the details. It is the word 'tell" in this sentence that gives us a lead into the word 'language' when used in relation to computers. You may (or may not) have heard such words as **BASIC, COBOL** and **PASCAL.** These are all programming languages and it is in these languages that the stored lists of instructions for computers are written. Learning a language sounds a rather alarming prospect to the average English person (I don't know about the Welsh or Scottish) but now I can let you into the big unspoken secret. All these languages consist simply of a very small number of the words you use every day. The meanings are a bit more precise but that's all there is to it. I have no intention of learning German – the idea of all that vocabulary and word endings is far too off-putting –but learning a computer language offers no such horrors. It is merely a question of remembering which English words are allowed and putting them in the right order.

BASIC

The overwhelming majority of computers use a language called **BASIC** and it is this language which will be used in this book. It contains only a small number of words which you will soon learn. You will have a little more trouble with what might be called the grammar. If a foreigner says "I am living in England since three years" we can all guess what he means but we are much cleverer than a computer is. We know he has made a mistake or two but we can work out what he is trying to say. Computers are arranged so that if you make a mistake of this type in your instructions to them they don't even attempt to do what they have been told. In fact computer language designers work on the assumption that you are certain to make mistakes and go to some trouble to make sure that when you do you find out about it quickly, and in a way which enables you to decide what was wrong as easily as possible.

It is absolutely impossible to damage your computer by giving it incorrect instructions. If you programmed your own washing machine you would certainly do stupid things like trying to spin when full of water, and you would be afraid to go out when it was running for fear of coming home to find the kitchen floor awash, but there is no corresponding danger with a computer. If you could do any harm to a computer by making similar mistakes the manufacturers would be beseiged by people quite rightly demanding their money back. They are not because every possible inappropriate thing you might do (except pushing it off the edge of the desk) has been allowed for.

GETTING STARTED

Let's assume that you have got your computer home, plugged it in to the power point and to the TV set, fiddled with the TV controls to get a good picture, everything has gone according to plan and you are wondering what to do next.

THE CURSOR

The computer is operated by typing on a keyboard and watching what happens on the screen. Most of what appears on the screen is what was typed on the keyboard but some, the interesting bit, is what the computer decides to put there. A cursor is a thing on the screen to show you where what you type next will be put. (It may be a solid black or white block or a horizontal line level with the bottom of the letters.) When you type something the cursor usually moves to the right ready for what you type next but if you have filled the width of the screen the cursor will jump to the left hand end of the line below. Most machines have a cursor which flashes all the time to make it easy to find on the screen.

KEYBOARDS

The first thing to do is to get used to using the keyboard. These are regrettably very varied. You may have one which looks almost the same as a typewriter with a shift key to change small letters into capitals and a long bar along the bottom to get a space but you may well have rubber keys with a bewildering number of different things written on, above and below them. I suggest you play with it to see what happens. There is a number of possibilities so you will have to be patient while I try to cover them all.

Basically what you type on the keyboard appears more or less exactly on the screen just as if you were using a typewriter and the screen was the paper. If you are used to a typewriter the way the shift and shift lock keys work may surprise you. A typewriter provides small letters on all the letter keys but capitals if shift is held down while another key is pressed. The shift key also alters the effect of the other keys so that you get either a 2 or quotation marks by pressing the same key. On some computers **SHIFT** has the typewriter effect on the non-letter keys but changes capital letters to small rather than small to capital. There may well be a key marked **CONTROL,** or some abbreviation for it which works like another shift key and gives the other keys a third meaning which will have some surprising effects on the screen but, as I stressed before, can't do any harm.

The Sinclair ZX81 and Spectrum computers put whole words on the screen when you press a single key or a single key preceded by one or more shift keys. If you look at the screen the cursor changes to show the effect the keys are going to have. This system is frustrating at first but saves time when you get used to it. It is explained in the manual but trial and error is probably a more effective way to make progress.

CORRECTING MISTAKES

There will certainly be a key for rubbing out mistakes, possibly marked **'DEL'** for **DELETE** or **'BS'** for back space though it may require the shift key to be held down to produce this effect. If you are typing and you notice a mistake you can press this key and what you typed last will vanish. You can press the **DELETE** again to remove what you typed before and thus work your way back to the mistake. Then you can type into the space created by moving back and replace the error with what you really meant.

LINES

On an electric typewriter there is a key for 'carriage return'. When the typist finishes whatever is required on one line this key is pressed to move the paper so that the next letter typed will be at the start of a new line. Those of us with manual typewriters have to reach up and push the carriage back for ourselves. There will be a similar key somewhere on the right hand side of your computer keyboard marked **'RETURN'** or **'NEW LINE'** or **'ENTER'** or something similar. You will find that it has rather more effect on the computer than the other keys. The way the computer is arranged to make giving it instructions simple is that the computer does not attempt to understand what you type as you type it. It just puts what you type onto the screen. This gives you a chance to see what you are typing and to correct any errors. When you are satisfied that the line contains exactly what you want (which may be only a single word but may in some machines stretch over several screen lines) you press the **'RETURN'** key. This is the signal to the computer that you have completed what you wanted to say and you want the computer to look at what you have typed.

If you typed something meaningless to the computer, say your name, then there are two possible ways it can deal with the problem. In most cases your name is left on the screen and the computer puts a message on the next line to say it doesn't understand. This may well be 'syntax error' or some other bit of 'computerese'. No harm has been done and you can type something else. Alternatively some machines try to help by leaving the cursor on the line when you press **RETURN** so that you can use the rub out key to correct it. If you eventually alter the line till it is one that the computer understands you can press the **RETURN** key, the computer accepts the line, and you can type another. You will notice that the word 'line' is used rather loosely. There are two possible meanings. Logically a line is that length of screen between the left and right hand sides, but we tend to use the word to mean what you typed between presses of the **RETURN** key. Since some computers can only put a smallish number of letters across the width of the screen, perhaps as few as thirty-two, and you might wish to type more than thirty-two letters before pressing **RETURN** it is usual for the cursor to jump automatically to the left hand end of a new line of the screen when you have typed your way to the right hand side. However everything typed up to pressing **RETURN** is called a line even if it is long enough to stretch over more than one line of the screen. Don't worry if you can't follow this straight away – you will not need to think about it because it will become automatic very rapidly.

I must stress again that errors are normal. The makers of computers go to a lot of trouble to protect you from yourself by not letting the computer follow meaningless instructions which cannot be what you meant to say. The quality of the error messages may well leave something to be desired because they are trying to save space inside the computer but it is the computer's job to help you to type correct instructions.

PRINT

We haven't got as far as writing programs yet but it is time to get the computer to do something meaningful.

Use your keyboard to get this on the screen

```
PRINT "HELLO"
```

then press the **RETURN** key.

You will see the word **HELLO** appear on the screen. **PRINT** is the first word of the **BASIC** language you have learned. It tells the computer to write onto the screen whatever appears after the word **PRINT.** You will find that you can write any message you like.

```
PRINT "TOMORROW IS TUESDAY"
```

will print the words between the quotation marks in the same way. Notice the quotation marks. If you leave them out you will get an error message which means that the computer did not know what to do.

The most important thing to be remembered from this chapter is that the computer doesn't bite if it doesn't like you. You can do what you like, apart from physical abuse, and it just keeps replying with error messages which are it's way of trying to help. You may be quite extraordinarly unlucky and type something it understands which causes it to do something which at present is not particularly sensible (similar to the washing machine leaving the inlet valve open even though it's full of water). If this happens and, for example, the keyboard ceases to have any effect, you can always unplug it and start again from scratch – the computer won't mind, it's much more patient than you.

2
Lining Up Numbers

"I've got a little list,
And they'd none of them be missed."

W. S. Gilbert - "The Mikado"

Chapter one introduced the idea that a computer is a machine for processing numbers and text. It is an extreme example of a product which is made without any decision having been made about what it is to be used for. This is called 'deferred design'. The washing machine contains programs of instructions for all the different sorts of washing you might want to do but you can put your own programs into your computer so that the same computer can, for example, play chess or keep accounts for a business. What it does depends on the program of instructions you put into it.

DOING SUMS

So far we have used one word of the **BASIC** language in which these instructions are written. It was the word **PRINT** and it tells the computer to put things on the TV screen. Before trying to store programs it will be useful to find out more about **PRINT.** Use your keyboard to set this on the screen.

```
PRINT 5+9
```

When you press the **RETURN** key after the 9 you will see 14 on the screen. This is the result of adding the 5 and the 9 together.

The computer worked this out in the instant between your pressing the **RETURN** key and the 14's appearance on the screen but you certainly didn't notice any delay. You will find that the computer is equally rapid if you do

```
PRINT 1234+9876
```

This is another use of the instruction **PRINT.** It tells the computer to work out whatever it finds to the right of the word **PRINT** and put the answer on the screen. Of course if what it finds on the right of the **PRINT** does not contain any arithmetic the answer will be what it found, as in

```
PRINT 7
```

or

```
PRINT "CHRISTMAS"
```

As well as adding you can subtract. Try

```
PRINT 13—8
```

and

```
PRINT 5476—345
```

Your keyboard will not have a normal multiplication sign because of the possibility of confusion with the letter X so the ⋆ is used instead.

```
PRINT 7⋆6
```

will, as you would expect, print the answer 42.

For division the sign / is used so that

```
PRINT 2ØØ/5
```

will give the answer 4Ø.

If you have had trouble in making the above examples work there are several possible reasons. One is confusion between the letter capital O and the number zero Ø. If you type

```
PRINT 4O3★17
```

when you mean

```
PRINT 4Ø3★17
```

then the computer will not understand you.

Notice that long numbers where you might usually use a comma to group the figures into threes must be typed without the commas. If you want to give the number 'two hundred and fifty thousand' to the computer then you must type

```
25ØØØØ
```

not

```
25Ø,ØØØ
```

THINGS TO DO

I suggest that you use **PRINT** to work out as many arithmetical answers as you have time for in the way I have shown you. It will get you familiar with the keyboard and used to the way your machine behaves when you make the inevitable mistakes. Some suggestions follow.

(1) Work out the number of hours in a week. (Don't forget that the times sign is a ★.)

(2) How far have you driven if the mileage recorder of your car goes from 47,563 to 56,102 in a year? (Don't forget to leave out the commas.)

(3) One inch is 2.54 centimetres so to change a metre, which is 100 centimetres, to inches you have to divide 100 by 2.54. (Don't forget that the divide sign is / and take care not to mix up the **Ø,** zero, with the capital letter **O**.)

(4) Find the average age of a group of old age pensioners whose ages are: 67, 73, 81 and 69. To do this you will have to add these numbers together and divide the result by 4. One way to do this would be to use

```
PRINT 67+73+81+69
```

making a note of the answer, which is 29Ø, and then use

```
PRINT 29Ø/4
```

but it would be better to do it all in one so like this.

```
PRINT (67+73+81+69)/4
```

The brackets force the computer to do the arithmetic in the brackets first.

If you did

```
PRINT 67+73+81+69/4
```

then it would divide 69 by 4 first and add the answer onto the other three numbers, which is not what we wanted.

(5) To change a temperature in Fahrenheit to the same temperature expressed in Centigrade you take away 32 then multiply the result by 5 and divide by 9. Change 98 degrees Fahrenheit to Centigrade. You will have to use brackets to make the computer do the subtraction before it does the multiplication.

The answer should be 36.6666

A PROGRAM

So far we have used the computer just like a calculator. You told it what to do and the answer appeared straight away. The whole point of a program is that you store instructions inside the computer and then get it to follow the instructions later.

Type this

```
10 PRINT "SUNDAY"
```

When you press **RETURN** the instruction will not be obeyed. You will not see **SUNDAY** on the screen as you would from

```
PRINT "SUNDAY"
```

The reason for the difference is the 10. This is called a line number. When you type a bit of **BASIC** with a line number in front the computer remembers it but does not obey it. You can show that it has remembered by typing

```
LIST
```

followed as usual by the **RETURN** key.

This is your second word of **BASIC** and it tells the computer to make a **LIST** of your program on the screen. In this case the program contains only one instruction so the list is quite short but you can easily add some more by typing

```
20 PRINT "MONDAY"
30 PRINT "TUESDAY"
```

Now when you type

```
LIST
```

you will see all three lines on the screen.

People using Sinclair machines may be surprised at all this mention of **LIST** because these computers work in an unusual way. In most computers **LIST** is the only way to find out what program has been stored in the computer but the Sinclair version of **BASIC** automatically shows a list of some or all of your program every time you type a new line. **LIST** is still available though and you will need it when you write longer programs.

I am assuming that you have typed the three lines numbered 1Ø, 2Ø and 3Ø as shown above and made sure, with **LIST** if necessary, that they are safely stored inside the computer.

What we need now is a way to tell the computer to obey the instructions of the program. The word for this is **RUN.**

If you now type

```
RUN
```

followed by the usual **RETURN** you will see on the screen

```
SUNDAY
MONDAY
TUESDAY
```

Typing **RUN** so as to make the machine follow the instructions of a program is called 'running a program' and when the machine is obeying instructions of a program it is said to be 'running'. You might like to extend the program to make it print the other four days of the week as well.

LINE NUMBERS

You may have wondered why I chose to number the lines 1Ø, 2Ø and 3Ø rather than 1, 2 and 3. Going up in in tens is usual because very often one needs to insert another line between two existing lines and this would be impossible if we needed to use a number between 1 and 2 as line numbers have to be whole numbers. However, you could use any number from 11 to 19 to set an extra line between line 1Ø and line 2Ø.

Suppose you had stored these lines in the computer, as shown by **LIST**

```
1Ø PRINT "JANUARY"
2Ø PRINT "MARCH"
```

If you were now to type a new line

```
15 PRINT "FEBRUARY"
```

then typing **LIST** would show

```
1Ø PRINT "JANUARY"
15 PRINT "FEBRUARY"
2Ø PRINT "MARCH"
```

You will notice that the computer is intelligent enough to arrange the lines inside itself in order of the line numbers regardless of the order in which they were typed. When you run the program it follows the order of the line numbers so that it does the lowest numbered line first and the highest numbered last. (Unless you want something different – this will be explained later.).

You will find that sometimes you wish to remove a line from the stored program, either because you have changed your mind about what you want the program to do or because it contains a mistake. Suppose your program contained a line like this

```
12Ø PRINT "DECMBER"
```

then if you typed

```
12Ø
```

and pressed **RETURN** immediately, then you would find, by using **LIST,** that line 12Ø was no longer in the program.

You are more likely to want to replace the line with the correct one, however, and you could do so by typing a new line with the same number.

```
12Ø PRINT "DECEMBER"
```

would replace the existing line 12Ø with the new version.

If you wish to remove a complete program from your computer then one possibility would be to remove the lines one at a time by the method above but there is a **BASIC** word, **NEW,** which has the desired effect all at once. I should use it cautiously if I were you because it is surprising how often you change your mind.

We have now covered four words of **BASIC, PRINT, LIST, RUN** and **NEW.** As I said earlier they are everyday English words which have a precise meaning. They will soon become very familiar.

EXPERIMENTING WITH PROGRAMS

You have now learned how to store, amend and delete programs from your computer and how to make it follow the instructions of a program. It will be a good idea to practise by writing programs to put on the screen, for example, the days of the week and months of the year as suggested above. You could also write programs to list the names of the members of your family and friends. This is not particularly ambitious as programs go but will develop your facility at entering, editing and running programs.

3
Variable Things

"O Woman! in our hours of ease,
Uncertain, coy, and hard to please,
And variable as the shade
By the quivering aspen made;"

- Scott, "Marmion".

MEMORY

We have seen how a computer is a machine which follows instructions stored inside itself and you have learned how to store and edit these instructions. Obviously the computer must contain something which could be described as 'memory' otherwise it would not be able to remember the lines of program you typed. You will have seen the abbreviation **'RAM'** mentioned when computers are described as having for example **'48K RAM'.** It stands for **'RANDOM ACCESS MEMORY'** and the number gives an indication of how large a program the computer has room for.

This memory has another purpose however. It is used for storing words and numbers while the computer is working on them. Storing words will be covered later but we will now deal with how numbers are stored.

You can imagine that the inside of your computer contains a large number of pigeonholes each with a label. Each pigeonhole has room for exactly one number to be put into it and there are **BASIC** instructions for putting a number in and looking at the number put in previously. For labels we use the letters of the alphabet and sometimes longer names. This makes some of **BASIC** look rather like an algebra textbook which most people, (understandably you may well think), find rather off putting, so I will resist my inclination to use letters like X, Y and Z as all the alphabet is available. There is some slight similarity to the concept of using letters in algebra but you need no mathematical knowledge beyond what is needed to cope with everyday life to understand what follows. It is a historical accident as much as anything that has put the idea into the public mind that computers are something to do with mathematics and that you need to be a mathematician to use them. It's just not true.

LET

Type this

```
LET H=12
```

remembering to press **RETURN** as usual.

The word **LET** is the **BASIC** instruction which is used to put a number into a pigeonhole. In this case it has been used to put the number 12 into the pigeonhole labelled H. You can show that it is in there by typing

```
PRINT H
```

When you press **RETURN** you will see the number 12 on the screen. **PRINT H** does not mean "print the letter H", but "print the number in the pigeonhole labelled H". This is the reason for the quotation marks in the previous chapter. If you want the computer to print the letter H you would have to use

```
PRINT "H"
```

rather than

```
PRINT H
```

This can be a little confusing at first so please make sure you see the difference. The H in the second case is the label on a pigeonhole, in this case a pigeonhole with 12 in it, whereas in the first case the computer doesn't need to care about the meaning of what it finds between the quotation marks – it only has to print what it finds there.

VARIABLES

To avoid using the word 'pigeonhole' all the time it is usual to use 'variable' as the name for them and to call the labels 'variable names'. This is appropriate, as you will find later, because the reason they are useful is that the number stored in a variable can vary as your program runs.

You saw in chapter 2 how to use **PRINT** to make the computer do arithmetic as in

```
PRINT 7★12
```

You can do exactly the same things with the numbers stored in variables as you can with numbers typed on the keyboard. You just have to mention the name of the variable instead of a number. If you have put the number 12 into H by using **LET** as described above then

```
PRINT 7★H
```

will tell you exactly the same answer. **7★H** means 'seven times the number in H'. You will be able to predict the outcome of typing

```
PRINT 9★H
PRINT 17—H
```

and

```
PRINT H★H
```

If you try

```
PRINT H★H★H★H★H
```

then the computer will work it out more quickly than you could yourself. (It's 248832.)

A PROGRAM

Suppose petrol costs £1.84 per gallon. The program that follows tells you the cost of a list of numbers of gallons. It is not a very ambitious program but it shows the idea.

```
10 LET C=1.84
20 PRINT C
30 PRINT 2★C
40 PRINT 3★C
50 PRINT 4★C
60 PRINT 5★C
70 PRINT 6★C
```

When you type **RUN** after entering the program and using **LIST** if necessary to check that it is correct the computer puts the cost of each number of gallons from 1 to 6 on the screen. If you had a printer the list could be given to the pump attendant in a (very primitive) garage. The advantage of putting the variable name C into the program in lines 20 to 70 rather than the number 1.84 is not only that it is easier to type. If you wanted a list for two star petrol instead of four star it would only be necessary to change one line, line 10, to

```
10 LET C=1.79
```

and when you used **RUN** again you would get the correct prices. Because C is a variable the number in C can be changed to make the same program do a different job.

Variables are important because they allow the possibility of writing programs in such a way that they can be used to work on different numbers each time they are used. A program which prints pay slips has to use different numbers, obtained from clock cards and tax tables, for each employee. If the program worked on fixed numbers it would be almost useless.

Most people buy petrol by value rather than by volume so that what you need to know is how many litres you can get for £1, £2, £3 and so on. You might like to write a program to find these numbers. It would have to start with a line to put the price of one litre into a variable, say P for price, using **LET.** The lines following would divide each amount of money by the number in this variable. It will be necessary to use 1ØØ, 2ØØ, 3ØØ and so on for the amounts of money since you will want to put the price of a litre in pence. Don't forget that the sign for divide is / and remember the possibility of confusion between the capital O and the Ø.

INPUT

The instruction **LET** as used so far always puts the same number into a variable. It would be preferable to tell the computer what number we wanted used while a program is running because then we would not need to change a line of program to make the program work on a different number. The **BASIC** word for giving a number to a running program is **INPUT.** Type this program. Remember first to type **NEW** to clear the memory of previous programs.

```
1Ø INPUT N
2Ø PRINT 16★N
```

Check with **LIST** that this is the program in your computer.

The purpose of the program is to calculate the total cost of any number of 16p stamps. The **INPUT** works like this. When you type **RUN** the computer as usual obeys the instructions in order of the line numbers until it gets to the **INPUT.** Then it stops and waits for the user to type. Most computers put a question mark on the screen to show that the program is waiting for you to type something. When **RETURN** is pressed the computer puts the number typed into the variable mentioned in the **INPUT** instruction, in this example N, which is being used to hold the number of stamps, and continues with the next line of the program.

Type **RUN** and you will be able to find for example that 5 sixteen pence stamps cost 8Øp. You will have to type **RUN** again for a second go when you might find that 137 stamps cost 2192p. Obviously it would be a good idea to change line 2Ø to

```
2Ø PRINT 16*N/1ØØ
```

so that the answer comes in pounds and pence.

When **INPUT** is used the computer has as usual to be ready to deal with errors. What if you typed something that wasn't a recognisable number? There are lots of possibilities. You might type **'TUESDAY'** or **'SEVEN'** or **'4.3.67'** or you might just press return without pressing any other key first. The last case might be assumed by the computer to be Ø but the others require it to take some sensible action. Some **BASICs** ask the user to try again by putting a message like **'?REDO FROM START'** on the screen and waiting for another attempt. In others the return key refuses to take effect and you can use the **RUB OUT** key to correct your number. Eventually you can press return on a valid number and it will be used.

Almost all versions of **BASIC** have an improvement on the **INPUT** instruction which enables the program to tell the user what he or she is expected to type. This is done by putting some words in quotation marks after the word **INPUT** and followed by a semi-colon. The following program asks you the length, width and price per square metre of a rectangular carpet and tells you what it will cost.

```
10 INPUT "LENGTH";L
20 INPUT "WIDTH";W
30 INPUT "PRICE PER SQ. M.";P
40 PRINT L*W*P
```

When you **RUN** this program you will see the advantage of using **INPUT** in this way. When the computer wants you to type the length the word **LENGTH** appears on the screen so you know what is expected. You are then asked for the width and price of one square metre by lines 20 and 30. Line 40 multiplies the length by the width to get the area and multiplies the result by the price of a square metre so it can print the price of the whole carpet.

You will no doubt be able to think of a number of ways to use programs similar to the one above. The idea can be applied to such calculations as those required for gas and electricity bills, rates and income tax.

This chapter has introduced some very simple but extremely powerful ideas. The concept of a variable and of telling your computer numbers while it is running a program open a whole range of possibilities which may well not have occurred to you.

People familiar with computers tend to treat these elementary notions as if they were obvious when talking to the newcomer to computing which is a possible explanation of the glassy-eyed bewilderment that an explanation of even the simplest program often elicits. I would strongly advise that you re-read the first three chapters and practise the examples thoroughly before moving on. It is much easier to understand new ideas if the foundations are thoroughly understood.

4
Dots and Dashes

"Printing has destroyed education"

Disraeli – "Lothair"

This chapter does not introduce any important new aspects of **BASIC.** Of necessity it has been impossible to tie up all the loose ends in the story so far. We will attempt to answer a few questions that may have occured to you and show some refinements.

COMMAS

The **PRINT** instruction is probably the most used word in **BASIC** because the object of a program is to set the computer to show the user some results. Up to this point we have used it to print only one item for each **PRINT**

```
PRINT "HELLO"
PRINT 19★21
```

and

```
PRINT X
```

all **PRINT** only the one object mentioned after the word **PRINT.**

All versions of **BASIC** allow you to **PRINT** more than one item by putting a comma between the items. The table of petrol prices could have been improved by getting the computer to put the number of gallons and price on the same line as in

```
PRINT 3,3★C
```

The effect of the comma is to cause the second item, the number 3★C, to be printed several spaces to the right of the first item, the number 3. This works if the item is a word in quotation marks as well so that

```
PRINT "THREE", 3★C
```

would cause the word **THREE** to be put on the left and the cost of the three gallons on the right of it.

In the same way one could make both items to be printed words in quotation marks. If you wanted headlines for the columns of the table the first thing to do would be to **PRINT** the words **GALLONS** and **PRICE.** You could use a line like this to do it.

```
PRINT "GALLONS", "PRICE"
```

The result of including these improvements would be a program like this.

```
10 INPUT "COST OF A GALLON PLEASE";C
20 PRINT "GALLONS", "PRICE"
30 PRINT "ONE",C
40 PRINT "TWO",2★C
50 PRINT "THREE",3★C
60 PRINT "FOUR",4★C
```

and so on.

You will find if you run a program on these lines that the commas have the effect of putting the things **PRINT**ed by each line neatly under each other to make columns. The precise effect of a comma varies in the different versions of **BASIC** but the most usual effect is to move the point at which printing will next be done fourteen positions across. If you have used a typewriter with the ability to set tabs you will recognise the system. On such a typewriter you can press the tabulator key and the carriage jumps so that the next item typed will be in a pre-set column. Putting a comma into a **PRINT** line is like asking the computer to press its tab key when it gets to the comma. It is difficult to be precise about this because computers are very different in the number of letters on each line so that in some cases there are only two tab positions and thus two columns but you may have as many as five.

You might like to try the effect of a line like

```
PRINT "TOM","DICK","HARRY"
```

to see what happens.

SEMI-COLONS

Sometimes you need to **PRINT** items next to each other rather than spaced out as they would be if you use a comma in the **PRINT** instruction. This is done as follows.

```
PRINT "COST IS ";6★C
```

This line would cause the number worked out by doing 6 times the number in the variable C to be printed after a space after the IS. I put a space between the S of IS and the closing quotation marks so as to make sure the number didn't start straight after the S which looks a bit untidy.

BLANKS

If you want to **PRINT** a blank line to separate two lines printed on the screen then it can be done by using a **PRINT** with nothing following it. If a program contained lines like these

```
10 PRINT "PETROL PRICE TABLE"
20 PRINT
30 PRINT "GALLONS","COST"
```

then a blank line would appear on the screen between the words **PRINT**ed by lines 10 and 30.

VERY LARGE NUMBERS

Inside a computer a fixed amount of memory is used for each number whether the number is small, say 0, or large, like a million. This means that there is a largest number your computer is capable of using. This limitation is not likely to be a problem however since the size of it will be about

170,000,000,000,000,000,000,000,000,000,000,000,000

that is 17 followed by thirty-six 0's.

If the computer has to **PRINT** a large number the space taken up by printing it in this form would be very large and such numbers are very hard to read so a different system is used. Try this

```
PRINT 9000000*9000000
```

The result written in the normal way would be

81000000000000

but the result shown on the screen will be

```
8.1E13
```

The 13 in this system tells you how far the computer has put the decimal point from where it ought to be. In this case it has put a point between the 8 and the 1 and it has to be moved 13 places to the right.

In the same way

```
8.1E-13
```

would be **PRINT**ed instead of

```
0.00000000000081
```

Most people are unlikely to use such large or small numbers but you may well see them if you make mistakes.

VARIABLE NAMES

All versions of **BASIC** allow you to use the letters of the alphabet as the names of variables. In the last chapter we used H, C and P but could have used any of the other letters. Almost all versions let you use longer names as well. You will recall that a variable name is just the label on a pigeonhole for putting a number in. When a program is running a new pigeonhole is created each time a new variable name is encountered in the program so it is not too difficult for the designers of **BASIC** to make longer names possible. Thus instead of using C to store the cost of a gallon of petrol we could have used **COST** or even **COSTOFAGALLON** (you can't put spaces in). This is supposed to make programs easier to understand but I have found that it causes trouble because people learning **BASIC** forget which words are part of **BASIC**, like **LET**, **PRINT** and **LIST**, and which are variable names invented by the programmer. You will almost certainly be advised to use long names by your machine's manual but I will use single letters throughout this book.

POWERS

There is another arithmetic operation as well as addition, subtraction, multiplication and division. It is called 'raising to a power'. The sign for this is ↑. If you use the line

```
PRINT 2 ↑ 5
```

the result will be 32 because

```
2★2★2★2★2
```

is five twos multiplied together

This will be of use to some readers but if you are not familiar with the idea you won't need to worry about it again.

BINARY NUMBERS

You may have been intimidated by clever eleven year olds who have been taught things like.

```
1ØØ1 + 111ØØ = 1ØØ1Ø1
```

This sort of thing has been introduced by maths teachers because it is 'relevant to computers'. They are right in a way because inside the computer everything is done by 1's and Ø's rather than the ordinary digits Ø,1,2,3 ... 9. However the reason for inventing such languages as **BASIC** was to avoid the need for messing about with how the machine actually works. A driving instructor who started his first lesson with the details of the carburettor or the geometry of the Ackerman steering principle would not help you to drive the car from one place to another and this situation is similar. Using **BASIC** avoids the complications and lets you drive the machine.

5
Lets and Loops

"To business that we love we rise betime, And go to 't with delight"

Shakespeare -" Anthony and Cleopatra"

MORE ON LET

When the **LET** instruction was introduced we used it to put a number into a variable, as in

```
LET H=12
```

or

```
LET A=1
```

The effect of these instructions, you will remember, is to put the number on the right of the equals sign into the pigeonhole, or variable, whose name appears on the left of it. The **LET** instruction is in fact more versatile than this and it is now time to show you how much more it can do. It is difficult, at this stage, to demonstrate practical examples because other parts of **BASIC** are needed as well but if you are prepared to take this on trust you will find that it helps later.

Firstly you can put on the right of the equals sign anything that the computer can work out. This means that possible program lines containing the **LET** instruction include

```
33Ø LET H=24★7★52
```

or

```
100 LET T=(98.4-32)*5/9
```

or

```
227 LET A=(10000/35)*1.84
```

I have put a line number in front of these instructions because you are not likely to want to use **LET** in this way except as part of a program. You may have guessed what these **LET**'s were intended to do. The line 330 works out the number of hours in a year putting the number into H. Line 100 was to change 98.4 degrees Fahrenheit, blood temperature, to Centigrade, and put the resulting number into T. The line numbered 227 calculated the cost of petrol for 10000 miles driving at 35 miles per gallon if a gallon of petrol costs £1.84. I have used brackets to make the computer do the sums in the right order where there is more than one arithmetical operation in a line. Of course it would have been equally possible to put each of the expressions on the right of a **PRINT** as in

```
227 PRINT (10000/35)*1.84
```

but this would result in the number appearing on the screen. If you use a **LET** then the number gets put into a variable instead. You will see later why this might be a useful thing to do.

It is also possible for **LET** to use the number in a variable as one of the numbers for its arithmetic, as well as numbers typed into the line. A program might contain lines like

```
500 LET D=P*1.47
```

or

```
230 LET C=(F-32)*5/8
```

The first of these multiples the number in P by 1.47 and puts the result into D. The number in P is left unchanged but D will contain a new number. If we had arranged for P to contain a number of pounds and the exchange rate was 1.47 dollars for each pound then D will contain the number of dollars you can get for that number of pounds. The second example converts any temperature from Fahrenheit to Centigrade. If we put a number like 98.4 where the F is then the line would only convert this one temperature but by using the variable F one can put the desired number into F beforehand and use this line to change it to Centigrade.

So far we have shown how to put a completely new number into the variable on the left of the equals sign but the final use of **LET** is slightly different. Consider this line

```
70 LET N=N+1
```

The letter N appears on both sides of the equals sign. What happens when this line is **RUN** is that the number in N gets bigger by one. The right hand side tells the computer to add one onto the number in N and the left tells it where to put the answer – in this case into N, so the effect of the line is to increase N by one. You would need to do this if the computer was counting things. If the number in N increases by one every time something happens then provided N started off containing Ø it will contain the number of times the event has occured.

In the same way programs often contain lines like

```
190 LET C=C—1
```

or

```
430 LET P=P*2
```

The first of these reduces the number in C by one while the second doubles the number in P.

I hope you have been able to take in all this information about **LET** without too much trouble. These ideas will be repeated continually in the rest of the book so there will be plenty of revision but you will find it easier to move on if you are quite sure of it now.

PAYING THE RATES

A program follows which uses the **LET** instruction to do some calculations concerning rates. Most readers will know that every property has a number called the "rateable value" which is fixed by the valuation officer and which is used to work out how much should be paid on that property. Every year the local councils decide on a sum of money called "the rate" expressed as a number of pence per pound of rateable value. To find out what you have to pay you multiply the rateable value of the house by the rate. It is possible to pay either in two six-monthly instalments or to pay ten instalments at monthly intervals. The program asks for the two numbers needed to work out this year's rates then asks how many instalments you are going to pay and tells you how much they will be. Here is the program.

```
10 PRINT "PROGRAM TO CALCULATE RATES"
20 INPUT "RATABLE VALUE IN POUNDS";V
30 INPUT "THIS YEAR'S RATE IN PENCE";P
40 LET M=V*P/100
50 INPUT "HOW MANY PAYMENTS";N
60 LET I=M/N
70 PRINT
80 PRINT "RATES THIS YEAR ARE: ";M
90 PRINT "PAYABLE BY ";N;" PAYMENTS"
100 PRINT "OF ";I
```

I have used V for the rateable value and P for the rate in pence. In line 3Ø these are multiplied and the result divided by 1ØØ to change it from pence to pounds. The number of instalments is put into N by line 5Ø so that the rates payable, in M, can be divided by this number to give the size of each payment. You will notice that I have used a blank **PRINT** to get a blank line and have included words to show what the numbers **PRINT**ed mean. The semi-colons are to make the numbers come next to the words when they appear on the screen. You may wish, when you have **RUN** the program, to improve the appearance of the way it gives its results by changing the form of the **PRINT** instructions in lines 9Ø and 1ØØ.

LOOPS

Every program so far has done one job once only. You may well have felt that a sledgehammer approach has been used to crack a rather insubstantial nut in almost every case and I would have to agree. The reason is simple. One almost never writes programs with the intention of using them only once. The whole power of computers lies in their ability to repeat the same task frequently, rapidly and automatically. Now that you are familiar with the notion of a variable and the use of the **LET** instruction you are in a position to appreciate how this can be done. One more **BASIC** word will be needed. The word is **GOTO.**

I mentioned earlier that when you **RUN** a program it obeys the instructions in order of the line numbers unless you want something different. The instruction **GOTO** is one of the ways to change the order of working. Consider this program (but don't **RUN** it yet).

```
1Ø PRINT "PHOENIX PUBLISHING ASSOCIATES"
2Ø GOTO 1Ø
```

The word **GOTO** in line 2Ø is followed by a line number, here the line number 1Ø. The **GOTO** does nothing except tell the computer what to do next. The effect of this is that after printing the words in line 10 the computer gets to line 20 which sends it back to line 10 again. Obviously this cycle will be repeated indefinitely unless something interrupts it. Now you can change the **PRINT** instruction to your own name and watch the screen fill up

What actually happens when you **RUN** such a program depends, I'm afraid, on which computer you are using. There are two main possibilities. One is that when the screen fills the printing on the top line vanishes off the top, all the rest of the screen moves up a line, and the next line to be **PRINT**ed goes into the blank line thus created at the bottom. This is called 'scrolling'. Since each line printed consists of the same words you see a flickering effect. In other cases the program stops running when the screen is full giving you a chance to read what is on the screen. When satisfied you can press any key and printing will continue till you have had another screenful. Some machines have a key which can be pressed to change between the two systems.

In either case there is certain to be a way to interrupt the program so you can do something else. It may be marked **ESC,** for 'escape' or possibly **BRK** or **BREAK.** Also fairly common is 'control C'. This means pressing the C key while holding down the **CONTROL** key. You will undoubtedly have this information available in the manual supplied with your machine somewhere. I am sorry to have to resort to this advice but there is a sad lack of standardisation. If all else failed you could unplug the computer and start again but if you fell back on this you would of course lose your program.

The word 'loop' means a piece of program with a backward jump at the end causing the piece of program to be repeated when it is **RUN.** A 'jump' here means an instruction which changes the order in which instructions are obeyed. You now know one jump instruction, **GOTO.**

Here is another program with a loop.

```
10 LET N=1
20 LET V=11
30 PRINT N;" TIMES ELEVEN MAKES ";V
40 LET N=N+1
50 LET V=V+11
60 GOTO 30
```

As you might have realised by looking at line 30 the program **PRINT**s an eleven times table. The first time it gets to line 30 the number in N is one and the number in V is eleven so it tells you that

```
1 TIMES ELEVEN MAKES 11
```

Lines 40 and 50 increase the number in N by one and the number in V by eleven. So when the **GOTO** in line 60 sends the computer back to line 30 you will see

```
2 TIMES ELEVEN MAKES 22
```

This cycle will be repeated indefinitely until it stops because you interrupt by pressing a key, or on some machines, till the screen is full.

Your petrol program should now look like this.

PRICE OF PETROL

This program will tell you how many litres of petrol at 40.5p you can get for each number of pounds.

```
10 LET P=40.5
20 PRINT 100/P
30 PRINT 200/P
40 PRINT 300/P
50 PRINT 400/P
```

```
60 PRINT 5ØØ/P
70 PRINT 6ØØ/P
80 PRINT 7ØØ/P
```

and so on.

The improved petrol price table can be produced like this.

```
1Ø INPUT "PRICE OF ONE GALLON";P
2Ø LET C=P
3Ø PRINT N;" GALLONS COST ";C
4Ø LET N=N+1
5Ø LET C=C+P
6Ø GOTO 3Ø
```

If you choose a variable to contain the number of gallons and another to contain their cost you can write a program using the pattern of the one above to tell you the cost of each number of gallons. Similar possibilities will occur to you.

6
Fors and Nexts

"Birds in their little nests agree
With Chinamen but not with me."

Belloc - "On Food"

The parts of **BASIC** covered so far have, I hope, been easy to understand if you have studied the examples carefully. By now you will be familiar with your keyboard and will have become used to setting up the computer and how it behaves. You will have lost the initial fear of the computer which makes so many people act as if they were trying to learn lion-taming or hang-gliding and will be treating your computer with something like the attitude you show to the vacuum cleaner or TV set. You may perhaps have been disappointed by the rather limited scope of the uses to which your computer has so far been put. I'm afraid that this is unavoidable. The advanced features of **BASIC,** which make impressive programs possible, depend on the elementary parts.

So far we have learned the use of the **BASIC** instructions **PRINT, INPUT, LET** and **GOTO.** Each of these words has its own rules in the grammar of **BASIC. PRINT** has to be followed by something printable and can either be typed after a line number so that it is part of a program or without a line number so it is obeyed at once. **INPUT** on the other hand has to be part of a program and must be followed by a variable name. **LET** is more complicated because it needs a variable name then an equals sign then something to be worked out. **GOTO** only needs a line number after it.

FOR AND NEXT

The next **BASIC** words are more powerful and therefore need more explanation of the rules for their use. Unlike all the other words so far they must both be present in a program. If you use **FOR** in a program there has to be a **NEXT** later. If this is not the case then either the computer will not behave as you expected or it will give you a 'mistake' message when you **RUN** your program - in just the same way as using a left hand bracket, (, forces you to use a right hand bracket,) , later.

The purpose of using **FOR** and **NEXT** is to cause the part of the program between them to be repeated a number of times when the program is **RUN.** Here is an example.

```
10 FOR N=1 TO 10
20 PRINT "WE ARE DOOMED"
30 NEXT N
```

This very short program causes the **PRINT** instruction in line 20 to be obeyed ten times so that you see the words enclosed by the quotation marks on the screen ten times. You will notice that the variable name N is mentioned after the word **FOR** in line 10 and after the word **NEXT** in line 30. Any variable name could have been used but it has to be the same one both times.

The part of line 10 after the equals sign controls how often the part of the program between the **FOR** and the **NEXT** will be used. Here the equals sign is followed by '1 TO 10' which, as you would expect, means ten times. If you replaced line 10 with

```
10 FOR N=1 TO 5
```

then the words would be printed five times.

In the example shown the part of the program repeated, line 2Ø, had the same effect every time it was used, you saw **"WE ARE DOOMED"** every time. It is much more useful to make the repeated part do something slightly different each time by mentioning the variable in the **FOR** and the **NEXT** in it. To make this clear try this program.

```
1Ø FOR N=1 TO 1Ø
2Ø PRINT N
3Ø NEXT N
```

Lines 1Ø and 3Ø are the same as before so they cause line 2Ø to be used ten times but this time line 2Ø **PRINT**s the number in N. The way a **FOR — NEXT** loop works is to increase the number in the control variable, in this case N, each time the computer gets to the **NEXT.** This means that the first time line 2Ø is obeyed the number in N is 1, the next time it is 2, the next time 3 and so on until it reaches 1Ø. This explains why the numbers 1, 2, 3 . . . 1Ø were **PRINT**ed by the program.

We have, up to this point, used a control variable which started at one. There is no particular reason for this. Try

```
1Ø FOR A=1Ø TO 2Ø
2Ø PRINT A,5★A
3Ø NEXT A
```

The numbers printed by this program are the number in A and five times that number. Line 1Ø causes this number to run from ten to twenty. Five times the number will be 5Ø, 55, 6Ø, 65 and so on up to 1ØØ. The program prints a five times table. Notice that there are eleven lines printed, not ten. '1Ø TO 2Ø' includes both these numbers so there are eleven of them.

This should enable you to understand the following program which once again prints our petrol price table.

```
1Ø PRINT "PETROL PRICE TABLE"
2Ø PRINT
```

```
30 PRINT "GALLONS", "COST"
40 FOR G=1 TO 20
50 PRINT G,G★1.84
60 NEXT G
```

I have made the program print the cost of each number of gallons from one to twenty. You might like to make it print a table for petrol at any price rather than just using £1.84 every time. You would need to start with an **INPUT** to ask the user for the price of one gallon.

The numbers in the **FOR** instruction which tell the computer the bottom and top numbers for the control variable can themselves be variables. Below is a version of the program which asks the user, when the program is **RUN,** to type how many gallons the table is to go up to.

```
10 INPUT "HIGHEST NUMBER OF GALLONS";H
20 FOR G=1 TO H
30 PRINT G,G★1.84
40 NEXT G
```

Line 10 asks for the top number of gallons and puts it in H. In line 20 the **FOR** contains H as the highest number to use so you can make the program print whatever number of lines you like.

I expect you will be able to predict the results from the next program before you **RUN** it.

```
10 INPUT "FIRST NUMBER";F
20 FOR N=F TO F+10
30 PRINT F,F★F
40 NEXT N
```

It illustrates the fact that you can use F+10 as the top limit for the number in N in the loop controlled by the **FOR** instruction.

This program always prints eleven lines on the screen each line containing a number and the result of multiplying the number by itself. **RUN** it several times using different answers to the **FIRST NUMBER** question to convince yourself that you understand what is happening. In fact, you could put anything the computer could work out as the limits, though you may not yet see a reason for wanting to.

STEP

The **FOR — NEXT** system for making loops is even more versatile when you know about an optional extra which can be included. To make a table to convert the Fahrenheit temperatures in a recipe book to the Centigrade ones on the dial of the oven one might use a program like this.

```
10 FOR F=200 TO 800
20 LET C=(F—32)*5/9
30 PRINT F,C
40 NEXT F
```

If you use this program it tries to **PRINT** 601 temperatures, first 200 degrees then 201 degrees and so on up to 800. For cooking you would be quite happy with a table going up in steps of 50 rather than 1 and you could get it like this.

```
10 FOR F=200 TO 800 STEP 50
20 LET C=(F—32)*5/9
30 PRINT F,C
40 NEXT F
```

Only one alteration has been made, the addition of **STEP 50** after the 800 in line 10. This causes the number in F to increase not by one as it normally would but by 50 so that the numbers used will be 200, 250, 300 and so on, which is much more useful.

The step does not have to be a whole number. One might want a table going up in steps of a half or any other fraction. The following program makes a table for converting distances in miles from one to ten to Kilometres but does it in steps of half a mile. One mile is 1.8 Kilometres.

```
10 PRINT "MILES","KILOMETRES"
20 FOR M=1 TO 10 STEP 0.5
30 PRINT M,M*1.8
40 NEXT M
```

You could use the same approach to make a table for converting Kilometres to miles. One Kilometre is 0.625 miles. Try the effect of using different steps. You could get the program to ask you the top and bottom limits and the step to be used first, the limits and the step would have to be variables.

Sometimes we need the control variable to decrease rather than increase. This can be done by making the step a negative number.

The program

```
10 FOR N=10 TO 1 STEP —1
20 PRINT N
30 NEXT N
```

prints the numbers ten to one in 'rocket count down' fashion.

The rest of this chapter will consist of examples of how what you have learned may be used. Loops have applications which may well not have occurred to you. I'm afraid some of them may seem rather artificial but the ideas will be useful when you incorporate them into larger programs.

A 'NUMBER CRUNCHER'

All programs so far have apparently worked instantly. You type **RUN** and as soon as you press **RETURN** you see at once what the program does. The computer is fast but its operations are by no means instantaneous. Now that we know about loops we can give the computer plenty to do so we will have to wait for it to finish. Consider this program

```
10 LET S=0
20 FOR N=1 TO 50
30 LET S=S+N
40 NEXT N
50 PRINT S
```

The program works out the total of

1+2+3+4+5+. . . . .+47+48+49+50

Line 30 was used to add the number in N onto the number in S and this happened 50 times with N running from 1 to 50. Since S started off at 0 it will contain the total after fifty additions. Run the program using different numbers in place of the 50 to familiarise yourself with how long things take. This idea of adding by using a running total will be useful later.

DELAYS

It is sometimes necessary to make the computer wait for a fixed length of time between using one part of a program and the next. If you had printed some instructions on the screen you would want to give the user time to read them before moving on to the next part of the program. You can make the computer wait by using a **FOR — NEXT** loop which does nothing. Try this.

```
10 PRINT "STARTING"
20 FOR D=1 TO 100
30 NEXT D
40 PRINT "FINISHED"
```

You will find that by using a different number from 100 in line 20 you can vary the length of the delay. You might find it necessary to use a number as big as 1000 to make the delay suitable.

NESTED LOOPS

You can put one **FOR — NEXT** loop inside another. Suppose you wanted to **PRINT** a word across the screen like this.

```
GORDON
    GORDON
        GORDON
            GORDON
                GORDON
```

Precisely how many will fit depends on the width of the screen so I will use numbers that work with mine. You could do it like this.

```
10 FOR L=1 TO 14
20 FOR S=1 TO L
30 PRINT "     ";
40 NEXT S
50 PRINT "GORDON"
60 NEXT L
```

The **FOR** and **NEXT** in lines 10 and 60 make the computer repeat lines 20 to 50 fourteen times. I have chosen the letter L for the control variable and made it run from one to fourteen because I wanted the program to print fourteen lines. Each line has to start with a number of spaces and the number of spaces has to increase for each successive line. These spaces are printed by the **FOR — NEXT** in lines 20 and 40. You will notice that you can **PRINT** blank spaces by putting them in quotation marks. I have put a semi-colon after the spaces in the **PRINT** because we wanted the next thing **PRINTed** to come immediately afterwards on the screen rather than on the next line as it otherwise would. To make sure we got more spaces on successive lines I made the control variable S run from 1 to L. This has the desired effect because L is increasing for each line. When the computer gets to line 5Ø the spaces have been printed so we can print the name.

7
Ifs and Thens

"It is the stars,
The stars above us, govern our conditions"

Shakespeare – "King Lear"

Anyone who has played chess against a machine will recall the uncanny feeling that the machine has a mind of its own, especially when it starts to win. You can't avoid the impression that the computer is thinking just as another person would. Programs which appear to think require the computer to make decisions and behave differently according to circumstances. Our programs up to this point have not used the decision making ability of **BASIC.**

IF — THEN

You need to know about two more **BASIC** words, **IF** and **THEN.** These are another pair which, like **FOR** and **NEXT,** must both appear. You can't have one without the other. Look at this program line.

```
200 IF A=Ø THEN GOTO 1ØØ
```

When the computer gets to this line one of two things will happen. If the number in A is Ø then the next line to be used will be line 1ØØ and work will continue from there but if the number in A is not Ø then the next line after line 2ØØ will be used. The computer's next action has been determined by the number in A.

The equals sign in A=Ø here has a different meaning from that we have used so far. In

```
LET A=Ø
```

the equals sign tells the computer to put a number into A, but in

```
IF A=Ø THEN . . .
```

the equals sign is asking the computer to compare the number in A with Ø. A=Ø here is not an instruction to do something but a condition which may or may not be true. The **IF** tells the computer to work out whether it is true and the **THEN** tells it what to do if it is.

Many programs use a 'menu' of the Chinese Restaurant variety to ask the user what he or she wants to do – you make a choice by numbers. A program might start like this.

```
1Ø PRINT "TEMPERATURE CONVERTER"
2Ø PRINT "DO YOU WANT TO CONVERT:"
3Ø PRINT " 1.FAHRENHEIT TO CENTIGRADE"
4Ø PRINT " 2.CENTIGRADE TO FAHRENHEIT"
5Ø INPUT "NUMBER PLEASE";C
6Ø IF C=1 THEN GOTO 1ØØ
7Ø IF C=2 THEN GOTO 2ØØ
```

At line 1ØØ there would be a program to do the first job and at line 2ØØ a program to do the second.

When you write programs using **IF** in this way you have to take care to think about what will happen if the condition is not true as well as making sure the correct thing happens if it is. In the example above you would have to think about what happens if the user types a 3 as his answer to the **NUMBER PLEASE?** question. Neither condition will be true in lines 6Ø and 7Ø so the computer will obey the instruction in the next line after line 7Ø.

If this is line 100 the machine behaves just as if 1 had been typed in answer to the question. Possibly the best thing to do here is to assume that the 3 must be a mistake and give the user another chance to make a choice. You could do this with a line 80 which simply said

```
80 GOTO 20
```

This would mean that the question would be repeated until either 1 or 2 were given as the answer. This results in the following program. I suggest that you type it and **RUN** it.

```
10 PRINT "TEMPERATURE CONVERTER"
20 PRINT "DO YOU WANT TO CONVERT:"
30 PRINT " 1.FAHRENHEIT TO CENTIGRADE"
40 PRINT " 2.CENTIGRADE TO FAHRENHEIT"
50 INPUT "NUMBER PLEASE";C
60 IF C=1 THEN GOTO 100
70 IF C=2 THEN GOTO 200
80 GOTO 20
100 INPUT "CENTIGRADE";C
110 PRINT C;" C =";32+C*9/5;" F"
120 GOTO 220
200 INPUT "FAHRENHEIT";F
210 PRINT F;" F =";(F-32)*5/9;" C"
220 END
```

You will see a new **BASIC** word **END** in line 220. It was necessary to prevent the computer from doing line 200 after printing an answer in line 110 as it normally would. **END** tells the computer to stop work on the program.

CONDITIONS

Other conditions exist as well as equals.

> means "is bigger than"
< means "is less than"
<> means "is not equal to"

On each side of these signs you have to put the numbers to be compared. These may be actual numbers like the Ø, 1 and 2 mentioned above, variable names like the A or anything the computer could work out. You could, if you wanted to, use a condition like

```
IF 2★B <> A+1 THEN . . .
```

After the **THEN** in these instructions the computer needs to be told what to do if the condition is true. Up to now I have used **GOTO** as the thing to be done but any instruction may be used as in

```
IF N=B THEN LET N=Ø
```

or

```
IF R<Ø THEN PRINT "RATE OF INTEREST MUST BE
POSITIVE"
```

or

```
IF M<Ø THEN END
```

STOP CODES

It is very often useful to set a program to repeat something until you decide you have had enough. As an example suppose you had a list of amounts of money in pounds which each needed to be changed into French Francs. You won't want to keep typing **RUN** for each new number of pounds. The solution is to use a special number like Ø to show that you have reached the end of the list. Suppose £1 is worth 11.75 Francs. This program will do the job.

```
10 PRINT "POUNDS TO FRANCS AT 11.75"
20 PRINT
30 INPUT "HOW MANY POUNDS";P
40 IF P=0 THEN GOTO 80
50 LET F=P*11.75
60 PRINT P;" POUNDS = ";F;" FRANCS"
70 GOTO 20
80 END
```

Line 40 checks whether the number of pounds typed, in P, is 0. If it is not the rest of the program converts it to Francs and prints the result, otherwise line 80 is reached so the program stops. Notice that line 70 sends the computer back for a new number of pounds after each conversion.

MUGPROOFING

Almost all useful programs involve taking different action according to circumstances so **IF** and **THEN** are very frequently used. One application you may not have thought of is to protect the user from himself by checking that what is typed in answer to **INPUT** instructions is sensible. Good programs are written on the assumption that errors by the user are normal and take precautions against them. Consider a program which needs to ask the user for a date of birth. It might start like this.

```
10 PRINT "PLEASE TYPE YOUR DATE OF BIRTH"
20 PRINT "AS NUMBERS"
30 PRINT
40 INPUT "DAY";D
50 INPUT "MONTH";M
60 INPUT "YEAR";Y
```

The rest of the program might, for example, be concerned with astrology. As it stands there is ample scope for blunders by the user. The day should be a number in the range 1 to 31 but if say, 1ØØØ, were typed the program would use it. The month should be a number from 1 to 12 but any other number would be accepted. The same considerations apply to the year. This sort of thing gives computers a bad name.

The solution is to use **IF** to detect silly numbers and ask again if they are found. You will be able to follow the following modified version.

```
1Ø PRINT "PLEASE TYPE YOUR DATE OF BIRTH"
2Ø PRINT "AS NUMBERS"
3Ø PRINT
4Ø INPUT "DAY";D
41 IF D<1 THEN GOTO 4Ø
42 IF D>31 THEN GOTO 4Ø
5Ø INPUT "MONTH",M
51 IF M<1 THEN GOTO 5Ø
52 IF M>12 THEN GOTO 5Ø
6Ø INPUT "YEAR";Y
61 IF Y<1884 THEN GOTO 6Ø
62 IF Y>1983 THEN GOTO 6Ø
7Ø PRINT "USING THE DATE OF BIRTH"
8Ø PRINT D;"/";M;"/";Y
```

When this is **RUN** each question will be repeated until a sensible answer is given because after each answer has been typed **IF** is used to send the computer back to the line where the question is asked if the answer was outside a sensible range. I have allowed for users aged, in 1984, between 2 and 1ØØ. There is an error not detected, the user who types, for example Ø.5 as an answer. This will have to wait for a later chapter.

THINGS TO DO

If you ran the temperature program at the start of this chapter you will have perhaps been irritated by the need to type **RUN** and select option 1 or option 2 before each temperature. You could modify the program so that once a selection has been made the question **FAHRENHEIT?** or the question **CENTIGRADE?** is repeated and answers given until the user indicates an end by typing Ø as in the pounds and Francs program. In this case however the **GOTO** should send the machine to give the 1 or 2 choice again in line 3Ø. This creates a problem of how to let the user abandon the program when no more conversions are required. One way would be to treat any number other than 1 or 2 as indicating that the program is to stop. A change to line 8Ø will effect this.

According to my copy of the 1984-1985 PAYE coding guide, income tax is payable at 3Ø% on the first £146ØØ of taxable income, 4Ø% on that portion between £146ØØ and £172ØØ and at higher rates on any taxable income above this. Taxable income apparently means the balance after subtracting allowances. A single person has an allowance of £1785 and a married man an allowance of £2795. Thus a simplified system for working out someone's tax might go as follows.

1. Find their income.
2. Find the allowance (single or married).
3. Subtract allowance from income
   If the result is less than Ø then no tax and stop.
4. If the balance is less than £146ØØ then tax is 3Ø% of the balance and stop. (This is the end for most of us).
5. Subtract £146ØØ from the balance. Tax is 3Ø% of £146ØØ
6. If the new balance is less than £26ØØ (difference between £146ØØ and £172ØØ) then extra tax is 4Ø% of this balance and stop.
7. And so on. (I suspect that this part of the system will be of interest to few readers.)

This sort of procedure is ideal for programming by the use of **IF** and **THEN.** The program must first establish whether the user is single or married, by asking for a 1 or 2 answer and then ask for an annual income. A succession of **IF**'s will be needed to decide which part of the calculation is required. The form of the program is suggested by the procedure described. To work out 3Ø% of something you multiply it by Ø.3.

USING TAPES

By now you will be writing programs long enough to want to keep. Some people are lucky enough to be using a computer equipped with magnetic disc drives but most of us have to make do with cassette tapes. The instructions in **BASIC** for using them are different on the various machines but some advice is common to all.

Always use short tapes. It saves endless tiresome fiddling about looking for your program and winding from end to end if you use C12 or C15 rather than C30 or C45.

Always set the tape counter on the recorder to zero with the tape at the start and make a note of the counter position when you switch to record which must not be on the first few inches of tape. If you record on the non-magnetic leader part your program will be lost.

Save two copies of each program until you have established definitely that the system is working. Using a cassette recorder for programs can be completely reliable but it never is at first because there is so much scope for mistakes.

It sometimes helps to remove the playback lead when recording. Try this if your computer cannot find what you have recorded or if it tells you, with a message like **BAD DATA,** that it cannot understand it. You can establish whether anything has been recorded at all by listening to the tape with the leads removed from the recorder.

The most important advice is to persevere and not blame the equipment. Check and re-check that the leads are in the right holes and you have tried all variations of volume level.

8
Stringing Along

"The chief defect of Henry King,
Was chewing little bits of string"

Belloc. – "Henry King"

We have seen how you can use **BASIC** for calculations and how they can be organised. Perhaps the majority of computer applications involve working not mainly with numbers but with words. You may well have felt that programs up to this point coul have been improved if words could have been used rather than numbers for answering questions. We had to ask the user to select from a list of options by typing a number to show this choice. If we could ask for a word to be typed the computer would seem much more friendly to the user.

STRINGS

The word 'string' is used to mean a list of characters. In a **BASIC** program they are always enclosed in double quotation marks. We use the word 'string' rather than 'word' because all characters can be used as well as letters. (Except possibly double quotation marks). Thus possible strings include,

"THURSDAY"
"A"
"PNK 435 W"
"25/1/47"
"74LS123"

Strings, as you see, may consist of different numbers of characters. There is probably a maximum number of characters allowed in a string, usually 255, which is only rarely a problem but you may see an error message like **'LONG STRING ERROR'** later if you accidentally exceed it. It is also possible to have an 'empty' string with no letters in it. This is written "".

STRING VARIABLES

The word 'variable' was introduced as the name for a pigeonhole for keeping a number in. These are called 'numeric variables'. Also available are 'string variables' which are similarly like pigeonholes but are for keeping a string in. Because of the need to avoid confusion, both for you and for the computer, between string and numeric variables the names of string variables always have to have a dollar sign on the end. Thus Z stands for a number but Z$ stands for a string.

A surprisingly large number of the things which can be done with numbers can be done with strings. If a program starts with the line

```
10 INPUT "PLEASE TYPE YOUR NAME";N$
```

then, as with a similar **INPUT** using N rather than N$, the computer will wait for the user to type. When the user presses **RETURN** to show the end of what he or she wants to type the computer continues with the next line having put into the string variable N$ whatever was typed. For number **INPUT**s there are errors by the user which might be detected here by the computer but since N$ may contain any list of characters whatever is typed will be stored in N$. Later in the program the computer could address the user by name with a line like

```
250 PRINT "THANK YOU ";N$
```

and the name typed into N$ earlier would be printed. Notice the space between the end of **THANK YOU** and the quotation marks. We want the computer to **PRINT.**

THANKYOU GORDON

rather than

THANKYOUGORDON

Strings can be compared in the same way as numbers. A game playing program might start with

```
10 PRINT “DO YOU WANT TO BE BLACK OR WHITE”
20 PRINT
30 INPUT “CHOICE”;C$
40 IF C$=“B” THEN GOTO 100
50 IF C$=“W” THEN GOTO 200
60 PRINT “PLEASE TYPE B OR W”
70 GOTO 10
```

The **INPUT** in line 30 puts whatever the user types into the string variable C$. If this is a B or a W then lines 40 or 50 will cause the parts of the program which play the game to be done next. If the computer gets to line 60 then something different from B or W must have been typed so a helpful message is printed and line 70 causes the question to be asked again. This procedure will be repeated till a valid reply is obtained.

The signs > and < which we met earlier for comparing the sizes of numbers also have a meaning for strings but they work on alphabetical order for strings rather than on size as they do for numbers. Thus the condition

7<8

is true because 7 is less than eight, but

"SEVEN" < "EIGHT"

is not true because the word **SEVEN** does not come before the word **EIGHT** in alphabetical order.

'Alphabetical order' as far as the computer is concerned means more than it does in everyday language. All the characters on the keyboard have a position in it. In fact all characters have a number and these numbers are used to decide what order strings are in. The letters A to Z are usually numbered 65 to 9Ø. You may not see a use for this yet, but think about this improvement on the first example of this chapter.

```
1Ø INPUT "PLEASE TYPE YOUR NAME";N$
2Ø IF N$ < "A" THEN GOTO 5Ø
3Ø IF N$ > "Z" THEN GOTO 5Ø
4Ø GOTO 1ØØ
5Ø PRINT "TRY AGAIN"
6Ø GOTO 1Ø
```

The object of this is to prevent the program from continuing till the user has typed a word for his or her name which at least begins with a letter of the alphabet. If what were typed into N$ came before A or after Z then lines 2Ø or 3Ø cause line 5Ø to print the **TRY AGAIN** message. If on the other hand the computer gets to line 4Ø then what was typed must have been in the range A to Z, inclusive, so it might be a name. This would prevent the use of a name like ØØ7 or +++++.

ADDING STRINGS

Addition, subtraction, multiplication and division make sense with numbers but of these only addition may be used with strings. This will have many applications when you know more about **BASIC.** For the moment I hope you will find the following helpful. Suppose you use

```
10 INPUT "PLEASE TYPE YOUR NAME";N$
```

Later you will want the program to print N$ so as to address the user by name. If you make line 2Ø do this

```
20 LET N$=" "+N$+" "
```

then the effect of line 2Ø is to add spaces onto the front and end of the name typed and leave the new string in N$ so that later you can use

```
550 PRINT "GOODBYE";N$;"AND THANKYOU"
```

and spaces will appear between the end of **GOODBYE** and the name and between the end of the name and the start of **AND.** This improves the appearance of the sentence without having to worry about putting spaces in the **PRINT** line.

A PROGRAM

The applications of string which we have introduced so far have been as improvements to other programs. When you know more **BASIC** you will find them useful but here is a program which allows you to investigate how your computer deals with the concept of alphabetical order.

```
10 INPUT "FIRST WORD";F$
20 INPUT "SECOND WORD";S$
30 IF F$<S$ THEN GOTO 70
40 LET X$=F$
50 LET F$=S$
60 LET S$=X$
70 PRINT F$,X$
```

The program asks for two words, in lines 1Ø and 2Ø. Line 7Ø prints the same two words but line 3Ø first checks whether the words are in alphabetical order. If they are line 7Ø is used at once but if not lines 4Ø, 5Ø and 6Ø swap them over. This is done by copying the first word into the variable X$. The second is then copied into the space previously occupied by the first. The word in X$ is now copied into the space occupied by the second. The net effect of this is that they have exchanged positions. The use of a temporary variable in this way is often convenient for changing places.

Run the program and test it with pairs of words like for example **PRICE** and **PRITCHARD.** You will find that the usual alphabetical order applies. If you try ØØ7 and A you will find that digits come before letters in the computer's order. You might like to investigate the position of the various punctuation marks.

9
Arraying Things

"Solomon in all his glory was not arrayed like one of these."

Matthew 6:29.

A PROBLEM

'A firm employs thirty salespeople. Each month they are paid commission of 5% of that amount by which their sales for the month exceed the average of all thirty'. A program is required to calculate commissions each month.

How are we going to tackle this? The first thing to do is to make sure you understand the problem. The following steps will be needed.

1. Add together all 3Ø sales figures.
2. Divide by 3Ø to get the average.
3. For each salesperson compare their sales with the average.
4. If they have exceeded the average their commission is 5% of the difference, otherwise they receive nothing.

The first step will require the use of an **INPUT** instruction thirty times to get all thirty numbers into the computer. We will have to use a **FOR — NEXT** loop to do this because the only alternative would look like this.

```
1Ø INPUT "SALES NO 1";A
2Ø INPUT "SALES NO 2";B
3Ø INPUT "SALES NO 3";C
```

```
40 INPUT "SALES NO 4";D
```

and so on.

This approach is ruled out on three grounds. Firstly it is very tiresome to type thirty lines like this. Secondly it makes the program difficult to alter if the number of salesmen changes and thirdly we are going to run out of letters of the alphabet to use as variable names. The third objection can be overcome in most versions of **BASIC** by using longer variable names but the first two are sufficient to force us to use something like this.

```
10 FOR S=1 TO 30
20 INPUT M
30 NEXT S
```

This is useless as it stands. The user is not told what number the computer is expecting so errors like entering the same number twice are almost certain; and worse, since each number goes into M, the number previously in M will be replaced by the next number so that at the end of this **FOR —NEXT** loop, when S has reached 30, M will contain the sales of the thirtieth salesman and all the others will have been lost. We can get round the first problem like this.

```
10 FOR S=1 TO 30
20 PRINT "MONTHS SALES FOR NUMBER ";S
30 INPUT M
40 NEXT M
```

The user will be prompted by line 20 where the number of the required salesman will appear on the screen to ask for his sales to be typed, however we still haven't solved the problem of losing each number in M because the next input over-writes it by being stored in the same place.

You may well have guessed an answer to the problem. Since we need to know the total sales of all thirty salesmen so we can work out the average we could keep a running total as we **INPUT** them. This leads to a program starting like this.

```
10 LET T=0
20 FOR S=1 TO 30
30 PRINT "MONTHS SALES FOR NUMBER ";S
40 INPUT M
50 LET T=T+M
60 NEXT M
```

Line 10 starts the running total in T at 0, and line 50 adds each monthly sales number onto the number in T after it has been **INPUT**ed. As a result when we get to the line after line 60 the total of all thirty sales will be in T ready to be divided by thirty to get the average. This will be done with a line

```
70 LET T = A/30
```

and we are ready for the next task – working out the commission for each salesman.

Now we really are stuck. We have worked out the average sales but we have lost the sales figures. You can't decide what a salesman's commission should be unless you know what his sales were and the program doesn't know them because during the process of **INPUT**ing the figures it merely added each figure onto a running total and then lost it.

It is because of this problem – the need to have lots of numbers inside the computer simultaneously – that computer languages use the idea of an 'array'. I will explain the concept and then return to the salesman problem.

ARRAYS

Variable names used so far have been A, H, M and so forth. The array allows us to increase the number of variable available enormously by using for example A(20), X(79), P(600). The number in brackets indicates that what is intended is just one variable out of a number. The same program might use A(1), A(2), A(3), A(4) and so on up to A(1ØØØ) or even more if the computer has enough memory. You can do almost everything with one of the numbers of an array that you can do with an ordinary variable.

```
PRINT A(23)
```

and

```
INPUT A(23)
```

and

```
LET A(23)=3.1416
```

and

```
IF A(23)=1 THEN . . . .
```

all do what would have been done if A had been mentioned instead of A(23) except that the number used will be the one stored in a pigeonhole called A(23) rather than the one stored in a pigeonhole called A.

If you are going to use array variables in a program then you must warn the computer in advance of the fact that it needs to set aside space to store the numbers. You do this by putting a **DIM** instruction at the start of the program. This looks like this.

```
1Ø DIM A(1ØØ)
```

This warns the computer that A(1), A(2), A(3) up to A(1ØØ) may be used later. 1ØØ is called the 'dimension' of the array A, hence the **BASIC** word **DIM.** If the program tries to use, say, A(1Ø5), then there will be an error message such as **ARRAY SUBSCRIPT ERROR.**

How large an array is possible depends on the amount of memory in your computer. Try typing (without a line number)

```
DIM A(5ØØØ)
```

and you may well see a message like **OUT OF MEMORY.** If you have 48k RAM and an efficient system you may not see this unless you try something as big as

```
DIM A(1ØØØØ)
```

but there is an upper limit for every machine.

Why does this concept help? All we have done is to create as many variables as we like subject to the amount of memory in the computer. The reason that arrays are so useful is that the number in bracket can itself be a variable. I am lost in admiration of whoever thought of this first – without it computers would be very much less powerful. **RUN** this program.

```
1Ø DIM V(1ØØ)
2Ø FOR N=1 TO 1ØØ
3Ø LET V(N)=N★N
4Ø NEXT N
```

Nothing appears to happen because there are no **PRINT**s or **INPUT**s but if you do for example, type in:

```
PRINT V(21)
```

You will see the number 441. While the program was running it put into each of the hundred variables V(1) to V(1ØØ) a number obtained by multiplying the number in brackets by itself. Thus V(3) will contain 9, V(4) will contain 16, V(5) will contain 25 and so on up to V(1ØØ) which will contain 1ØØØØ: all this from a four line program. It is the use of V(N) in line 30 which is so effective. Because the **FOR - NEXT** loop causes

V(N) to be used 1ØØ times with a different number in N every time the small program does a tremendous amount of work.

You might like to try this program to get you used to using arrays in this way.

```
10 DIM Q(10)
20 FOR N=1 TO 10
30 INPUT "NUMBER PLEASE";Q(N)
40 NEXT N
50 FOR L=10 TO 1 STEP -1
60 PRINT Q(L)
70 NEXT L
```

The program **INPUT**s ten numbers and **PRINT**s them in reverse order. Line 3Ø is used ten times and stores the number it **INPUT**s in Q(1), Q(2), Q(3) and so on because N is running from 1 to 1Ø. Lines 5Ø to 7Ø print them in reverse order by making L run from 1Ø down to 1 and **PRINT**ing Q(L). I have deliberately chosen to use N for the first loop and a different letter, L, for the second to clarify a point which often causes confusion. Line 3Ø uses Q(N) and line 6Ø uses Q(L) but they both refer to the same numbers, the ten numbers **INPUT** by line 3Ø. The point is that N and L both contained numbers in the 1 to 1Ø range so both Q(N) and Q(L) stand for the same variable.

It would have been possible to obtain the same effect by using

```
10 DIM Q(10)
20 FOR N=1 to 10
30 INPUT "NUMBER PLEASE";Q(N)
40 NEXT N
50 FOR L=1 TO 10
60 PRINT Q(11-L)
70 NEXT L
```

Here I have made L in lines 5Ø to 7Ø run from 1 to 1Ø rather than from 1Ø to 1 but have changed line 6Ø to **PRINT** Q(11-L). Since L is running from 1 to 1Ø the result of taking it from 11 will make it run from 1Ø to 1. The contents of the brackets in an array variable may be anything that may appear on the right of a **LET** instruction. You could use A(2★N+1) for instance.

Now we are in a position to solve the problem posed at the start of this chapter. We must store the thirty numbers in an array so that when we come to work out the commissions they will still be available. The first thing to do is to ask the user for the thirty numbers which will be added onto a running total as it is typed.

The old version was

```
10 FOR S=1 TO 30
20 PRINT "MONTHS SALES FOR NUMBER ";S
30 INPUT M
40 NEXT M
```

But now we will use

```
10 DIM M(30)
20 LET T=0
30 FOR S=1 to 30
40 PRINT "SALESMAN NO ";S
50 INPUT M(S)
60 LET T=T+M(S)
70 NEXT S
```

The **DIM** has been put in at the start to make room for numbers for the thirty salesmen in the array M so that their sales figures can be **INPUT** by line 5Ø into M(1), M(2), M(3)... up to M(3Ø). This is done by using M(S) when S is running from 1 to 3Ø. The line which accumulates the running total in T now has to add M(S) onto the total rather than M.

Now we can work out the average in the same way as before with

```
80 LET A=T/30
```

but we will this time be able to work out the value of each salesman's commission because their sales are waiting in M(1), M(2), M(3)...

We had better put the results in two columns with the salesman's number in the first column and his commission on the right so the next thing to do is **PRINT** headings with

```
90 PRINT "SALESMAN","COMMISSION"
```

Now we need to run through all thirty comparing their sales with the average and working out the commission for those who are going to get any. The neatest way to do this is

```
100 FOR S=1 TO 30
110 LET C=0
120 IF M(S)<A THEN GOTO 140
130 LET C=0.05*(M(S)-A)
140 PRINT S,C
150 NEXT S
```

The **FOR — NEXT** makes sure that all thirty are dealt with by making S go from 1 to 30. In each case the commission will be put into variable C. Line 110 sets this to 0 every time, line 120 decides whether this man's sales are above average. If they are not the **GOTO** send the computer to line 140 where the salesman's number, in S, and his commission, in C, are printed. If the jump does not take places because he is going to get some commission then the number in C is changed by line 30 to 5% of the excess of his sales over the average.

AN IMPROVEMENT

The program now solves the problem posed at the start of this chapter. If you had to work out the commissions each month you could run it, the computer would ask you to type each salesman's sales and the list of commissions would be printed. This solves the problem but it is not the best possible program because it only works for thirty salesmen. A good program should be as versatile as possible so here is a final version which asks the user how many salesmen are to be used each time it is run.

```
 10 INPUT "HOW MANY SALESMEN";N
20 DIM M(N)
30 LET T=0
40 FOR S=1 to N
50 PRINT "SALESMAN NO ";S
60 INPUT M(S)
70 LET T=T+M(S)
80 NEXT S
90 LET A=T/N
100 PRINT "SALESMAN","COMMISSION"
110 FOR S=1 TO N
120 LET C=0
130 IF M(S)<A THEN GOTO 140
140 LET C=0.05*(M(S)-A)
150 PRINT S,C
160 NEXT S
```

The first line asks how many salesmen's figures are to be processed and puts the number into N. From there on every reference of 30 has been replaced with N. You would be able to test the program using a small number for N to make sure it worked before spending a long time typing numbers for the whole sales force.

10

Functioning Properly

"Chance favours only the prepared mind"

Louis Pasteur

Many people buy a personal computer with no other intention but to play games with it and a lucrative industry has grown up to supply games programs. The essence of many games is the element of chance. Without the random deal of the cards or fall of the dice many games would lose much of their appeal. All versions of **BASIC** provide some means of generating random numbers but they have serious applications as well as uses in game playing. If a computer system were going to be used to control traffic lights it would be highly desirable for reasons of safety to test the program thoroughly before lettig it loose on the roads. A program would be written using random numbers to simulate the random arrival of traffic at junctions so that the likely effect of the light control program on traffic flows could be investigated.

RANDOM NUMBER

A random number means a number chosen in such a way that you can't predict beforehand what it is going to be. Unfortunately there is more than one possible way in which your **BASIC** may give a random number. Probably the most common is where

```
LET X=RND
```

sets X to a number in the range Ø to Ø.999999 inclusive. It may be necessary to use

```
LET X=RND(1)
```

to produce exactly the same effect. The 1 in brackets in these versions is used by the computer to select where to start in a list of random numbers inside the computer. You don't need to understand this yet - just use **RND(1)** whenever you want a random number.

One other possibility is that

```
LET X=RND(100)
```

sets the contents of X to a random whole number in the range 1 to 1ØØ.

I suggest that you try this program to find out what your computer does.

```
10 FOR N=1 to 5
20 PRINT RND(1)
30 NEXT N
```

It is quite likely that this will produce a list of numbers like

```
0.897654
0.56432
0.126542
0.990077
0.101265
```

If the program doesn't work try removing the (1) then you should obtain a similar result. If not try making it (1ØØ), though any other largish number would do in place of the 1ØØ, and you will find that your computer uses the third method. You will have to bear this in mind reading what follows. I shall use the **RND(1)** convention, you will have to make the appropriate alterations.

In the line

```
LET P=RND(1)
```

we have introduced a new concept in **BASIC.** By using the word **RND** we have told the computer to produce a number by some method, in this case a random method. **BASIC** contains a number of similar methods for producing numbers all invoked by mentioning their three letter name. You can put the three letter name, as we have seen; in **PRINT** instructions and **LET** instructions and also wherever a number might have been used. These number creating devices are called 'functions'. A large number of functions are available though some have rather specialised uses.

Readers with mathematical inclinations will recognise such functions as **SIN, COS** and **LOG,** if you know about these you will know what to do with them – if not don't worry they won't be mentioned again in this book.

INTEGER

One function which will be of use is **INT.** This stands for 'integer part'. 'Integer' merely means 'whole number'. If you try

```
PRINT INT(3.4)
```

the number **PRINT**ed will be 3 because 3 is the whole number below 3.4. in the same way

```
PRINT INT(15.9)
```

will produce 15. Notice that **INT** creates not the nearest whole number to the number in brackets but the whole number

below it. This rule also applies to negative numbers so that

```
PRINT INT(-5.7)
```

will **PRINT** not -5 but -6 since -6 is **below** -5.7.

The number in brackets may be variable so that a line like

```
120 LET A=INT(P)
```

will set the contents of variable A to the whole number next below the number in P.

The random number produced by a dice is a whole number in the range 1 to 6 but the random numbers created by **RND** can be used to simulate that. Since **RND** creates a number from Ø to Ø.999999 multiplying it by 6 will give a number from Ø to a bit less than 6. This is progress but we need a whole number, we need to use **INT.** Since this would make a number from Ø to 5 we must add 1 to get a number from 1 to 6. The result is

```
LET D=1+INT(6*RND(1))
```

which puts into the variable D a number which could have been selected by a dice.

The following program could be used for a game played with two dice to dispense with the need to throw the dice.

```
10 LET A=1+INT(6*RND(1))
20 LET B=1+INT(6*RND(1))
30 PRINT A,B
40 INPUT "MORE";Z$
50 IF X$<>"N" THEN GOTO 10
```

Lines 1Ø and 2Ø create the two numbers and line 3Ø **PRINT**s them. Line 4Ø waits for the user to type something. Line 5Ø makes sure that if anything other than N for No was typed the process will be repeated. If you needed the total of the two dice, rather than the individual number, you could change line 3Ø to

```
30 PRINT A+B
```

You might like to use **RND** to write a similar program for 'head or tail' games rather than dice games. You need to print "**HEADS**" half the time and "**TAILS**" the other half. Since half the random numbers created by **RND(1)** are above Ø.5 and the other half are below (as near as makes no matter) you can compare a random number with Ø.5 to decide whether to print **HEADS** or **TAILS.**

A STATISTICAL PROGRAM

Dedicated 'Monopoly' players will have noticed that the numbers 2 to 12 which you get when you add the numbers on two dice do not occur with equal frequency. Seven happens most often and two and twelve are much less likely. This program simulates the throwing of two dice a large number of times and shows how often each possible total occurred. It illustrates an application of an array which we have not yet come across.

```
10 DIM F(12)
20 FOR N=1 TO 100
30 LET A=1+INT(6*RND(1))
40 LET B=1+INT(6*RND(1))
50 LET T=A+B
60 LET F(T)=F(T)+1
70 NEXT N
80 PRINT "TOTAL","OCCURENCES"
90 FOR M=2 TO 12
100 PRINT M,F(M)
110 NEXT M
```

The array F created by line 1Ø is going to be used to contain the frequency with which each of the possible totals came up. F(2) will hold the number of 2's, F(3) the number of 3's and so on. Line 2Ø makes sure that the dice will be rolled 1ØØ times. This is done by lines 3Ø and 4Ø and the total of the two is worked out by line 5Ø. Line 6Ø does the counting. By adding one to F(T) when T is one of the numbers 2 to 12 we will be recording the fact that the total was whatever number was in T. After this has been done the **NEXT** line causes the process to be repeated.

Lines 8Ø to 11Ø print the results. After printing headings for the columns we let M run from 2 to 12 printing M and F(M) each time. You will find the total 7 happens about six times as often as 2 or 12. If you experiment by changing the 1ØØ to bigger numbers, say 1ØØØ or 1ØØØØ you will have to be quite patient while you wait for the computer to produce its results.

ROUNDING

When you use **BASIC** to work out a number it very often produces its answer correct to a greater accuracy than you need. One kilogram is 2.2Ø5 pounds so to make kilograms to pounds conversion table you could use

```
10 PRINT "KILOS","POUNDS"
20 FOR K=1 TO 10
30 LET P=K*2.205
40 PRINT K,P
50 NEXT K
```

This has given three decimal places in the number of pounds which is unnecessary for buying potatoes. One solution which may occur to you is to change line 3Ø to

```
30 LET P=INT(K*2.205)
```

This has the disadvantage that it gives you the whole number below the correct value which is satisfactory for 17.1 pounds but for 17.9 pounds we would prefer 18 since it is closer. If you add Ø.5 before using **INT** the result will be what we need. Adding Ø.5 to 17.1 makes 17.6 so the **INT** of it is still 17 but adding Ø.5 to 17.9 gives 18.4 which has an **INT** of 18 as required. If you make line 3Ø

```
30 LET P=INT(2.205*K + 0.5)
```

then you get the number of pounds correct to the nearest whole number.

This idea can be extended to correct a number to any required accuracy. Here is a petrol price table program for petrol at 39.7 pence per litre.

```
10 PRINT "LITRES","COST"
20 FOR L=1 TO 20
30 LET C=L*39.7
40 NEXT L
```

This gives the cost of each number of litres in pence with a decimal point. We would prefer the numbers in £ and pence correct to the nearest penny. You can do it by changing lines 3Ø to

```
30 LET C=(INT(L*39.7+0.5))/100
```

The INT makes the number of pence into the nearest whole number and the /1ØØ changes the result to £ and pence.

ANOTHER MUGTRAP

Another use for the **INT** function is for testing whether or not a number is a whole number. Suppose a program needs to ask the user for a number and we wish to make sure that only valid numbers are accepted. If you were writing the ultimate football pool predictor program it would, I suppose, have to ask you for each Saturday's results. You would not want the program to accept, for example, a score of 2.7 goals because it is impossible, as would be a negative number or to accept say 1ØØ goals because it never happens. The part of the program which asks for a score would contain a line like

```
200 INPUT "SCORE";S
```

To prevent the acceptance of negative numbers the next line could be

```
210 IF S<0 THEN GOTO 200
```

and to prevent ridiculously large numbers you could use

```
220 IF S>30 THEN GOTO 200
```

but to prevent fractions you need to compare the number the user has typed with the whole number below it. This can be done by

```
230 IF INT(S)<>S THEN GOTO 200
```

The **INT** of a whole number is the same number and therefore acceptable so the condition will be true only for input numbers showing a fraction part. Thus if the number in S **is** a whole number the computer moves on to the line 23Ø, otherwise it will be sent back to line 2ØØ to ask again.

11
Reading Data

*"The months in succession come round
and you don't find two Mondays together"*

W. S. Gilbert – "Trial by Jury"

READ AND DATA

Up to now we have used two methods for providing information for the computer to use in programs.

The first is **INPUT**

```
INPUT A
```

and

```
INPUT A$
```

Put a number or a string of characters into A or A$

The second is **LET.** You could do

```
LET A=3.142
```

or

```
LET A$="CIRCLE"
```

to do the same job.

The reason for using one rather than the other is that **LET** puts the same information into the variable every time the program is **RUN** whereas **INPUT** causes the program to stop so that the user can type in the information. To give a specific example if you wrote a program to check your electricity bill every three months it would be sensible to use **INPUT** to give it the meter readings, which will be different every time, but you could use **LET** to set the price of one unit and the standing charge, which are fixed (we hope).

Often a program uses a large amount of 'built in' information. A program for printing a calendar for any year will need to use the number of days in each month. Suppose you decided to keep the month lengths in an array called M. There are 31 days in January, 28 in February, 31 in March, disregarding the leap year problem for the moment, so you might start the program like this.

```
10 DIM M(12)
20 LET M(1)=31
30 LET M(2)=28
40 LET M(3)=31
50 LET M(4)=30
```

and so on.

This means typing twelve very similar lines. When you find yourself doing this there is usually a neater way using **FOR** and **NEXT.** You might think of this.

```
10 DIM M(12)
20 FOR N=1 TO 12
30 INPUT M(N)
40 NEXT N
```

It is not a solution because using **INPUT** means that we will have to type all the month lengths every time we **RUN** the program. We need two new **BASIC** words, **READ** and **DATA.**

The start of the program could be,

```
10 DIM M(12)
20 FOR N=1 TO 12
30 READ M(N)
40 NEXT N
50 DATA 31,28,31,30,31,30
60 DATA 31,31,30,31,30,31
```

READ puts a number into the variable mentioned after it just like **INPUT** does but differs in that the number is not taken from the keyboard. Instead the computer looks for a **DATA** statement in the program and takes the number from there. A **DATA** statement, there are two examples in the program above, consists of the word **DATA** followed by a list of numbers separated by commas. When each number has been used the computer remembers that it has been used so that next time **READ** occurs it takes the next unused number. The **DATA** statements are used in order of their line numbers. You will notice that I put the lengths of the twelve months of the year into two **DATA** statements. This has the advantage over using only one that if you make a mistake it is easier to put it right.

To get yourself used to the idea I suggest that you use the following program which was written for changing speeds in miles per hour into Kilometres per hour. The interesting speeds are the various speed limits in force.

```
10 READ M
20 LET K=M*8/5
30 PRINT M,K
40 GOTO 10
50 DATA 30,40,50,70
```

The **GOTO** in line 40 sends the computer back to line 10 so that the job will be repeated but because the computer remembers that the number 3Ø in the **DATA** has been used it will read the number 4Ø into M next time. Thus you see a table showing the four speeds in both units. When the computer tries to make a fifth **READ** it finds that no **DATA** remains and the program stops, giving you a message like

```
OUT OF DATA IN 1Ø
```

because line 1Ø was the **READ** when no more data could be found.

STRING ARRAYS

The calendar program mentioned above would need to use the names of the months as well as their lengths. Up to now we have used arrays of numbers, like the lengths of the months, but not of strings. You can use string arrays in just the same way provided you remember that the names have to have a dollar sign like the names of ordinary string variables. These lines show how.

```
1Ø DIM M$(12)
2Ø FOR N=1 TO 12
3Ø READ M$(N)
4Ø NEXT N
5Ø DATA "JANUARY","FEBRUARY","MARCH"
6Ø DATA "APRIL","MAY","JUNE"
7Ø DATA "JULY","AUGUST","SEPTEMBER"
8Ø DATA "OCTOBER","NOVEMBER","DECEMBER"
```

This puts the twelve names into M$(1) to M$(12) so that if, for example, you were to use M$(7) later it would contain JULY.

It is in the handling of strings that the versions of **BASIC** show the most annoying differences. You may find that your computer uses a slightly different convention in which the **DIM** statement needs to contain information not only on the number of strings in the array, which was twelve here, but also on how long each string is to be. In such **BASICs** all the strings in an array have to be the same length. This is not a great restriction because the right hand end of each one can be packed with spaces. If you are using a computer which works like this then the first line would have to be

```
10 DIM M$(12,9)
```

This tells the computer that M$ is to contain twelve strings and that their length is to be nine characters. The longest month, SEPTEMBER, has nine letters so they will all fit.

We can use the ideas of this chapter to make an attempt at a calendar printing program. The idea is to ask the user two things, whether it is a leap year and the day on which the first of January falls, and use this to list all 365 (or 366) days. It is possible to calculate both items of information from the number of the year but I want to keep things simple and concentrate on how the arrays work. The program starts by setting up arrays to contain the names of the months, the names of the days and the lengths of the months. Here is the first part.

```
10 DIM M$(12),D$(7),M(12)
20 FOR N=1 TO 12
30 READ M$(N)
40 NEXT N
50 DATA "JANUARY","FEBRUARY","MARCH"
60 DATA "APRIL","MAY","JUNE"
70 DATA "JULY',"AUGUST","SEPTEMBER"
80 DATA "OCTOBER","NOVEMBER","DECEMBER"
90 FOR N=1 TO 7
100 READ D$(N)
110 NEXT N
120 DATA "SUNDAY","MONDAY","TUESDAY"
130 DATA "WEDNESDAY","THURSDAY","FRIDAY"
140 DATA "SATURDAY"
150 FOR N=1 TO 12
```

```
160 READ M(N)
170 NEXT N
100 DATA 31,28,31,30,31,30
190 DATA 31,31,30,31,30,31
```

There is only one new point here. In line 10 three arrays have been mentioned in one line. You can tell the computer about as many arrays as you like after the word **DIM.** Next we must sort out the problem of the leap year.

```
200 INPUT "IS IT A LEAP YEAR (Y/N)";A$
210 IF A$="N" THEN GOTO 250
220 IF A$="Y" THEN GOTO 240
230 GOTO 200
240 LET M(2)=29
```

This asks the question and according to whether Y or N is typed does or does not alter the length of February, stored in M(2), to 29. If the reply is not Y or N the question will be repeated until it is.

Now we must find out which day the year starts on. We will need a number for this but will have to ask the user to type one of the day names so it will be necessary to check what the user types against the seven names in D$ to find which position in D$ it occupies. If what was typed is not found the user will have to be asked to try again.

```
250 INPUT "WHICH DAY IS JAN 1";X$
260 LET P=0
270 FOR N=1 TO 7
280 IF X$=D$(N) THEN LET P=N
290 NEXT N
300 IF P=0 THEN GOTO 250
```

P is going to hold the number in the week, 1 for Sunday, 2 for Monday and so on, of the day the user typed. Line 260 sets it to 0 so that if the computer gets to line 300 without having matched the user's answer, in X$ with one of the correct names the question will be asked again because P will still be Ø. If on the other hand a match was found by line 28Ø P contains the required number.

Now we can **PRINT** the calendar. It will be necessary to use a loop twelve times to run through all twelve months. For each month the number of lines printed will be the number of days in that month. To get the day names right P will be increased by one every time we print a day but if P gets above seven, because a Saturday has been printed, then P will need to be changed to 1 because the next day will be a Sunday.

```
31Ø FOR N=1 TO 12
32Ø PRINT
33Ø PRINT "— — — — — — — ";M$(N);" — — — — — — —"
34Ø PRINT
35Ø FOR S=1 TO M(N)
36Ø PRINT D$(P),S
37Ø LET P=P+1
38Ø IF P>7 THEN LET P=1
39Ø NEXT S
4ØØ NEXT N
```

You will notice that this starts with a **FOR** and ends with a **NEXT** to go through all twelve months. For each month the name of the month is printed first, with some dashes and blanks to improve its appearance, then there is another **FOR — NEXT** to run through all the days of the current month. The upper limit on the **FOR — NEXT** is M(N) to make sure the correct number of days is printed. The **PRINT** line has to print the name of the day, which is one of the names in D$, and the number of the day in the month, which is in S because S is the number in the **FOR — NEXT** loop running from one to the month length.

I hope you will type in the calendar program and understand it. It would be wild optimism to expect however that it will work first time. It is very difficult to avoid typing errors in a forty line program and even an apparently slight error can cause a program to fail so check very carefully because the error message from the computer may not be very helpful. You may have to dip into your machine's manual to look for details on how precisely the string arrays work in your **BASIC.**

Don't forget that you can use **PRINT** to find out how far the program got before things went wrong. Suppose you type **RUN** and the machine says

```
OUT OF DATA IN 160
```

Looking at line 16Ø we see that the computer was trying to **READ** a number into M(N). It looked for a number in a **DATA** statement but they had all been used up. It would be sensible to use

```
PRINT N
```

to see how many numbers it did find. Almost certainly you will see 12 indicating that it has found eleven numbers but couldn't find a twelfth. Possibly you have missed one of the twelve month lengths but it could be that a comma is missing so that 3Ø,31 for example has been **READ** as 3Ø31 so that you have 3Ø31 days in November and no number left for December. The most effective way to solve such problems is careful thought but you must bear in mind that such errors as a missing comma can have surprising effects.

IMPROVEMENTS

A better version of the program will ask the user for the year number and do its own calculations to decide if it's a leap year (easy) and which day is the first of January (hard). Also we will be able to print each month in table form with the day names as column headings and the weeks in rows rather than just giving a list of the days.

12
Subroutines

"I shall return."

General Douglas MacArthur.

We have encountered one **BASIC** word, **DATA,** wnich has the unusual property that it is part of a program but does not give the computer anything to do - it is not an instruction. A line beginning with the word **DATA** merely sits in your program waiting for a line with **READ** in it to be used. All the other words cause something to happen, **DATA** just waits. If the computer gets to a **DATA** line while it is running the line is ignored.

REMARKS

There is another **BASIC** word with the same property, that is not doing anything, it is **REM,** short for **REMARK.** The point is that in all but the shortest programs you need to put messages to yourself to remind you what various parts of the program do or how they work. A line beginning with **REM** may contain anything you like, it will appear when the program is listed but have no other effect whatever. If a program contains a large number of good quality comments, as **REM** lines are called, then it is very much easier to understand it and improve it.

At the start of a program it is normal to include comments giving its name, purpose, author and similar information. To make the information stand out it is usual to put blank **REM** lines and underlining. Please remember that the sole purpose of this is so you will see it when the program is listed. It doesn't make the computer do anything and will be ignored when the program is **RUN.** Here is an example.

```
10 REM ******************************
20 REM
30 REM FOOTBALL POOL PREDICTOR
40 REM
50 REM (VERSION 3.4 19/5/84)
60 REM
70 REM COPYRIGHT GORDON BENNET 1984
80 REM
90 REM ******************************
```

Later in this program there will be sections to perform the various tasks the program will need to do. Each one should have a **REM** to indicate its purpose. They would in this case no doubt include such things as

```
200 REM *** ASK FOR SATURDAY'S RESULTS
```

and

```
400 REM *** TOTAL LAST SIX SCORES
```

and

```
1000 REM *** CALCULATE PROBABILITY OF DRAW
```

The fact that the author of a program has put in such **REM**s is of course no guarantee that the program will in fact do what he wanted but it will make it much easier to sort out the problems.

LONG PROGRAMS

All the programs in this book have been comparatively short. You can more or less take it all in at one reading. This is because they were each intended to illustrate a programming point rather than to perform a practical job. Useful programs tend to be much longer. A computer with 16K of memory can contain hundreds of lines of program and 48K or 64K machines are not uncommon. Writing a long program is a different problem from writing a short one. More planning is required and you have to test the different parts separately because mistakes are more difficult to find when programs are longer. One of the ways to facilitate this is the use of **REM** lines to separate the sections when you list the program but there is another - the subroutine.

SUBROUTINES

I suggest that you read this now without worrying too much about the details and return to it when you find you need to, as you will later. It is much easier to come to grips with a difficult idea under the pressure of necessity.

A subroutine is part of a program which is written to do a specific job and written in such a way that it can be used whenever required. There are two new **BASIC** words, **GOSUB** and **RETURN.** Suppose you decide to improve the way your program shows its results by printing the following.

```
100 PRINT
120 PRINT "____________________"
130 PRINT
```

The three lines print a line of dashes separated by blank lines.

You might need to do this many times at different places in a long program and you would find yourself typing the same three lines over and over again. To avoid this you could make the three lines into a subroutine like this.

```
1000 REM LINE OF DASHES SUBROUTINE
1010 PRINT
1020 PRINT "——————————————————————
———————————"
1030 PRINT
1040 RETURN
```

When in the program you needed to print a line of dashes you would use **GOSUB** to do it like this.

```
230 GOSUB 1000
```

and

```
550 GOSUB 1000
```

and

```
710 GOSUB 1000
```

When the computer gets to such instructions while the program is running these work like **GOTO** in that line 1000 will be used next. Why not use **GOTO** then? The reason is that when the dashes have been printed we want work to resume from the line after the **GOSUB.** If you use **GOTO** you will have to end the subroutine with another **GOTO** to send the computer back again. This won't do because you would have to know what line number to jump back to. Since we will wish to use the subroutine at different lines in the program the line to return to is different each time. **GOSUB** causes the computer to remember from which line it came when it jumps to a subroutine so that **RETURN** can cause it to jump back to the correct place. This system is intelligent enough for one subroutine to be able to invoke another by itself containing a **GOSUB** so that the computer has to simultaneously remember two places to **RETURN** to. It is even possible for a subroutine to call itself by containing a **GOSUB** to its own first line. This is called 'recursion'. If I were you I should avoid it till you are thoroughly familiar with using subroutines in a straightforward way.

To summarise, a subroutine is part of a program invoked, or to use a bit of computer jargon, 'called' by the word **GOSUB** followed by the number of its first line. It ends with the word **RETURN** which causes a jump back to the line *after* the line where the **GOSUB** was.

A NOTE FOR BBC USERS

The BBC and Electron computers have a version of the subroutine called a 'procedure'. These have several advantages one of which is that the programmer gives them names. This means that they can be called by name rather than having to remember the number of the first line as one would with the ordinary subroutine. The details will be found in the instructions under **DEFPROC, PROC** and **ENDPROC.** However **GOSUB** and **RETURN,** as described above, are also available.

PARAMETERS

The word 'parameter' is one which has been picked up from 'computerese' so that one can hear it in use by politicians and other experts every day. It is usually difficult to decide what they think it means but in programming it has a definite significance.

The subroutine used to introduce the idea earlier did the same job every time it was used. It printed a line of dashes. You will recall that the reason for using it was to avoid the need to write the bit of program which prints a line of dashes every time we needed the job done. Consider a program concerned with prices. It might frequently be necessary to print a price correct to two decimal places, to make it an exact number of pence. Here is a subroutine to do the job.

```
2000 REM SUBROUTINE TO PRINT X TO 2 DP
2010 PRINT (INT(100*X+0.5))/100
2020 RETURN
```

There is a snag. It is probable that the number you wanted printing would on one occasion be in a variable P, later in Q and at another place in the program in B. If you changed the subroutine to make it print P you would need a different subroutine later for printing Q. This defeats the object of having subroutines which was to enable us to use the same bit of program more than once.

This is why the subroutine was specifically written to print the number in X. You use it like this. If you want the number in P printed you use two lines.

```
340 LET X=P
350 GOSUB 2000
```

Later, when you want Q printed, you use

```
570 LET X=Q
580 GOSUB 2000
```

Whenever you want to print a number to two decimal places you copy it into X, using **LET,** and then call the subroutine with **GOSUB.** The function of the X is to carry a value into the subroutine. X is called a 'parameter' of the subroutine, in this case it is an input parameter.

OUTPUT PARAMETERS

In a program which asks the user to type something, and most of them do, one might use a subroutine to do the asking. As an example consider a program which might be used by a teacher for dealing with homework marks. At various places in the program the user would have to be asked to type a number which must be a whole number in the range 0 to 10 inclusive. Each time it will be necessary to check that the number typed meets these conditions so it would be a good idea to write a subroutine to do the job to avoid having to write the checking part over and over again. Here is a possible subroutine.

```
3000 REM TO GET A WHOLE NUMBER IN
3010 REM THE RANGE 0 TO 10 FROM THE USER.
3020 REM THE NUMBER IS RETURNED IN N
3030 INPUT "NUMBER PLEASE";N
3040 IF N=INT(N) THEN GOTO 3070
3050 PRINT "FRACTIONS NOT ALLOWED"
3060 GOTO 3030
3070 IF N > —1 THEN GOTO 3100
3080 PRINT "MUST BE POSITIVE"
3090 GOTO 3030
3100 IF N < 11 THEN GOTO 3130
3110 PRINT "MUST BE 10 OR LESS"
3120 GOTO 3030
3130 RETURN
```

The subroutine checks that the number is a whole number, is above -1 and less than 11. If it fails any of these tests the user is told why and asked to type the number again. This is repeated until an acceptable number has been typed in which case the **RETURN** is reached. The subroutine sends the computer back to the line after the **GOSUB** which called it with the number in the variable N. N is an output parameter used to return a value to the calling part of the program.

TO SUM UP

I hope you have been able to follow the logic of this chapter but I expect you have felt that the discussion of subroutines and parameters has been rather detached from reality. It will seem more positive when we use the ideas in programs. The way in which subroutines are used in **BASIC** is something which has attracted much adverse criticism by people who prefer the language **PASCAL** in which the programmer is more or less forced to structure a program as a collection of subroutines from the start. In **BASIC** you can do a lot of useful work without ever using a subroutine but, as I mentioned earlier, in some circumstances they are very useful.

13
Looking Back

"Them that asks no questions isn't told a lie."

Kipling – "A Smuggler's Song"

RECAPITULATION

You will recall that there are two sorts of variable in **BASIC,** number variables and string variables. Number variables are places for storing numbers while string variables are for storing words, or more correctly, string of characters, since all the symbols on a keyboard, including punctuation and digits as well as letters, are allowed. The names of string variables are distinguished by a dollar sign on the end of them. This means that a program may contain statements like

```
LET N=17
```

and

```
LET N$="EDWARD BEAR"
```

but that

```
LET N="CHRISTOPHER ROBIN"
```

would cause an error message because you can only put numbers into N, as would

```
LET N$=6
```

because you can only put words into N$.

On the other hand you could use

```
LET N$="SIX"
```

because **SIX** is a word in this sense rather than a number. You could also have

```
LET N$="ØØ7"
```

This puts the string ØØ7 into N$ rather than the number 7. I hope this isn't confusing! To clarify consider these two short programs.

```
10 LET N=7
20 LET M=5
30 PRINT N+M

10 LET N$="ØØ7"
20 LET M$="ØØ5"
30 PRINT N$+M$
```

If you think about them you will realise that the first prints the number 12 but the second prints the string ØØ7ØØ5.

STRING FUNCTIONS

We have used the concept of a function in **INT.** This function works on one number, its argument, and produces another number, a returned value. Thus in

```
LET X=INT(Y)
```

the argument is the number in Y and a value is returned in X. In this case both the argument and the returned value are numbers but there are functions for which one or both are strings.

The function **LEN,** short for length, returns a number, the number of characters in a string argument. Try this program.

```
10 INPUT "WORD PLEASE";W$
20 PRINT W$;" HAS ";LEN(W$);" LETTERS"
```

When you **RUN** it it asks you for a word. If you type **TUESDAY** it prints 7. If you type **ABRACADABRA** it prints 11. If you press return without typing anything else it should print Ø. Each time **LEN** is telling you the number of letters in the string W$. Notice that the argument, that is the contents of the brackets after the **LEN,** has to be a string. If you tried

```
PRINT LEN(7)
```

you would get an error message, possibly something like **TYPE MISMATCH,** because 7 is a number, but

```
PRINT LEN("7")
```

like

```
PRINT LEN("S")
```

will both cause 1 to be printed because both these strings are of length 1.

A QUIZ PROGRAM

We are going to develop a program which asks the user questions, accepts answers, and states whether the answers are right or wrong. You could call this a quiz or an educational program. One way to do this is to use the multiple choice system so that the questions look like this.

WHO IS THE PRIME MINISTER?

1. MR. WILSON.
2. MRS. THATCHER.
3. MR. LLOYD-GEORGE.
4. MR. BALDWIN.

ANSWER?

The user is expected to type one of the numbers 1, 2, 3 or 4. This approach is relatively easy to program but you might well think it less than satisfactory. GCE examining boards are increasingly using such tests so that they can be marked by computer but we will attempt something better.

If we ask the same question a number of possible answers is acceptable. For example **MRS. THATCHER, MARGARET THATCHER, MAGGIE THATCHER** or even **THE RT. HON. MARGARET THATCHER** should all be considered correct. What we need to do is search through the user's answer looking for **THATCHER.** If it is there then the answer is right. The same approach can allow for spelling mistakes to some extent. If we were to ask

WHO IS THE LEADER OF THE LABOUR PARTY?

then mis-spelling of **KINNOCK** such as **KINNOCH** or **KINNOK** could be taken as indicating that the user knew the answer. We need to search for a string, **KIN** say, and accept the answer if it is there. This will accept mis-spelling of the correct answer but reject answers such as **HATTERSLEY** or **FOOT.**

In order to do this we will store the questions in **DATA** lines in the program, following each question with the string which has to be present in an answer if it is to be marked right. For the two questions discussed so far this means that the program will contain the lines.

```
1000 DATA "WHO IS THE PRIME MINISTER"
1010 DATA "THAT"
1020 DATA "WHO IS LEADER OF THE
LABOUR PARTY"
1030 DATA "KIN"
```

This method makes it easier to increase the number of questions in the quiz.

BASIC VERSIONS

String handling is an area where **BASIC** versions differ. If you look at your machine's instruction book you may well find references to **LEFT$, RIGHT$** and **MID$.** These are fairly common but the popular Sinclair machines use a different method to produce the same results. I will deal with the first method first and describe the Sinclair variation afterwards.

MICROSOFT VERSION

LEFT$, RIGHT$ and **MID$** are functions which are like **LEN** in that they require a string as an argument. They also need a number. You will see that the names all have a dollar sign on the end. This indicates that the value returned is a string rather than a number as is the case with **LEN.**

The easiest way to show what they do (if you haven't guessed from the names) is to use the program.

```
10 LET A$="ABCDEFGH"
20 PRINT LEFT$(A$,3)
30 PRINT RIGHT$(A$,4)
40 PRINT MID$(A$,3,4)
```

Line 20 prints the string **ABC. LEFT$(A$,3)** means the leftmost three characters of **A$.** In line 30 the string printed is **EFGH** because **RIGHT$(A$,4)** is the rightmost four characters of **A$. MID$** is more complicated. The **MID$(A$,3,4)** in line 40 will print **CDEF,** a string obtained from **A$** starting at the third character and containing four characters.

SINCLAIR VERSION

The Spectrum keyboard does not have **LEFT$, RIGHT$,** or **MID$** on it but the machine is capable of producing the same effect. If you want part of a string you can get it out like this.

```
LET X$=A$(3 TO 7)
```

This sets **X$** to a string with five letters taken from **A$,** starting at the third and ending at the seventh. Run this program to try it out.

```
10 LET A$="SIR CLIVE SINCLAIR"
20 PRINT A$(4 TO 7)
30 PRINT A$( TO 5)
40 PRINT A$(13 TO )
```

Line 20 will print " CLI', the fourth to seventh letters of A$. Line 30 has no starting number in the brackets. The computer will assume that 1 is intended and print **'SIR C'.** Line 40 does not contain a finishing number so the last one will be assumed and **'NCLAIR'** is printed.

THE QUIZ

Now we can start the quiz program. A **FOR — NEXT** loop will be used to repeat the process of asking a question and checking the answer.

```
10 FOR N=1 TO 10
20 PRINT "★★★★★ QUESTION NUMBER ";N;" ★★★
★★"
30 PRINT
40 READ Q$,A$
```

These lines start the loop and print a heading for the question. Since the loop control variable is N the number in N will be 1 for the first question, 2 for the second question and so on. This is why line 2Ø tells the computer to print 'question number N'. I have made N run from 1 to 1Ø to allow for ten questions but this could easily be changed. Line 3Ø prints a blank line and line 4Ø **READ**s the question and the right answer from the **DATA** lines. You will notice that it is possible to **READ** two items with one **READ** line. This will put the first item of **DATA,** which will be a question, into Q$, and the second item found, an answer, into A$.

Now we must ask the question and give the user a chance to type an answer. This is easily done.

```
5Ø PRINT Q$;
6Ø INPUT T$
```

The question was in Q$ so line 5Ø prints it. I have put a semi-colon so that the cursor will be on the same line as the question when the user is given an opportunity to reply by line 6Ø.

We now have three strings in the computer. The question is in Q$ and we shan't need it again but we will need to compare the user's answer in T$ (we have used T for Try) with the right answer in A$. If the user's answer is shorter than the right answer there's no need to do any more since the answer must be wrong. This situation is dealt with by

```
7Ø IF LEN(T$)<LEN(A$) THEN GOTO
```

I haven't put a line number in after the **GOTO** because we don't know yet how long the rest of the program is going to be. It can be filled in later.

If the user's answer is equal to or longer than the right answer we will have to do some comparing. You might think of something like

```
IF T$=A$ THEN
```

but this won't do. The user's answer in T$ might be **MARGARET THATCHER** but the right answer in A$ is going to be something like **THAT.** As explained earlier this will cope with the possibility of alternative right answers and most mis-spelling.

It will be necessary to compare **THAT** with successive bunches of four characters from **MARGARET THATCHER** to see if any four match. **"THAT"** will first be compared with **"MARG"** then with **"ARGA",** then with **"RGAR",** then with **"GARE"** and so on. Eventually **"THAT"** gets compared with "THAT" and we decide that the user's answer was correct. Here I have of course been using the Right Honourable Lady merely as an example. We must write our program in such a way that it works for any strings.

You can do it like this.

```
80 LET F=0
90 FOR P=1 TO LEN(T$)—LEN(A$)
100 IF A$=MID$(T$,P,LEN(A$)) THEN LET F=1
110 NEXT P
```

The variable F which is set to 0 in line 80 is being used as what is called in programmer's jargon a 'flag'. Lines 90 to 110 do the comparing setting F to 1 if a match is found. Thus if we get to line 120 with F still equal to 0 we will know that there was no match.

Lines 90 to 110 require some explanation! P is running from 1 upwards so as to start with the first letter of T$. It does not however need to run up to the last letter of T$ because we are taking bunches of letters equal in length to the user's answer. Hence the top limit for P is **LEN(T$)—LEN(A$).** Line 100 compares A$ with such a bunch of letters. **MID$** has been used to extract the correct bunch. It has to start at the Pth letter and its length has to be the same as the length of A$. **MID$(T$,P,LEN(A$))** does this. You will recall that **MID(X$,3,4)** takes four letters of X$ starting at the third.

Now all we need to do is tell the user if the answer was right or wrong. These lines will do it.

```
120 IF F=Ø THEN GOTO 15Ø
13Ø PRINT "RIGHT"
14Ø GOTO 16Ø
15Ø PRINT "WRONG"
16Ø NEXT N
```

If F still contains Ø the answer was wrong so line 5Ø causes a jump to print the fact. Otherwise it must have been right and the computer will get to line 13Ø. The **NEXT** N in line 16Ø sends the computer back to line 1Ø to repeat the whole process with a new question.

All that remains is to fill in the line number we left blank in line 7Ø. It needs to jump to line 15Ø because the answer given was wrong so line 7Ø becomes

```
7Ø IF LEN(T$)<LEN(A$) THEN GOTO 15Ø
```

The program is now complete. All we need to do is invent some questions and try it out.

Here is the complete program.

```
1Ø FOR N=1 TO 4
2Ø PRINT "★★★★★ QUESTION NUMBER ";N;" ★★★★★"
3Ø PRINT
4Ø READ Q$,A$
5Ø PRINT Q$;
6Ø INPUT T$
7Ø IF LEN(T$)<LEN(A$) THEN GOTO 15Ø
8Ø LET F=Ø
9Ø FOR P=1 TO LEN(T$)—LEN(A$)
```

```
100 IF A$=MID$(T$,P,LEN(A$)) THEN LET F=1
110 NEXT P
120 IF F=0 THEN GOTO 150
130 PRINT "RIGHT"
140 GOTO 160
150 PRINT "WRONG"
160 NEXT N
1000 DATA "WHO IS THE PRIME MINISTER"
1010 DATA "THATCH"
1020 DATA "WHO IS LEADER OF THE LABOUR
PARTY"
1030 DATA "KIN"
1040 DATA "WHO WROTE 'PARADISE LOST'"
1050 DATA "MILT"
1060 DATA "WHO WAS THE SECOND HUSBAND OF
MRS. WALLACE SIMPSON"
1070 DATA "SIMPS"
```

I have included four questions. I suggest you supply your own.

The Sinclair version needs to use the bracket convention for extracting parts of strings. This only affects line 100 but I have included other minor alterations. Here it is.

```
 10 FOR N=1 TO 4
 20 CLS
 30 PRINT "★★★★★ QUESTION NUMBER ";N;" ★★★★★"
 40 PRINT
 50 READ Q$,A$
 60 PRINT Q$
 70 INPUT T$
 80 IF LEN (T$)<LEN (A$) THEN GO TO 160
 90 LET F=0
100 FOR P=1 TO LEN (T$)—LEN (A$)
110 IF A$=T$(P TO P+LEN (T$)—1) THEN LET F=1
120 NEXT P
130 IF F=0 THEN GOTO 160
```

```
140 PRINT "RIGHT"
150 GO TO 170
160 PRINT "WRONG"
170 NEXT N
1000 DATA "WHO IS THE PRIME MINISTER"
1010 DATA "THATCH"
1020 DATA "WHO IS LEADER OF THE LABOUR
PARTY"
1030 DATA "KIN"
1040 DATA "WHO WROTE 'PARADISE LOST' "
1050 DATA "MILT"
1060 DATA "WHO WAS THE SECOND HUSBAND OF
MRS. WALLACE SIMPSON"
1070 DATA "SIMPS"
```

14
Working Through

LORD SANDWICH: "You will either die at the rope's end, or of the pox."
MR JOHN WILKES: "That must depend on whether I embrace your lordship's principles or your lordship's mistress."

The best way for an author to annoy readers is to use expressions like 'this is left as an exercise for the reader' or 'it will be readily apparent that . . .'. In this chapter we will give answers to the problems that have been suggested in the text. In most cases the solution given is only one of a number of possibilities. We also include a number of programs intended as examples of how to use **BASIC** to achieve practical results.

THINGS TO DO WITH PRINT (Page 19)

The number of hours in a week is given by

```
PRINT 7*24
```

The distance the car has been driven is given by

```
PRINT 56102 — 46563
```

The number of inches in a metre can be found by

```
PRINT 100/2.54
```

To change 98 degrees Fahrenheit to Centigrade you can use

```
PRINT (98—32)★5/9
```

DAYS OF THE WEEK (Page 15)

You can list the days of the week like this

```
10 PRINT "SUNDAY"
20 PRINT "MONDAY"
30 PRINT "TUESDAY"
40 PRINT "WEDNESDAY"
50 PRINT "THURSDAY"
60 PRINT "FRIDAY"
70 PRINT "SATURDAY"
```

PRICE OF PETROL (Page 28)

This program will tell you how many litres of petrol at 40.5p you can get for each number of pounds.

```
10 LET P=40.5
20 PRINT 100/P
30 PRINT 200/P
40 PRINT 300/P
50 PRINT 400/P
60 PRINT 500/P
70 PRINT 600/P
80 PRINT 700/P
```

and so on.

(Page 33)

The improved petrol price table can be produced like this.

```
10 INPUT "PRICE OF ONE GALLON";P
20 LET C=P
30 PRINT N;" GALLONS COST ";C
40 LET N=N+1
```

```
50 LET C=C+P
60 GOTO 30
```

(Page 35)

A better version using **FOR** and **NEXT** might look like this.

```
10 INPUT "PRICE OF ONE GALLON";P
30 PRINT "GALLONS","COST"
40 FOR G=1 TO 20
50 PRINT G,G*P
60 NEXT G
```

INCOME TAX (Page 61)

The income tax program could be as follows. I have kept it shorter than it could have been by not going as far as the highest tax bands.

```
10 PRINT "TAXABLE INCOMES UP TO £21800 ONLY!"
20 INPUT "PLEASE TYPE YOUR ANNUAL INCOME";I
30 PRINT "ARE YOU 1. A MARRIED MAN"
40 PRINT " OR 2. NOT A MARRIED MAN"
50 INPUT "1 OR 2";R
60 LET A=0
70 IF R=1 THEN LET A=2795
80 IF R=2 THEN LET A=1785
90 IF A=0 THEN GOTO 30
100 LET T=0
110 LET B=I—A
120 IF B < 0 THEN GOTO 240
130 IF B > 14600 THEN GOTO 160
140 LET T=0.3*B
150 GOTO 240
160 LET T=0.3*14600
170 LET B=B—14600
180 IF B > 2600 THEN GOTO 210
190 LET T=T+0.4*B
```

```
200 GOTO 240
210 LET T=T+0.4*2600
220 LET B=B—2600
230 LET T=T+0.45*B
240 PRINT "INCOME ";I;" TAX ";T
```

HEADS OR TAILS (Page 83)

Here is the program for use in dice games. I have included ★'s in the **PRINT** lines to make the results stand out.

```
10 IF RND(1)<0.5 THEN GOTO 40
20 PRINT "★★★★★★ HEADS ★★★★★★★"
30 GOTO 50
40 PRINT "★★★★★★ TAILS ★★★★★★★"
50 INPUT "MORE";A$
60 IF A$ <> "N" THEN GOTO 10
```

PRINTING A CALENDAR (Page 90)

The improved calendar program follows. It asks the user which year is required and uses this to decide whether it is a leap year and to decide on which day of the week the first of January falls. The dates are printed in seven columns one for each day of the week. To make this happen it is necessary to print a number in an exact number of spaces for which two more functions came in useful.

The functions **STR$** and **VAL** are for changing a string to a number and a number to a string so that

```
LET N$=STR$(1234)
```

sets N$ to the string "1234". On the other hand

```
LET N=VAL("1234")
```

sets N to the number 1234. If you try to find **VAL** for a string which is not understandable as a number, for example

```
LET N=VAL("12A")
```

then N will be set to zero. This is used by lines 70 and 80 to make sure the user gives a genuine number as the year for which a calendar is required.

```
10 REM *** CALENDAR PRINTER
20 DIM M$(12),D$(7),M(12)
30 REM *** GET YEAR INSISTING ON A NUMBER
40 REM WITH FOUR DIGITS
50 INPUT "WHAT YEAR";Y$
60 IF LEN(Y$)<>4 THEN GOTO 50
70 LET Y=VAL(Y$)
80 IF Y=0 THEN GOTO 50
90 REM *** CALCULATE DAY OF FIRST OF
JANUARY
100 LET X=Y+INT(Y/4)+6
110 LET P=1+INT(7*(X/7—INT(X/7)))
120 REM *** READ DAY NAMES
130 FOR I=1 TO 7
140 READ D$(I)
150 NEXT I
160 DATA "SUN","MON","TUE","WED","THU","FRI","SAT"
170 REM *** READ MONTH NAMES AND LENGTHS
180 FOR I=1 TO 12
190 READ M$(I),M(I)
200 NEXT I
210 DATA "JANUARY",31,"FEBRUARY",28,"MARCH",31
220 DATA "APRIL",30,"MAY",31,"JUNE",30
230 DATA "JULY",31,"AUGUST",31,"SEPTEMBER",30
240 DATA"OCTOBER",31,"NOVEMBER",30,"DECEMBER",31
250 REM *** CHECK FOR LEAP YEAR & SET
260 REM FEBRUARY LENGTH TO 29 IF SO
270 IF INT(Y/4)<>Y/4 THEN GOTO 300
280 IF INT(Y/400)=Y/400 THEN GOTO 300
```

```
290 LET M(2)=29
300 REM *** MONTH PRINTING LOOP STARTS
HERE
310 GOSUB 600
320 FOR I=1 TO 12
330 PRINT "                ";M$(I);"             ";Y$
340 PRINT "   ";
350 REM *** PRINT DAY HEADINGS
360 FOR J=1 TO 7
370 PRINT D$(J);"   ";
380 NEXT J
390 PRINT
400 REM *** MOVE OUT TO POSITION OF FIRST
DAY
410 IF P=1 THEN GOTO 460
420 FOR K=1 TO P—1
430 PRINT "   ";
440 NEXT K
450 REM *** DAY LOOPS STARTS HERE
460 FOR K=1 TO M(I)
470 REM *** PRINT THE NUMBER IN K SO THAT
480 REM    IT OCCUPIES EXACTLY 5 SPACES
490 PRINT RIGHT$("    "+STR$(K),5);
500 LET P=P+1
510 REM *** REM IF SATURDAY MOVE TO SUNDAY
520 REM    ON A NEW LINE
530 IF P < 8 THEN GOTO 560
540 LET P=1
550 PRINT
560 NEXT K
570 GOSUB 600
580 NEXT I
590 GOTO 660
600 REM *** PRINT A LINE OF DASHES WITH A
610 REM    BLANK LINE ON EITHER SIDE
620 PRINT
630 PRINT "————————————————————————
```

```
——————————————————————————"
640 PRINT
650 RETURN
660 END
```

If you type this program you will have to be careful with the number of spaces between quotation marks, the number matters because they are used to keep the columns straight. The easiest way to get it right is to run the program and adjust it to correct the faults.

As listed above the calendar fits nicely on a 48 column screen. If your computer fits less than 48 characters on a line you will have to make each line narrower. I got it to fit the 32 column ZX Spectrum by reducing the number of spaces between quotation marks in lines 330, 340, 370 and 430. It was also necessary to print each date in four spaces rather than five. In Sinclair Basic this can be done by replacing line 490 with the lines

```
490 LET Z$="    "+STR$(K)
491 PRINT Z$(LEN(Z$)—3 TO );
```

A DATA PROCESSING PROGRAM

The following program is for maintaining a list of names, each with a number. It can sort the list either in order of the numbers or in alphabetical order of the names. A program of this type would be of use to a teacher who had to sort mark lists but I am sure you will think of your own applications.

The program uses a 'menu' system to allow the user to select what he or she wants to do. The options are selected by number. You will see that program is divided up into sections by the **REM** lines so that you will be able to work out how it works. You may wish to improve the program by including options to use the tape recorder to save the information and subsequently read it in again.

```
10 REM ******* EXAMPLE PROGRAM ON
SORTING ****
20 PRINT "THIS PROGRAM ASKS YOU FOR A"
30 PRINT "LIST OF NAMES EACH WITH A NUMBER"
40 PRINT
50 PRINT "IT WILL SORT THE LIST EITHER BY"
60 PRINT "NAME OR BY NUMBER"
70 PRINT
80 INPUT "HOW MANY NAMES";M
90 IF M > 2 THEN GOTO 120
100 PRINT "TRY AGAIN"
110 GOTO 80
120 DIM N$(M),N(M)
130 REM ****** ASK FOR NAMES AND NUMBERS
****
140 FOR I=1 TO M
150 GOSUB 1000
160 NEXT I
200 REM ****** OFFER MENU
*********************
210 PRINT "DO YOU WANT TO:"
220 PRINT "   1.LIST YOUR DATA"
230 PRINT "   2.SORT YOUR DATA"
240 PRINT "   3.AMEND"
250 PRINT "   4.GIVE UP"
260 PRINT "   5.START AGAIN"
270 INPUT "1/2/3/4/5";C
280 IF C<1 THEN GOTO 200
290 IF C>5 THEN GOTO 200
300 IF C<>1 THEN GOTO 400
310 REM *********** PRINT WHOLE LIST
*********
320 FOR I=1 TO M
330 PRINT I,N$(I),N(I)
340 NEXT I
350 GOTO 200
400 IF C<>2 THEN GOTO 600
```

```
410 REM ★★★★★★★★★★★ SORT THE LIST
★★★★★★★★★★★
440 INPUT "1.BY NAME OR 2.BY NUMBER";S
450 FOR I=1 TO M
460 FOR J=1 TO I—1
470 IF S<>1 THEN GOTO 500
480 IF N$(J)>N$(J+1) THEN GOSUB 2000
490 GOTO 520
500 IF S<>2 THEN GOTO 520
510 IF N(J)>N(J+1) THEN GOSUB 2000
520 NEXT J
530 NEXT I
540 GOTO 270
600 IF C<>3 THEN GOTO 700
610 REM ★★★★★★★★★ AMEND A RECORD
★★★★★★★★★★★★★★★
620 PRINT
630 INPUT "WHICH ONE";I
640 IF I<1 THEN GOTO 630
650 IF I>M THEN GOTO 630
660 GOSUB 1000
670 GOTO 200
700 IF C<>4 THEN GOTO 800
710 REM ★★★★★★★★★★★ USER WANTS TO GIVE UP
★★★★★★
720 INPUT "ARE YOU SURE";A$
730 IF A$="Y" THEN END
740 GOTO 200
800 REM ★★★★★★★★★★★ USER WANTS TO RESTART
★★★★★★
810 INPUT "ARE YOU SURE";A$
820 IF A$="Y" THEN GOTO 80
830 GOTO 200
1000 REM ★★★ SUBROUTINE ASKS FOR RECORD
NO I ★★
1010 PRINT
1020 PRINT "RECORD NUMBER";I
```

```
1030 INPUT "NAME",N$(I)
1040 INPUT "NUMBER";N(I)
1050 PRINT N$(I),N(I)
1060 INPUT "CORRECT";C$
1070 IF C$<>"Y" THEN GOTO 1010
1080 RETURN
2000 REM *** SUBROUTINE SWAPS RECORDS J &
J+1
2010 LET X$=N$(J)
2020 LET N$(J)=N$(J+1)
2030 LET N$(J+1)=X$
2040 LET X=N(J)
2050 LET N(J)=N(J+1)
2060 LET N(J+1)=X
2070 RETURN
```

HANGMAN

The following program plays the familiar hangman game. There seems to be some doubt as to the rules but in this version a letter is only inserted once if guessed correctly even if it occurs more than once in the word. If you try to guess the word and you guess wrongly you lose a life.

The **RESTORE** instruction in line 40 tells the computer to start using the **DATA** lines from the beginning rather than carrying on from where it has got to as it normally would. This is necessary because the user is given the opportunity to play the game again when it finishes.

```
10 REM ********* HANGMAN ***************
20 REM
30 REM **** SELECT WORD ***************
40 RESTORE
50 LET P=1+10*RND(1)
60 FOR I=1 TO P
70 READ W$
80NEXT I
```

```
100 REM ★★★★ PREPARE FOR NEW WORD ★★★★★
110 REM ★ SET UP BLANK WORD
120 LET B$=""
130 FOR I=1 TO LEN(W$)
140 LET B$=B$+"-"
150 NEXT I
160 REM ★ SET UP NUMBER OF LIVES
170 LET L=10
180 REM ★ SET UP WORKING COPY OF WORD
190 LET C$=W$
200 REM ★★★★ LETTER LOOP STARTS HERE ★
210 PRINT
220 PRINT "   ";B$;"    ";L;" LIVES LEFT"
230 INPUT "LETTER OR GUESS";G$
240 IF LEN(G$)>1 THEN GOTO 500
250 REM ★ SEARCH COPY OF WORD FOR GUESSED
LETTER
260 LET P=0
270 FOR I=1 TO LEN(C$)
280 IF G$=MID$(C$,I,1) THEN P=I
290 NEXT I
300 IF P>0 THEN GOTO 390
310 PRINT "SORRY - LIFE LOST"
320 LET L=L-1
330 IF L>0THEN  GOTO  200
340 PRINT "YOU ARE HANGED"
350 PRINT "   THE WORD WAS ★★★★";W$;"★★★★★"
360 INPUT "ANOTHER GO";A$
370 IF A$="Y" THEN GOTO 10
380 GOTO 999
390 REM ★ INSERT CORRECT GUESS IN BLANKS
400 LET T$=LEFT$(B$,P-1)
410 LET T$=T$+G$
420 LET T$=T$+RIGHT$(B$,LEN(B$)-P)
430 LET B$=T$
440 REM ★ REMOVE FOUND LETTER FROM COPY
450 LET T$=LEFT$(C$,P-1)
```

```
460 LET T$=T$+"★"
470 LET T$=T$+RIGHT$(C$,LEN(C$)-P)
480 LET C$=T$
490 GOTO 200
500 REM ★★★ USER HAS ATTEMPTED TO GUESS
THE WORD
510 IF G$<>W$ THEN GOTO 310
520 PRINT "    WELL DONE - CORRECT"
530 GOTO 350
1000 DATA "AMBIDEXTROUS","HAEMOPHILIA"
1010 DATA "INTERFERENCE","REACTIONARY"
101020 DATA "PARASITE","SUPERSTRUCTURE"
1030 DATA "SYMPHONY","OSTRICH"
1040 DATA "INDIVIDUAL","FREQUENCY"
1050 DATA "DOWNSTAIRS","CHICANERY"
```

MORTGAGE PROGRAM

The next program prints a table showing your position at the end of each year of paying a mortgage. The subroutine at line 1000 is concerned only with printing the money neatly and if you wished you could replace lines 290 to 330 simply with one line.

```
290 PRINT Y,I,A
```

The point is that most versions of **BASIC** will not print columns of figures with the decimal points under each other. This subroutine shows one possible way to improve the appearance of the numbers.

```
10 REM ★★★ PAYING A MORTGAGE ★★★★★★
★★★★★★★
20 REM
30 REM ★ ASK FOR DETAILS
40 INPUT "RATE OF INTEREST";R
50 INPUT "AMOUNT OF LOAN";A
```

```
60 INPUT "HOW MUCH TO PAY EACH MONTH";P
70 REM ★ START AT YEAR0
80 LET Y=0
100 REM ★ PRINT HEADINGS
110 PRINT "YEAR","INTEREST","MONEY OWED"
200 REM ★ CALCULATE POSITION AFTER THIS
YEAR
210 REM ★ CALCULATE YEARS INTEREST
220 LET I=R★A/100
230 REM★ ADD INTEREST AND SUBTRACT
PAYMENTS
240 LET A=A+I -12★P
250 REM ★ ADVANCE THE YEAR
260 LET Y=Y+1
270 REM★ PRINT ALL THESE TO THE NEAREST P
280 PRINT Y,
290 LET X=I
300 GOSUB 1000
310 LET X=A
320 GOSUB 1000
330 PRINT
340 REM ★ CHECK FOR END
350 IF A>0 THEN GOTO 200
360 PRINT "DEBT PAID OFF"
370 END
1000 REM ★★★ SUBROUTINE PRINTS X IN POUNDS
1010 REM ★★★ AND PENCE
1020 REM ★ ROUND X TO 2 DECIMAL PLACES
1030 LET X=(INT(100★X+.5))/100
1040 LET X$=STR$(X)
1050 REM ★ IF X IS A WHOLE NUMBER ADD A POINT
1060 IF X=INT(X) THEN LET X$=X$+"."
1070 REM ★ IF POINT IN RIGHT PLACE PRINT NOW
1080 IF MID$(X$,LEN(X$)-2,1)+"." THEN GOTO 1120
1090 REM ★ IF NOT ADD A 0 AND TRY AGAIN
1100 LET X$=X$+"0"
```

```
1110 GOTO 1080
1120 PRINT RIGHT$("          "+X$,10),
1130 RETURN
```

15
Winding Up

"This darned machine will drive me mad,
I wish that they would sell it.
It never does just what I want,
But only what I tell it."

Anon.

AIMS

This book claims, as promised at the beginning, to be only an introduction - but a very thorough one - to how to use **BASIC**. If you were a complete newcomer at the start but have read the text carefully and used the examples on your computer you will be able to use the common **BASIC** keywords easily and be able to spot the cause of the usual error messages. You will be able to understand the details of other people's programs and be able to adapt program listings from books and magazines to run on your machine provided they do not use non-standard language features which the author has regretably not seen fit to explain. You will have picked up some ideas about how to set about using the computer on your own problems but you would probably be apprehensive about starting some major programming job and worried that you have chosen an inappropriate and lengthy method when a simpler one is available. There are books to give advice but in this area precept is no substitute for experience. Use a computer whenever you get a chance.

Students often say, "I've learned **BASIC** but I don't seem to be able to write programs". The difficulty they, and possibly you, find is that understanding how the language works is not the same as being able to use it. If you were given a Swedish grammar and dictionary and told to translate today's 'Times' leader into Swedish by lunchtime you would be unlikely to produce something that ABBA would regard as acceptable. Using a language effectively requires experience and practice. You will find if you persist with **BASIC** and write simple programs whenever an opportunity arises that it will get very much easier. At first it is difficult to decide what is a simple problem to program and what is a difficult one. Here again only experience will help - but one useful rule is: "if it's hard, give up". I justify contradicting the "if at first you don't succeed ..." maxim by saying that all programs should be simple, or at least constructed of simple pieces joined together. If you find yourself lost in the complexity of what you are trying to do you must break it into smaller bits. A large program should consist of understandable chunks each no larger than you can see on the screen at once.

DEBUGGING

Inevitably you will write programs that don't work. There are two sorts of 'bug', a word used by computer jargon enthusiasts for any sort of mistake. The first sort will be detected by your computer when you try to run the program and it tells you that you have made a mistake in the syntax of **BASIC**. You will soon learn to avoid these. More difficult are the variety of bug which causes the computer to do something but not to do what you wanted. You told it to do something silly and it did. Here is a list of some common causes of trouble.

1. Inverted logic in conditions. Think hard about such lines as

```
IF X = 100 THEN ...
```

could you possibly have meant

```
IF X <>100 THEN ...
```

2. Forgetting to initialise variables. If as in, for example, the quiz program discussed earlier the variable S is used to hold the player's score one might put

```
LET S=Ø
```

early in the program to start the score at zero. If at the end the user is given a chance to play the game again then does S get reset to zero for the second attempt? If not the score will grow illogically large.

3. Reversed order in **LET**. The line

```
LET A$=Y$
```

copies the contents of Y$ into A$. The present contents of A$ are lost. It is terribly easy to get this the wrong way round.

4. 'Off by one' errors. The following program fragment was intended to use **READ** five times to get five values into the array W$

```
100 LET C=1
110 READ W$(C)
120 LET C=C+1
130 IF C<5 THEN GOTO 11Ø
```

In fact it only **READ**'s four words because when C is increased from four to five in line 12Ø the fact that it has reached five will prevent the bckward jump in line 13Ø.

5. Forgetting trivial cases. A program which works successfully for 'normal' numbers may collapse when confronted with Ø or 1. The following program fragment was intended to change a word (in W$) from capital to small letters.

```
2ØØ X$=""
21Ø FOR I=1 TO LEN(W$)
22Ø LET X$=X$+CHR$(ASC(MID$(W$,I,1))+ASC("a")
—ASC("A"))
23Ø NEXT A
24Ø LET W$=X$
```

It works except for the case where W$ happens to be an empty string, that is W$="". **LEN**(W$) will then be Ø so that when the **FOUR-NEXT** tries to extract the first letter of W$ it doesn't exist.

You will certainly make all the above mistakes and more! One can easily fall into the trap of becoming convinced that the computer has 'gone wrong' - it won't do what you want. This is a mistake, it's always your fault, and the satisfaction to be gained by getting your own programs to work is enormous. Good Luck.

Appendix

A list of common **BASIC** keywords.

This is a quick reference guide. It is not comprehensive and of necessity does not give full details. I have included a number of words which are not covered in the text of the book but which you may well encounter.

X and X$ are used here to stand for any variable and string variable.

[expression] stands for a number or something the computer can work out which produces a number.

eg

7

or

X

or

2★X

or

(-B+SQR(B⋆B-4⋆A⋆C))/(2⋆A)

[string expression] stands for a string or something the computer can work out which produces a string.

eg

"FRED"

or

A$

or

A$+"FRED"

or

"!"+Z$+"!"

ABS A function which returns it argument unchanged if it is a positive number but makes it positive if it is negative.

LET X=ABS([expression])

ASC A function with a string argument which returns a number. The argument should be only one character and the number returned is the number it has been given in what is called the **ASCII** code. This is the American Standard Code for Information Interchange which gave all characters a number for use in teleprinters. A is a number 65 and Z is number 9Ø.

LET X=ASC([string expression])

CHR$ A function which returns a single letter string and requires a number argument. It returns the character of which the number is the **ASCII** code. (See ASC)

CLEAR Is used by some **BASICs** to create space for string variables. If you get a message like 'out of string space' this is the word to look up.

CLS Clears the screen in some **BASICs**.

CONTINUE In some versions this word is used to re-start a program if you have interrupted it with a break key on the keyboard while it was running.

DATA Has no effect on the running program. Lines beginning with **DATA** exist to provide values for **READ** to use.

DIM This word warns the computer that you intend to use array variables so that it can have space ready.

```
DIM X([expression])
DIM X$([expression])
```

FOR Causes a portion of program to be used a number of times. It must be used in conjunction with a subsequent line beginning with **NEXT**

```
FOR X=[expression] TO [expression]
FOR X=[expression] TO [expression] STEP [expression]
```

GOSUB Tells the computer which line number to use next but to remember the line number of the **GOSUB** for use when **RETURN** is encountered.

```
GOSUB [line number]
```

GOTO Tells the computer which line number to use next.

```
GOTO [line number]
```

IF Causes the computer to decide what to do next according to whether a condition is true.

```
IF [condition] THEN [some other keyword]
```

INT A function which returns the next whole number below its argument, or its argument unchanged if the argument is already a whole number.

LET X=INT([expression])

INPUT Causes the computer to wait for the user to type.

INPUT X
INPUT X$

LEFT$ A function which returns a string and which requires a string argument and one number argument.

LETX$=LEFT$(Y$,I)
sets X$ to the leftmost I characters of Y$

LET Moves a value to a variable.

LET X=[expression]
LET X$=[string expression]

LIST Lists the current program. In most machines it may be followed by a line number to start at.

LPRINT This word is sometimes provided to **PRINT** on a printer, if you have one rather than on the screen.

MID$ A function which gives a string result and requires a string argument and two number arguments.

LET X$=MID$(Y$,I,J)
sets X$ to J characters of Y$ starting with the characters in position number I.

NEXT Used in conjunction with **FOR.**

NEXT X

PEEK A function provided because of the inadequacy of some versions of **BASIC.** Its argument is a number which has to be what is called an address and it returns the number the computer has stored in that address. **BASIC** was invented precisely so that you wouldn't have to do things like this.

PI This looks like a variable name but isn't. In some **BASICs** you can use it to provide the maths teacher's favourite number 3.14159.

POKE Allows you to over-ride **BASIC** and force the computer to store a number at a particular address inside itself. As in the case of **PEEK** it shouldn't be necessary. If you use it be careful. You can't do any physical damage but you might, for example, cause the keyboard to stop working and have to unplug the computer and start again.

> **POKE [address],[expression]**

PRINT Prints the print items following on the screen.

> **PRINT** [expression or string expression]
> **PRINT** prints a blank line

Optionally several items can be printed separated by commas to space them out or semi-colons to pack them together.

READ Puts values into variables taking the values from **DATA** lines.

> **READ [variable]**

Optionally several variables may be present separated by commas.

REM Has no effect whatever. Lines beginning with **REM** are only present so that they will be seen when the program is listed to provide information to the programmer.

RESTORE Causes the next **READ** to take values from the first **DATA** in the program rather than carrying on from where it has got to.

RETURN Causes the next line used to be the one after the last **GOSUB** used.

RIGHT$ A function which returns a string result and requires a string argument and a number argument.

> **LET X$=RIGHT$(Y$,I)**
> sets X$ to the rightmost I characters of Y$.

RND A function which returns a random number. An argument is required by some versions of **BASIC** to tell the computer where to start in its random number table. In most **BASICs** random numbers are in the Ø to 1 range.

RUN Causes the computer to follow the instructions of the program starting at the first. In most machines **RUN** may optionally be followed by a line number to start at.

STR$ A function with a number argument which returns a string. The returned string is what would appear on the screen if the number were printed.

> **LET X$=STR$([expression])**

SQR A function which returns the square root of its argument.

> **LET X=SQR([expression])**

The expression cannot have a negative value.

USR This is a function often provided to allow you to leave the safety of **BASIC** and use programs written in other languages. Avoid using it by accident.

VAL A function with a string argument which returns a number. The string should be one which would look like a number if printed, for example "231", though for practical purposes it will be a string variable. The returned value is the value of the number it looks like.

```
LET X=VAL([string expression])
```

If the string expression doesn't look like a number it returns zero.

INDEX

GETTING STARTED WITH BASIC
THE BEGINNERS GUIDE TO COMPUTING

If you have decided to begin 'Computing' and are too proud, or embarrased, to ask your friends, colleagues or your own children to help you then this book is for you.

Using 'BASIC', the common language of home computers, through examples such as calculating your mortgage payments, filing your address list, working out petrol consumptions, and much more, you will move steadily forward from beginner to confident user.

'now I can understand what my children are doing – and talking about'
Liz Bisset, mother

'I wish I had started with a book like this'
Andy Thomson, Sales Manager

'I could feel my fear of computers vanishing as I worked through the book'
Nina Duncan, student

ILLUSTRATION: GRAHAM THOMPSON
DESIGN: IVOR CLAYDON/BOB HOOK

PHOENIX PUBLISHING ASSOCIATES

ISBN 0-9465-7615-7
90000
9 780946 576159

£5.95

GETTING STARTED WITH BASIC JOHN PARRY