REWARD BOOK

$10.95

START WITH BASIC ON THE COMMODORE VIC 20

A FUN BOOK FOR A FUN COMPUTER!

Don Monro

Illustrated by Bill Tidy

Start with

BASIC

on the Commodore VIC 20

by Don Monro

Illustrated by Bill Tidy

RESTON PUBLISHING COMPANY, INC.
Reston, Virginia
A Prentice-Hall Company

First published 1982
by The Tiny Publishing Company Ltd,
16 Princes Gardens, London SW7 1NE

In memory of my father

Hector Monro

This edition of START WITH BASIC ON THE COMMODORE VIC 20 is produced by Reston Publishing Company, Inc., A Prentice-Hall Company, Reston, Virginia. It is not for sale in the United Kingdom and Eire.

ISBN: 0-8359-7071-X
0-8359-7070-1 (pbk.)

10 9 8 7 6 5 4 3 2
Printed in the United States of America.

CONTENTS

PREFACE

I have long wanted to write a real computer book for real people. The advent of the VIC 20, a real computer that real people can afford, and the encouragement of my colleague John Roberts have created the opportunity. As for the time, well, when you are training for your first marathon, you can't take a holiday, but you can write a book instead. The marathon took 3:57:01; the book a bit longer. Both were completed on 23 August 1981. I am grateful for the tolerance of Ann, Andrew, Douglas and Heather while all of this was going on.

Jan Williams has acted as editor for this book in her own time, and I am indebted to her for the splendid results. Bob Gleadow of Commodore (UK) kindly arranged for us to have a preproduction VIC despite the scepticism of some of his colleagues. This enabled first myself and then Malcolm Clarke to try out the examples after typesetting to ensure (we hope) that no serious errors get through. Thank you, Malcolm, and also Peter Lindsey for some useful observations on the graphics.

When I timidly approached the great Bill Tidy to ask if he would illustrate my book, I was surprised by his acceptance and delighted by the appropriate and amusing drawings, so promptly produced, which do so much to enliven the text. Thank you, Bill Tidy.

And so, Dear Reader, over to you...

One

WELCOME!

1 More than just a toy

All around us the computer revolution is taking place. It is the most important social and cultural change since the industrial revolution 150 years ago. Some people think that this revolution is taking mankind into a new historical age. It all began about 40 years ago with bulky, expensive monster computers that could do very little compared to modern machines. Since then the size and cost of computers has been falling faster and faster, recently because of the silicon chip which is at the heart of today's computers. The VIC 20 is a real computer which is capable of all the tasks that bigger, more expensive machines used to do. It is one of the very first real computers that most households can afford. Very soon the ability to use and program a computer is going to be a basic skill as important as reading and writing. If you have a VIC 20, you're a kind of pioneer; you and your family have a great educational opportunity.

It would be a great pity to miss out on this. It is all too easy to buy a VIC, play with it for a while using a few programs or games that someone else has written, and then let the whole thing be discarded like some forgotten toy. VIC is not a toy, it is a real computer. Since computer skills are already in great demand, you owe it to yourself to make the most of it. This book is intended to help unleash the computer experts hiding inside real people, by helping them learn to program a real computer (VIC) using a real programming language (BASIC).

2 Start with BASIC

2 What's so special about computers?

Nothing much. A computer is a machine that can do a few quite ordinary things, but it is fast and does not make mistakes. It does arithmetic, and always gets the answers right. It remembers things perfectly, and this means that it can remember the orders you give it and repeat them perfectly forever, or at least until you stop it. It can compare things and decide what to do as a result. These are very simple things. All the fuss about computers is because they are new. There is nothing difficult about them, only different.

3 What is BASIC?

Computers obey instructions. A computer program is a list of instructions given to a computer. These instructions are given in a language that people can understand, but which is adapted to the requirements of computers. Almost anyone can learn to program a computer, and BASIC is an excellent language for this purpose. The word BASIC stands for 'Beginners All-purpose Symbolic Instruction Code' - which means that it was originally intended for learners. It is a simple, even friendly language as languages go, and this is a good thing for you and for the people who make computers because BASIC goes well with small machines. Because of the rapid growth of 'microcomputers' like VIC, it has become a very important computer language.

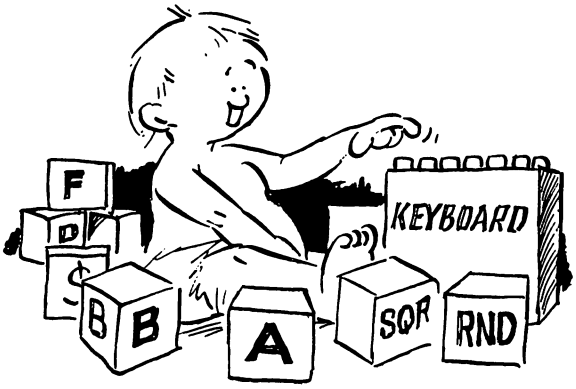
4 This book is for you

This book is intended to help you learn. It is not a library of programs, either useful or useless. I want to show you how easy it is to do your own BASIC programming, and to help you teach yourself. I won't deny that there are a few hurdles along the way, but together we can take them in our stride. All you need is your VIC computer. You don't need a cassette unit, although round about Chapter 9 you would begin to find one useful for saving programs. You already know how to connect VIC to your television and get it running. Probably you have run a few programs on it - but this is not essential. All you need now is to start on the next page and follow it through. When you reach the end you will know all about BASIC as it applies to the VIC. Good luck!

Two

FIRST THINGS

FIRST



1 A real little program

Let's start by trying to be useful. You and Sonia went to the local tape trap to start a library of TV movies. You bought 23, but 9 were too violent even for you, so you took them back. On that trip you got a bit carried away among the weepies and came home with 17 new ones. How many do you have now? Of course you could work this out yourself, but here is a little BASIC program on two lines that you could use to make VIC do the hard arithmetic for you:

```
10  PRINT 23-9+17
20  END
```

From these small beginnings great programmers grow. You and VIC have to understand each other, and that is the purpose of the BASIC language. The great thing about BASIC compared to other languages is that you need only a few facts to start using it.

In BASIC you write a series of instructions telling VIC what to do, and in what order. The one above tells VIC first of all to work out and 'print' the sum 23-9+17 and secondly to stop.

You can see that each of the lines begins with a line number:

```
10  PRINT 23-9+17      is line number 10

20  END                is line number 20
```

The line numbers tell you and VIC in what order to obey the instructions given by the program. Therefore every line of a BASIC program has to begin with a line number.

4 Start with BASIC

Each numbered line of a BASIC program is called a statement. There are two statements in this simple example, the PRINT statement and the END statement. The PRINT statement asks the computer to work out the sum 23-9+17 and print the answer. The END statement indicates that the program is finished. On the VIC, the END statement is not essential (on some computers every program must end with an END, but not on VIC).

2 Take this, VIC!

To use a computer program, you first have to get it into the computer. This is what the keyboard is all about. It is laid out very much like a typewriter, but some of the special symbols such as '+' are not where a typist would expect them and there are some extra keys. Ignore for now all the little pictures on the fronts of the keys and concentrate on the tops. There is a special key called RETURN near the right hand end of the keyboard. This is a very important key as you are about to discover.

All you have to do to get the program into the computer is type it, beginning each line with its line number and ending each line with the RETURN key. The number of blank spaces inserted between things (or left out) during typing is unimportant. For example,

```
10  PRINT 2 + 3    and    10PRINT2+3
```

both mean the same thing. However you can't break up a word like PRINT or END. This is wrong:

```
10 PR  INT  2+3
```

The order in which lines are entered is not important either, because the line numbers tell VIC what the real order should be. It is usual to separate line numbers by 10, since a programmer may later wish to insert extra lines, although any separation is allowed.

EXERCISE:

Type the little program into your VIC. Be careful to begin each line with the line number. At the end of each line type RETURN. If you make a real mess of it, switch VIC off and on again and start over.

3 Let's look - the command LIST

It is very likely that you will make mistakes when you type BASIC programs at the keyboard. It is important to be able to examine a program as the computer sees it, because it is not always all there on

the screen for you to look at. So we should get used to using the LIST command right from the beginning. With it, VIC can be asked to display the latest version of your program.

EXERCISE:
Type in LIST and push RETURN to obtain a listing of the program.

How did VIC know that LIST was a command? Simple: statements of a BASIC program always begin with a line number. Anything that does not begin with a number is a command. If you type in a statement of BASIC and forget the line number, VIC will think it is a command.

4 Now correct our errors - line replacement

Because everyone will make typing errors, it is necessary to be able to correct them. You will see a bit later how to use the 'cursor' to do this. In BASIC it is always possible to change lines, insert new lines, and delete unwanted ones. This is easy because all lines of a BASIC program begin with line numbers, and whenever a new line is typed it becomes part of the program. Here's how:

(i) To replace or correct a line: just type it again and push RETURN.

EXAMPLE:

Your program is

```
10  PRONG 23-9+17
20  END
```

and you type in

```
10  PRINT 23-9+17
```

and push RETURN

The program is now correct, and is

```
10  PRINT 23-9+17
20  END
```

which you can check by the LIST command.

(ii) To insert a new line: type the new line and give it a line number that gets it in the right place. As an example, some new lines could be added to the same little program.

EXAMPLE:

With the same program already in the computer, you type

```
16  PRINT 73-2
14  PRINT 64+5
```

and the program then reads

```
10  PRINT 23-9+17
14  PRINT 64+5
16  PRINT 73-2
20  END
```

VIC has looked at the line numbers and put the new lines in the correct place. This is why the original line numbers were spaced by 10. This is also why the lines of the program could have been typed in any order. VIC will arrange your program by its line numbers.

(iii) To get rid of a line: type in only the line number and push RETURN.

EXAMPLE:

To delete those extra lines again, type

```
14  (just the line number and RETURN)
16  (ditto)
```

The program again reads

```
10  PRINT 23-9+17
20  END
```

This method of correcting a program is called 'line editing'. You can also edit directly on the screen as will be described soon.

EXERCISE:
Experiment with the line editing facilities. Be sure that you know how to add, change, and delete lines. You can check out each change you have made with the LIST command. Finally put the program back to its given form and list it to be sure.

5 All systems go - the command RUN

I hope you are patient enough to have gone through all of this very carefully. If so, you have made a good start. But I expect you and Sonia are anxiously waiting for the answer.

So far we have created a little program and learned to list and edit it. Now we want to try it. At last! The command RUN launches your program (this is nothing to do with the RUN/STOP key on the VIC keyboard). When the RUN command is given, VIC begins to obey the instructions given by the program.

Once VIC has taken over it is not possible to edit the program or use other commands until it is finished. The little example won't hold you up for too long! It will give up at the END statement. If you have made an error in the grammar of the program, VIC will tell you by saying

```
?SYNTAX
 ERROR
```

EXERCISE:
Type in the command RUN. If it doesn't work it is not typed correctly. Fix it if that happens. Where does VIC print its answers?

6 Screen editing

While you have been using your VIC, you will have noticed a small blue rectangle blinking away on your screen. It always seems to be just to the right of whatever you have just typed or on top of the thing you are about to type. This is called the 'cursor'. It is there to tell you where you are on the screen. You can actually move the cursor around. On the keyboard there are two keys marked CRSR, just below the RETURN key. One of these will move the cursor to the left or right, and the other will move it up and down. No prizes for working out which is which! To move the cursor down the screen, you press the key with the arrow

pointing down. Be careful - if you hold it down it will keep moving! To move the cursor up you have to push SHIFT and the same key at the same time - just like getting the top symbols on any keyboard. Similarly, the cursor is moved to the right or left by the other CRSR key.

When you are moving the cursor around on the screen, you do not affect anything that is on the screen already. But if you move it to the middle of a line and then start typing, you are replacing what was there already. When you have finished making changes to a line, push RETURN and VIC will record your changes. If you forget to push RETURN, you won't change the program even though you have changed the screen. If you want to enter a command, it has to go on a clear line. To get to a clear line you can either use the cursor or clear the screen by pushing SHIFT and CLR/HOME.

EXERCISE:

With the little test program on the screen - put it there with the LIST command - change PRINT to SPORT. Don't forget to push RETURN. Check that it worked by another LIST command.

You can delete things that you do not want. If you push the key marked INST/DEL, you DELETE the character to the left of the cursor, and the line gets shorter. Again you need to push RETURN to make the modified line permanent.

On the other hand, if you push SHIFT and INST/DEL, you open up a space where the cursor is and this enables you to INSerT extra things in the middle of the line.

You can use these screen editing facilities on a program which you have LISTed, or on a program that you are typing to clear up mistakes when you first make them.

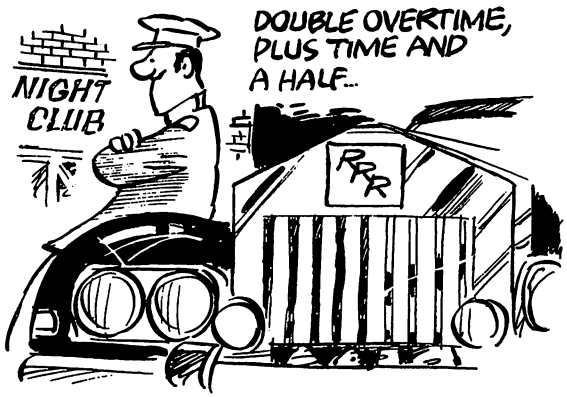
EXERCISE:

Use screen editing to practise replacing characters, deleting characters, and inserting them. This is a very powerful editing facility. Make sure you can use it before going on.

7 And for your last trick - look in the Appendix

This is a good time to introduce you to the summary of BASIC at the end of this book. If you ever need reminding of the rules for any part of BASIC, look them up in the back. Occasionally you will find extra information there. For example, I haven't told you everything about the LIST command!

Three



THE THIRD R

1 It's 'Rithmetic of course

One of the things computers can help us with is arithmetic - they can be made to do the hard work in adding up our accounts and working out (shudder) our taxes. That is one reason why many people want to learn BASIC - so they can program these things themselves. So we need to know what we can expect. It's very easy, actually, although there's a little sting in the tail.

2 Adding and subtracting

In the previous chapter, you and Sonia did a sum:

```
10  PRINT 23-9+17
20  END
```

Actually, VIC doesn't need the END statement. You could have only

```
10  PRINT 23-9+17
```

Numbers like 1, 2, 23, 9, 17, and so on, are called constants. Our program had the expression 23-9+17 which added and subtracted the constants 23, 9 and 17. Everyone knows that the order in which you add numbers is unimportant: 5+1 is the same as 1+5. This is not true for subtraction, as 5-1 is not the same as 1-5. Therefore the order in which you write numbers and operations like addition and subtraction is important. In BASIC, operations are done from left to right, just as we have all learned to do arithmetic. The meaning of

```
10  PRINT 23-9+17
```

is obvious to you, me, and the computer.

The symbols + and - also have a meaning if they come before a number, as in

```
10 PRINT +99      or      10 PRINT -73
```

and on VIC you can do something like

```
10 PRINT 6+-+-5
```

What does it mean? Try it to make sure.

3 Multiply and divide

In BASIC the symbol * is used for multiplication. This is written between the numbers to be multiplied. Like addition, multiplication is not order-dependent.

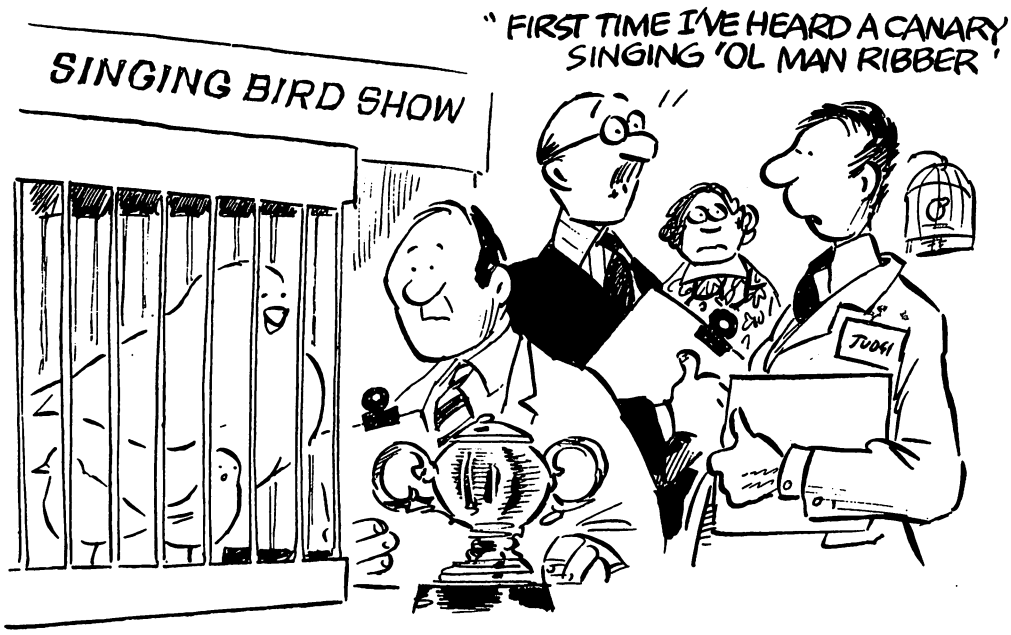
EXAMPLE:

There are 2.204 pounds to the kilogram. How many pounds does a nine kilogram canary weigh?

This will do it:

```
10 PRINT 9*2.204
```

Here there are constants with decimal points in them. You can always do this in BASIC.



EXERCISE:
In 1974, my publisher sold 318 copies of my book. For each one I received a royalty of 0.24 interplanetary credits. How much did I get? Will I soon be rich?

Numbers are divided when the symbol / is used. The order of the numbers does matter, since

15/3 is 5 and 3/15 is 0.2

EXAMPLE:
How many kilograms does a nine pound canary weigh? Do this:

```
10  PRINT 9/2.204
```

Do you believe in nine pound canaries?

EXAMPLE:
It's a long long way to San Jose; 434 miles to be exact. At 56 miles per hour the highway patrol will probably not stop me. How long will it take to get there? This is it:

```
10  PRINT 434/56
```

EXERCISE:
You can do 29.2 miles to the gallon and you have 14 gallons left. Can you make it to San Jose?

4 Raise a number to a power

There is one more operation in the arithmetic of BASIC, the raising of a number to a power. The symbol for this is the vertical arrow, ↑.

5 ↑ 3 means 5*5*5 or 125
3 ↑ 5 means 3*3*3*3*3 or 243

There are also fractional powers; for example

2 ↑ 0.5 is the square root of 2, or 1.4142...

EXAMPLE:
You bet one interplanetary credit and throw the dice 12 times. Each time you double your money. How much have you won? Ask VIC:

```
10  PRINT 2 ↑ 12
```

On the 13th throw you lose it all. Tough.

EXERCISE:

As you were leaving old St Ives, you met a man with seven wives. Every wife had seven brats and every brat had seven cats. Every cat had seven kits and every kit had seven lives. How many kit's lives were going to St Ives? The answer is 16807. Can you get it?

5 'Rithmetic stew - expressions

Now is the time to throw all the operations into the pot at once. This has been delayed because there is a complication.

After a successful seven-a-side football match, your team is awarded 3 pints of grog each plus another 11 to share between you. How much grog does each player get? VIC will tell you:

```
10  PRINT 3+11/7
```

Is it really evaluated from left to right? If so, it would mean

```
10  PRINT (3+11)/7    with the result 2. You've been cheated!
```

Or does it mean instead

```
10  PRINT 3+(11/7)    with the result about 4.57?
```

I think that you can see that the result you want is 4.57. Check that this is what VIC gives you. You can probably see that expressions are not evaluated from left to right - it is a bit more complicated.

It is necessary to have an exact set of rules about this arithmetic so that both you and the computer know what should happen. In BASIC some operations are given a higher priority than others. The order is:

( )	things in brackets	highest
↑	raising to a power	
* /	multiplication and division	
+ -	addition and subtraction	lowest

The computer will always look at an expression and do the things you have put in brackets first. Of course inside the brackets could be all kinds of things including more brackets, which have to be done first!

Among the actual arithmetic operations, powers are done first. Then the multiplications and divisions are done from left to right. Finally, the additions and subtractions are carried out, again from left to right. You

can always force the machine to do things in the order you want by using round brackets. If you do this, the brackets must always be in pairs - one right bracket for every left bracket, as

```
10  PRINT 1+2*(3+4*(5+6))
```

You can write a sequence of plus and minus signs, and VIC will work out what you want - the expression 1+-+-2 is allowed and has the value 3; similarly 1+(-2) is permitted because anything in brackets is a self-contained expression and can have + or - in front of it. You cannot write * or / operations on their own Therefore -3 is a correct expression on its own but *3 is not. Similarly 2/*3 is nonsense.

This business of priority in arithmetic is the sting in the tail I referred to earlier. People have a tendency to get caught, particularly by division. Notice that

7/3+4 is not 1 and 7/(3+4) is 1

EXAMPLE:

Stevastian Covett runs the mile in 3 minutes and 40 seconds. What is his average speed in miles per hour? This is slightly more complicated because the time is in minutes and seconds. But we can convert this to hours and get the average speed:

```
10  PRINT  1/((3+40/60)/60)
```

EXAMPLE:

An Acapulco diver jumped off a cliff and hit the water after 7.72 seconds. How high was the cliff?

The formula for falling under gravity, ignoring wind resistance, is

$$\text{distance} = \frac{1}{2} \times \text{acceleration} \times \text{time}^2$$

We have to use the right units. Working in seconds and metres, the acceleration under gravity is 9.81 metres/second/second, and so the answer is:

```
10  PRINT 0.5*9.81*7.72↑2
```

EXERCISE:

You do this one. Fernando deposited one interplanetary credit with the Interstellar Bank just over 7 years ago. Four times a year, Interstellar credited him with 4% interest. How much has he now? If

you can't do it, compound interest is used as an example in the next chapter.

6 All this and a calculator too - direct expressions

This is not part of the normal BASIC language, but many small computers will do it, including VIC. The difference between a line of a program and a command is the line number. You know that

```
10  PRINT 5↑3
```

is a line of a BASIC program. What happens if you type into VIC just

```
PRINT 5↑3
```

Oh dear! I suppose you think I should have told you about this earlier. It turns out that VIC will obey many statements of BASIC immediately if you type them in as a command - they call this 'direct mode'. It is useful, but not powerful because you would have to get your entire program on one line - which in theory you can - but it gives you no opportunity to correct it. Use it to get quick results, as from a calculator, but don't use it to do more serious programming.

In direct mode you don't have to type the word PRINT - you can use a question mark. Try this:

```
?2↑8
```

So try this:

```
10  ?2↑8
```

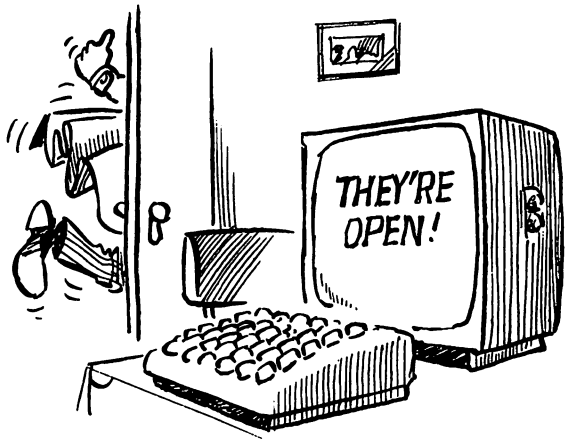
When you LIST this you will get a big surprise. This is not a normal BASIC feature, but a handy little thing that VIC does for you.

Finally, the Commodore manuals do not tell you if a particular BASIC statement can be used in direct mode. If you want to know, look it up in my Appendix.

So you can actually use VIC like a calculator. You know, the first hand-held calculators were as expensive as a VIC. And VIC can do much, much more. After all, this is only Chapter 3!

Four

TALKING TO BASIC



1 Send yourself a message

By 'talking to BASIC' I do not mean that the machine can literally speak to you, or you to it - although that isn't so many years off. What I do mean is that you can easily include messages in your BASIC program that appear on the screen so that when you run a program, it can tell you what to do. Also, when the program is running, you can give it values to work on.

The programs of the previous chapter showed their results in a somewhat unhelpful way. In a complicated program a lot of numbers appearing on the screen with no explanation could be very confusing. In a PRINT statement, a message can very easily be displayed on the screen. This greatly improves almost any program.

EXERCISE:
Try this program.

```
10  PRINT "HELLO YOU"
```

As can be seen from this example, a message can be produced by a PRINT statement simply by enclosing it in quotation marks. Whatever you put inside the quotation marks is displayed on the screen when you run the program. Earlier, you ran programs like

```
10  PRINT 3+11/7
```

to help you divide up the grog. The program

```
10  PRINT "3+11/7"
```

is quite different. Try it.

EXERCISE:
Now try this one.

```
10  PRINT "YOUR SHARE"3+11/7
```

Can you see what this does?

When you type this in, you will see that it is a bit too long for the screen. Just keep typing and see what happens. Don't forget to push RETURN when you get to the end of the statement. Is your program OK? Will it RUN? From now on we don't have to be too bothered about the lengths of our BASIC statements.

Normally several items to be printed are separated by commas, as in

```
10  PRINT "CATCH",2*11,2+2
```

but it is not necessary to include commas before or after messages in quotation marks. Here is a longer program which still doesn't do much. This one has two PRINT statements. You will see when you run this that each PRINT statement starts a new line on the screen.

```
10  PRINT "TWO AND TWO"
20  PRINT "ANSWER = "2+2
```

2 Inserting remarks - the REM statement

You can include statements in your BASIC programs which help to explain the meaning of the program to you, but which do not do anything when you run the program. The REM (for REMark) statement does this. It is always a good idea to use them, and in complicated programs they are essential.

The REM statement has the form

```
line number  REM any remark or comment
```

and added to the simple program this could be

```
10  REM  A DEMONSTRATION
20  PRINT "TWO AND TWO"
30  PRINT "2+2 ="2+2
```

Are we going too fast? Run this program. Be sure you understand the difference between what a REM statement does and the message facility

that goes with the PRINT statement.

REM statements help to explain programs. If you later add a cassette recorder or disk unit to your VIC, then you will save lots of programs that you have written. When you later return to them, the REM statements will help you to recall the details of your program in case you want to change it, or add to it.

3 Chucking old programs away - the command NEW

In the previous chapters, you learned how to create and edit programs, and to run them. Now you are going to want to make new programs that are more than one line long, and you don't want old programs hanging around to mess them up. There is a command that throws away your old program and lets you begin a new one. If you enter the command NEW (and push RETURN of course - you should be used to that by now) your old program is lost and you can start a new one. You can also clear the screen at any time - without losing your program - by holding down the SHIFT key and pushing CLR/HOME.

4 Giving values to a running program - the INPUT statement

You have seen how you can use BASIC to print numbers and messages. It is equally important to be able to give values to a running BASIC program yourself. This enables the usefulness of the computer to leap beyond what a mere calculator can do. In BASIC, a program is written using names for the values to be used, much as in algebra, and an INPUT statement in the program will ask you for the actual values when you run the program.

EXAMPLE:

You sell sea shells by the sea shore. Whenever a sea shell is sold you have to add on 15% sea shore tax (SST) which the SS tax inspectors have ways of making you collect. So every time you sell a sea shell you ask VIC how much to charge your customers. The price with the tax added on is 1.15 times the price before tax. Here is a program:

```
10 INPUT S
20 PRINT 1.15*S
```

When you run this program, VIC will ask for the price of the sea shells you are selling. The program calls this S. VIC prompts you for the value of S by putting a "?" on the screen. When this happens,

you should type in the sea shell price, push RETURN, and VIC will tell you what the sea shore sea shell selling price is.

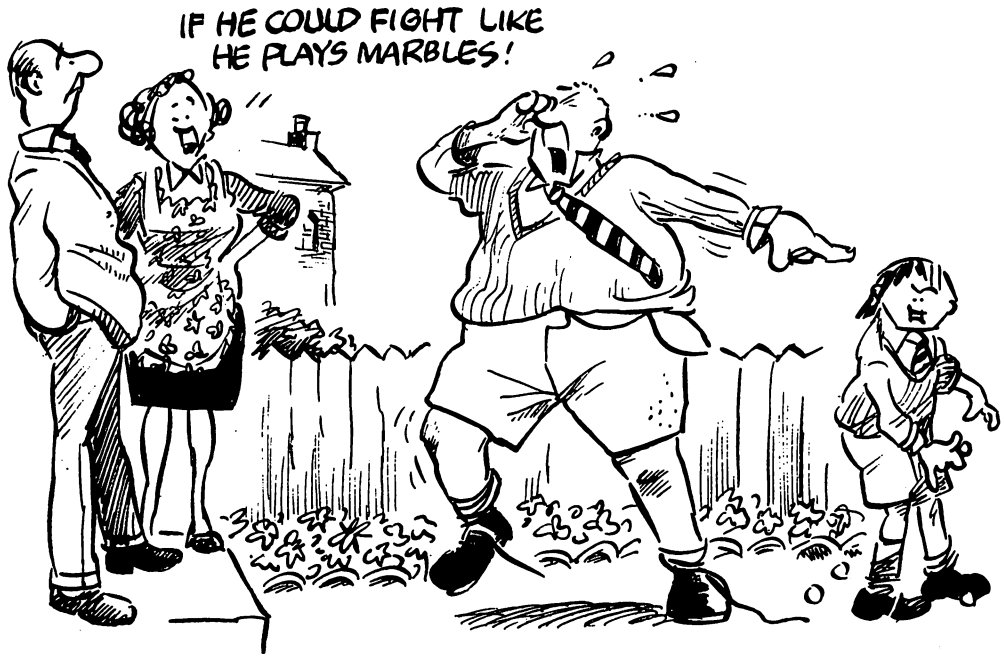
EXERCISE:

Run this program. When you are prompted, type in 100 and push RETURN. Did you get the answer 115? Good. Now every time you RUN this program it works out the sea shell sea shore selling price for you.

EXAMPLE:

Actually the sea shore tax program is lousy. Since we know about REM statements and how to print messages, we can make this a whole lot better:

```
 10  REM  TAX ADDER ONER
 20  REM  EXPLAIN
 30  PRINT "HELLO BOSS"
 40  PRINT "TYPE IN SEA"
 50  PRINT "SHELL PRICE"
 60  PRINT "FOR SS TAX"
 70  PRINT "ADDED ON"
 80  INPUT S
 90  REM  WORK IT OUT
100  PRINT "WITH TAX THAT'S"1.15*S
```



EXAMPLE:

Your two kids have different numbers of marbles (the glass things, not brains). This is because the big one keeps winning. You're a socialist. VIC will help you redistribute the wealth. You are going to tell it how many marbles each of the two has, and VIC is going to tell you how many they should have. Put another way, it will find the average of two numbers:

```
10  REM  SOAK THE RICH
20  PRINT "TYPE IN X AND Y"
30  PRINT "TO DIVIDE X+Y EQUALLY"
40  INPUT X,Y
50  PRINT "X = "X "Y = "Y
60  PRINT "GIVE EACH "(X+Y)/2
```

OK, Solomon. If the total is odd, are you really going to cut one in half?

There is another good thing added to this program at line 50. The program 'echoes' the input.

EXERCISE:

Run this program several times. The INPUT statement at line 40 will prompt you with "?". When you see this, you type in the values for X and Y with a comma between them and push RETURN. VIC should then give you the answer.

However, there are several things that you might do wrong. If you give too many values, that doesn't matter - VIC takes the first two. But if you give only one value, VIC will make you give more by prompting you again - this time with "??".

You get the message "REDO FROM START" if what you type is not a number. Make these errors deliberately this time so that you can see what happens.

A good programmer gives a lot of attention to the dialogue between the program and the person who runs it. The 'personality' of the program depends on the care that has been taken in making it friendly - this is up to you.

5 So now you know about variables in BASIC!

In the previous section, values were given names like S, X and Y. These are 'variables' to which you can give names when you write the program, and set the values later when you run the program. There are lots of names for ordinary variables allowed by BASIC. These are the single letters A to Z, any combination of one letter followed by a number, such as A0, A1, ..., A9, B0, B1, ..., B9, and any combination of two letters such as XY, BQ, and so on. There are a few names that you cannot use because they mean something special to BASIC: TI, ST, ON, OR, GO, TO, FN, and IF.

You may notice that the letter O (oh!) and the number 0 (zero) are easily confused. This is why the number 0 appears with a line through it on the screen of the VIC 20.

6 Another way of giving values to variables - the LET statement

To begin with, we could only do arithmetic with constants in a PRINT statement. Then we discovered how variables could be used instead, with their values given manually in response to an INPUT statement. Now we will see that there is a special statement for giving values to variables - the LET statement.

Here is an example:

```
100  LET TX=1.15
```

This gives the value 1.15 to the variable TX, and we could add this to our tax collecting program:

```
100  LET TX=1.15
110  PRINT "WITH TAX THAT'S "TX*S
```

A LET statement has the form:

line number LET variable name = expression

The expression can be more complicated, for example

```
80  LET C2=C1*(1+I/100)↑N
```

You will wish to know exactly what happens when a LET statement is obeyed. First of all, the expression on the right hand side is worked out. The result then replaces the value of the variable on the left hand side. LET statements are usually called assignment or replacement statements. On the VIC you can omit the word LET if you want, just writing

```
line number  variable = expression
```

But I'm not going to do that. Because of the way LET works, a statement like

```
66  LET I=I+1
```

is not a nonsense statement, but something very useful as we shall see a bit later.

EXAMPLE:

Let's convert pounds and ounces to kilograms. The number of pounds in a kilogram is 2.204. We'll make our program really friendly so that it introduces itself and explains that it wants to be given pounds and ounces so that it can convert them to kilograms:

```
 10  REM  CONVERT POUNDS
 20  REM  AND OUNCES TO
 30  REM  KILOGRAMS
 40  REM  WARM GREETINGS
 50  PRINT "HELLO I'M THE METRIC"
 60  PRINT "CONVERTER PROGRAM"
 70  PRINT "AT YOUR SERVICE"
 80  PRINT "TELL ME A WEIGHT"
 90  PRINT "IN POUNDS,OUNCES"
100  PRINT "AND I'LL TELL YOU"
110  PRINT "IT IN KILOGRAMS"
120  INPUT P,OZ
130  PRINT "THANK YOU"
```

The program first of all converts the given weight to pounds (there are 16 ounces in a pound):

```
140  LET PW=P+OZ/16
```

and then works out the result in kilograms:

```
150  LET K=PW/2.204
160  PRINT "THAT MAKES "K" KG"
170  PRINT "IT WAS TERRIFIC TO"
180  PRINT "SERVE YOU. PLEASE RUN"
190  PRINT "ME AGAIN ONE DAY"
```

You could use this program to work out how much your canaries weigh. If you try it, as you should, you will find that the 19 lines of this program more than fill the screen. So how can you LIST it? Look in the Appendix, at the LIST command. You can list whatever bits of it you want.

EXAMPLE:

Here is a money program. The computer will tell you how much money you get back from your investments.

```
10  REM  COMPOUND INTEREST PROGRAM
20  PRINT "HOW MUCH DO YOU INVEST"
30  INPUT C1
40  PRINT "WHAT IS INTEREST RATE"
50  INPUT I
60  PRINT "HOW MANY PERIODS"
70  INPUT N
80  LET C2=C1*(1+I/100)↑N
90  PRINT "RETURN IS " C2
```

This is based on the formula

$$r = c(1 + i/100)^n$$

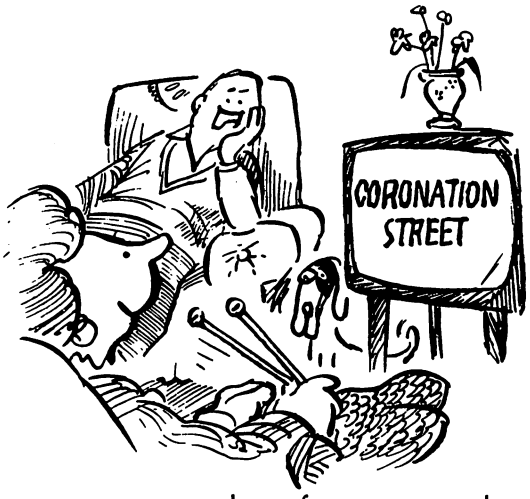
where

c = capital invested
i = interest rate in percent for each investment period
n = number of investment periods, or the number of times the interest has been compounded
r = the return

EXERCISE:

Using this program, find out the number of periods taken to at least double an investment at a rate of 8%. Is it worth it with inflation like it is?

Five



OVER AND OVER

AND OVER AGAIN

1 Forever

We will see here how to make a program go on and on forever - and we will have to learn how to stop it!

The GO TO statement is the hero (or villain!) of the piece. This enables you to change the order in which the statements of a program are obeyed. Normally they are taken one after another, in the order of their line numbers. However, if you put in a GO TO statement, it jumps to somewhere else. Try this:

```
10  REM  CHEEKY PROGRAM
20  PRINT "JUST TRY TO"
30  PRINT "STOP ME ";
40  GO TO 30
```

Notice the semicolon at the end of line 30. This makes printing carry on on the same line. You should RUN this program, and when you do you will see that it seems to go on mocking you forever. This is because the GO TO statement at line 40 sends VIC back to line 30 - again and again and again. Forever. Do not despair - nil illegitimati carborundum! stop it with the RUN/STOP key. That's better! If you want it to resume, type in the command CONT. If you make some changes to the program between stopping it and trying to CONT, you can't CONT. Try it and see.

So far so good. A repeating program can be stopped by pushing RUN/STOP. However, if there is an INPUT statement in the repeating part, you have to do something different.

EXAMPLE:

A program in the previous chapter showed you how to add tax on to the price of sea shells. I'll let you in on a secret - it doesn't have to be sea shells, it will work for anything. More complicated is the

problem of trying to deduce the price without tax if you know the price with tax. Suppose a banana warmer costs 100 interplanetary credits with banana warmer tax at 15%. What did it cost before tax? If you think the answer is 85, you're wrong. Here is a program to tell you. Notice that because it has a GO TO statement in it it will keep on running forever, taking away banana warmer tax as many times as you want:

```
10  REM  TAX TAKEAWAY
20  PRINT "THIS PROGRAM TELLS"
30  PRINT "WHAT YOUR BANANA"
40  PRINT "WARMER COST BEFORE"
50  PRINT "BANANA WARMER TAX"
60  PRINT "TYPE IN THE PRICE"
70  INPUT BT
80  PRINT "IT WAS "BT/1.15
90  PRINT "BEFORE TAX"
100 GO TO 60
```

Run this. If you type in 115, you should get 100 as the price before tax. Agreed? So what is the pre-tax price if the taxed price is 100? It's not even close to 85!

To stop this, push RUN/STOP and RESTORE at the same time. Although this clears the whole screen, your program is still there, as you can see by LISTing it.

I think you can see that a GO TO statement looks like this:

```
line number  GO TO another line number
```

and it forces VIC to jump to 'another line number' instead of carrying on in the normal order of line numbers.

2 Something tricky - self-replacement

Something cool happens if the same variable is used in both sides of a LET statement, such as

```
60  LET CT=CT+1
```

Here, CT has been used in calculating its own replacement value. You can use this for counting as in this program:

```
10  REM  LEARN TO COUNT
20  REM  SET FIRST VALUE
30  LET CT=1
40  PRINT CT
50  REM  REPLACE IT
60  LET CT=CT+1
70  REM  AND GO BACK
80  GO TO 40
```

Notice what is required for counting. First a starting value is set at line 30. Then the counter CT has one added to it over and over again.

EXERCISE:

If you run this program, it all happens a bit quickly. You can slow VIC down by holding down the CTRL key. Alternatively, you could add to the program

```
65  INPUT
```

INPUT what? Basically nothing. The computer will wait at line 65 for you to push RETURN. Try it. Either way you have to remember how to stop the program.

3 Adding up - your bank account

By using self-replacement a number of useful things can be done. One of the most useful is to be able to add things up. To do this you set aside a variable to hold the sum and give it a starting value. Inside a loop we add to the sum each time around the loop.

Here is a program for adding up. It prints the count J, and the sum of all the numbers up to J. An INPUT statement stops it each time around.

```
10  REM  COUNT AND SUM
20  REM  STARTING VALUES
30  LET J=1
40  LET S=1
50  PRINT "COUNT NOW "J
60  PRINT "SUM IS "S
70  PRINT "PRESS RETURN TO GO ON"
80  INPUT
90  REM  GET NEXT VALUES
100 LET J=J+1
110 LET S=S+J
120 GO TO 50
```

EXERCISE:

You might like to add to this program

```
85  PRINT "□"        ie SHIFT and CLEAR/HOME
```

where the message is obtained by pressing both the SHIFT and CLEAR/HOME keys. It will look like □ on the screen. Useful! Remember this for future use.

EXAMPLE:

VIC can check your latest bank statement. This program asks you for the starting balance, and then you enter each transaction and get your new balance. It uses BB for your bank balance, and TR for each transaction.

```
10  REM  WATCH THE BANK
20  PRINT "BANK BALANCE CHECKER"
30  PRINT "GIVE STARTING BALANCE"
40  INPUT BB
50  PRINT "NOW THE TRANSACTIONS"
60  PRINT "MINUS FOR WITHDRAWALS"
70  PRINT "PLUS FOR DEPOSITS"
80  INPUT TR
90  LET BB=BB+TR
100 PRINT "NEW BALANCE "BB
110 PRINT "NEXT"
120 GO TO 80
```

4 A better class of prompt

You may find it convenient to put a message in the INPUT statement, such as

```
80 INPUT "NEXT";TR
```

in the bank balance checker. This will include the message "NEXT" in VIC's prompt for the value of TR. This would then make line 110 unnecessary, and the prompt would look better because the ? will appear right after the word "NEXT". You can always have one message in an INPUT statement:

line number INPUT message ; variable list

You need the semicolon after the message.

5 Today's new word - recurrence

So what is this thing called recurrence? It is what happens whenever a variable is used for self-replacement. Therefore both counting and summing make a kind of recurrence. There are other kinds of recurrence.

For example, suppose an ageing snail is running out of energy so that it goes one metre today, half a metre tomorrow, a quarter of a metre the next day, and so on. If the snail lives forever, how far does it get? To work this out you need to do this sum:

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$$

(Do you know what the answer is?) You could write

```
10  LET S=1
20  LET N=0
30  PRINT "LATEST SUM "S
40  LET N=N+1
50  LET S=S+0.5↑N
60  GO TO 30
```

However, if you were clever you would notice that the bit added on is half the bit added on before, i.e.

```
10  LET S=1                    40  LET T=T/2
20  LET T=1                    50  LET S=S+T
30  PRINT "LATEST SUM "S       60  GO TO 30
```

What we have done is notice a different recurrence which gives each new term by dividing:

```
40  LET T=T/2
```

Do you understand? The second version is clever, but is it any better? Well, it will be a bit faster because raising to a power is slower than dividing.

EXERCISE:

Run this latest snail program. Evidently VIC thinks that the snail arrives. This is because eventually the new bit called T gets too small for VIC.

6 Rabbits certainly can breed!

Here is another fascinating recurrence. A long time ago a man nicknamed Fibonacci (1202 AD) was interested in population growth. He asked the

question 'How many rabbits would be produced from a single pair in n generations?' He assumed that every month a pair of rabbits produces another pair, and that rabbits begin to bear young when they are two months old. Honi Soit Qui Mal Y Pense! This led to the famous Fibonacci series, usually expressed as a recurrence:

$$F_n = F_{n-1} + F_{n-2}$$

F_n is the number of pairs after n months. We use $F_1 = 1$ and $F_2 = 1$ also.

Here is a program to do this:

```
10  REM RABBITS RAMPANT
20  LET F1=1
30  LET F2=1
40  REM MAKE NEXT TERM
50  LET F=F1+F2
60  PRINT "BREED "F
70  REM REVISE OLD TERMS
80  LET F1=F2
90  LET F2=F
100 GO TO 50
```

All these fascinating things arise from the ability of a BASIC program to loop back on itself using the GO TO statement, and from the fact that a variable can be used to calculate a new value for itself. This is the meaning of recurrence.

If you run this program, you are going to see some large numbers begin to appear after several generations. Soon, the numbers get too big for VIC to put on the screen. When this happens, VIC will switch to 'scientific notation' in which you see something like

```
1.13490317E+09
```

This means that the population is now 1.13490317 multiplied by 10 . The "E+09" means to multiply by 10 raised to the power 9, which is the same as moving the decimal place 9 places. The answer then is approximately 1134903170 - approximately because we do not actually know what the last digit is. VIC doesn't know either, because it can only store nine digits.

If you let the program run on even further, you will get to a point where the answer becomes too large for poor old VIC. How large is that? You can see that VIC can handle quite an impressive range of numbers.

Six

DECISIONS, DECISIONS

1 Comparing things - relational expressions

I hope that you have found everything that we have done so far to be fairly easy. Because now is the time to take a bit of a jump. So far it has been easy to follow programs from line to line because they have either been taken in order or formed a loop. Now is the time to let a program make a decision.

To do this, a new kind of expression gives us the answer TRUE or FALSE when we make a comparison. For example, you may have a variable BA which is the balance of your bank account. You want to know if you can afford that new interstellar scooter which costs 10000 credits. The expression

```
BA>10000
```

is either true or false because the symbol > means 'greater than'. You would then say in a BASIC program something like

```
70  IF BA>10000 THEN 30
```

which would jump to line 30 if you have more than enough.

In more general terms, to make a comparison, you use what is called a relational expression, which is

```
something  compared  something
              to
```

or, to be more exact,

arithmetic expression | relational operator | arithmetic expression

The relational operators that can be used to compare things are:

= equal to, i.e. A=B is TRUE if A=B, otherwise FALSE
> greater than, i.e. 10>5 is TRUE
< less than, i.e. 10<5 is FALSE

and the combinations

>= or => greater than or equal to, i.e. 5>=6 is FALSE
<= or =< less than or equal to, i.e. 5<=6 is TRUE
<> or >< not equal to, i.e. 5<>5 is FALSE

2 Be decisive - the IF...THEN statement

The IF...THEN statement uses a relational expression to make a decision about whether to jump to a chosen line number. This allows programs to decide what to do next. It looks like this:

line number | IF | relational expression | THEN | destination

If the relational expression is TRUE, the program jumps to the destination, which must be a line number that really exists.

EXAMPLE:
Now we can stop counting whenever we want, as in

```
10  REM  COUNT TO FIVE
20  LET J=1
30  PRINT J
40  LET J=J+1
50  IF J<=5 THEN 30
```

Try it. Later on we will find an even easier way to do this.

EXAMPLE:
It is unhealthy to let the recurrence programs of the previous chapter go on forever. Here is how to stop the program for doing

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$$

when the next term is less than 0.00001. This is when the snail is within a whisker of its destination. Change line 60 to

```
60  IF T>0.00001 THEN 30
```

EXERCISE:
Make the rabbits breed for exactly 25 generations.

3 More complicated decisions - NOT, AND, OR

This is a feature of VIC BASIC which not all computers have. You can use special relational expressions with NOT, AND, OR in them to make more complex decisions. They are called logical operators. You can have

relational expression — logical operator — relational expression

and again get a result TRUE or FALSE; for example in

```
30  IF I>10 OR J<20 THEN 66
```

Here is what they mean:

something OR something
is TRUE if either or both of the somethings are TRUE

thingie AND thingie
is TRUE only if both thingies are TRUE

NOT something has the result TRUE
if the something is FALSE and vice versa

If you want to make a really complicated expression, you have to know about the priority of these operations. Here it is:

AND	highest
OR	
NOT	lowest

and you can use brackets to get what you want.

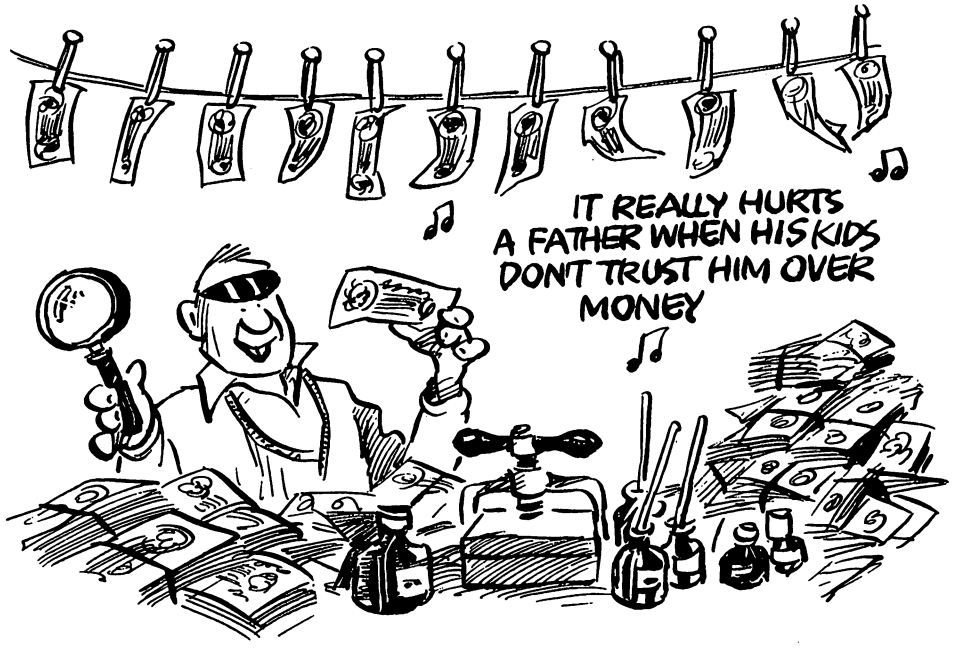
EXAMPLE:

Your two kids each get an allowance. The eldest has an allowance of A1 per week. The youngest gets A2 per month - this is a result of complex negotiations. Because there are not four weeks in a month, you think you might be able to fool them about who gets the most but VIC is on their side. Here is a program to check that A1*52 is more than A2*12, so that in a year the eldest gets more than the youngest, unless both are zero. As the kids have learned not to trust you, it also checks that neither allowance is negative. So what are the rules?

Both A1 and A2 must be greater than or equal to zero and A1*52 must be more than A2*12 unless both A1 and A2 are zero.

Here is the program they will use to check up on you:

```
 10  REM  DOWN WITH DAD
 20  PRINT "TYPE A1 AND A2"
 30  PRINT "BOTH >= 0 AND A1*52"
 40  PRINT "MUST BE > A2*12"
 50  PRINT "UNLESS BOTH ARE ZERO"
 60  INPUT A1,A2
 70  IF NOT((A1*52>A2*12 AND A2>=0) OR (A1=0 AND A2=0)) THEN 20
120  PRINT "DAD GOT IT RIGHT"
```



4 Try not to make ugly programs

Look at the statement from the above example. It is complicated:

```
70  IF NOT ((A1*52>A2*12 AND A2>=0) OR (A1=0 AND A2=0)) THEN 20
```

There are a number of ways of getting the same result. You could have written

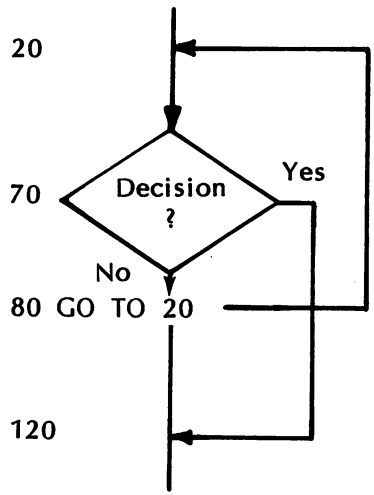
```
70  IF (A1*52>A2*12 AND A2>=0) OR (A1=0 AND A2=0) THEN 120
80  GO TO 20
```

Why not? Well, quite honestly to an experienced programmer it is repulsive. If you write an IF statement followed by a GO TO statement, you are creating an ugly structure in which the flow lines of your program cross. This is never necessary. If you are tempted to write

```
70  IF something THEN 80
80  GO TO somewhere
```

then write instead

```
70  IF opposite THEN somewhere
```



Sometimes the opposite would be achieved by the NOT as in the example above, or sometimes you would use a different comparison. These two are the same:

```
50  IF J>5 THEN 70
60  GO TO 30
70  END
```

```
50 IF J<=5 GO TO 30
```

Do you recognise this? There are no prizes for guessing which is best.

If you want to, you can make some really horrible things. Look at this:

```
 70  IF A1*52>A2*12 THEN 110
 80  IF A2><0 THEN 20
 90  IF A1=0 THEN 120
100  GO TO 20
110  IF A2<0 THEN 20
120  PRINT "DAD GOT IT RIGHT"
```

It is the same as the allowance program, but it is truly awful. Don't do things like this. Not only is a GO TO never necessary after an IF statement, a cluster of IF statements is never necessary. The NOT, AND, and OR operations are there to help you. Use them!

5 Another way of making decisions - the ON...GOTO statement

This one is not used as often as the IF statement. It is sometimes useful to be able to jump to one of several places in a program. The ON... GOTO statement does this. It looks like

```
line  ON expression GOTO   line    ,   line    , ...
                         number      number
                           a           b
```

In this statement you have to write GOTO as one word. When VIC sees an ON...GOTO statement, it works out the expression and jumps to the line number a if the integer part of the result is 1. We will find out about integer parts in the next chapter. It means that if the expression gives anything between 1 and slightly less than 2, the program jumps to line number a. Similarly, if the integer part of the result is 2, it jumps to line number b, and so on for the number of destinations you have given. If the expression gives a result less than 1, or one whose integer part is greater than the number of destinations, then the program just continues with the next line in order.

EXAMPLE:

```
60  ON I+J GOTO 10,20,30
```

If I+J is a number between 1 and just under 2, the program jumps to line 10.
If I+J is between 2 and just under 3, the jump is to line 20.
If I+J is between 3 and just under 4, the jump is to line 30.
If I+J is 4 or greater, the program carries on to the next line after line 60.

Seven

SOME FUNCTIONS

1 Little helpers

There are a number of operations with numbers that you can't do very easily by writing out a formula in BASIC, but which are needed quite often. BASIC provides a number of little functions to help you out. Here is what there is:

Name	Meaning
SQR(expression)	Square root of expression
ABS(expression)	Absolute value of expression
SGN(expression)	Sign of expression: 1 if >0, 0 if 0, -1 if <0
INT(expression)	Integer part - the largest integer not greater than expression
EXP(expression)	These are specialised mathematical functions, covered in Chapter 16
LOG(expression)	
SIN(expression)	
COS(expression)	
TAN(expression)	
ATN(expression)	
RND(expression)	Randy, the random number generator - has its own chapter, Chapter 15

To use a function, simply write it where it is desired, with its expression in brackets immediately after it. For example,

```
30  PRINT SQR(N*N+E*E)
```

prints the square root of N^2+E^2.

2 Trying them out

Here we consider SQR, ABS, and SGN in turn. INT is super-useful and will be looked at in detail in the next section.

(a) SQR

You could find a square root by an involved mathematical procedure every time you need it - some people need them quite often. This would mean spending a lot of time programming square roots - very boring. The SQR function saves us the trouble. The only thing to remember is that a negative number does not have a square root.

EXAMPLE:

If you walk 100 metres north and 300 metres east, how far are you from where you started?

Do you remember your hypotenuse formula? If you have a right-angled triangle, then the hypotenuse is

$$h = \sqrt{a^2 + b^2}$$

where h is the length of the hypotenuse and a and b are the lengths of the other two sides. Your little walk is an hypotenuse:

```
10  PRINT "GOOD MORNING, CAN I"
20  PRINT "FIND YOU A HYPOTENUSE?"
30  PRINT "TYPE IN TWO SIDES"
40  INPUT N,E
50  PRINT "HYPOTENUSE = "SQR(N*N+E*E)
60  GO TO 30
```

EXERCISE:

Try it out. It won't give any trouble about negative square roots. Do you know why I wrote N*N+E*E instead of N↑2+E↑2? Find out what happens if you try to SQR with a negative number.

(b) ABS

This function forces the sign of an expression to be positive. It can be used to keep out of trouble with SQR, as for example in

```
90  LET QR=SQR(ABS(T))
```

Very often a program is interested in the size of something

regardless of its sign, and this is when ABS is used.

(c) SGN

Sometimes it is important to know when a number is positive, zero, or negative, without caring about its actual value. In a program to manage your bank account, you might want to act differently if your balance were negative, zero, or positive:

```
50  ON SGN(BB)+2 90,30,10
```

This jumps to line 90 if you are in the red, line 30 if you're exactly broke, and 10 if you have some money left. Why was 2 added to SGN(BB)?

Here is a nifty little statement to transfer the sign of one variable to another; Z is supposed to have the same size as X but with the same sign as Y.

```
90  LET Z=SGN(Y)*ABS(X)
```

It will not work if Y is zero.

3 Cutting away decimal places - the INT function

The INT function is probably the most useful of the lot, which is why it is discussed here separately. By its definition, INT(X) gives an answer which is the largest integer which is not greater than X. This means, in simpler terms, that a positive number has its decimal places stripped away:

```
INT(73.7) is 73
```

but negative numbers move down:

```
INT(-73.7) is -74
```

Look carefully at this last example. INT does not just take away the decimal places - it moves down.

(a) Rounding

You may want to round a result to the nearest whole number. Although INT does not do this on its own, it can be very easily forced to. To round to the nearest whole number, simply add 0.5 before using INT:

```
400  LET RN=INT(XN+0.5)
```

Nearly everyone knows how to do that, but here is a nifty trick. You can round a number to any coarseness you want, which can be pretty useful with money for example. All you do is scale the values so that the required coarseness is represented by integers, round it, and then undo the scaling. Sounds complicated? Here are some examples.

(i) After calculations of interest on an investment, your friendly Interstellar Savings Corp rounds your interplanetary credits to the nearest centeroonie (there are 100 centeroonies to the credit). Without rounding, one credit invested for 7 epochs with interest at 8% per epoch would give

```
30  PRINT (1.08)↑7
```

But rounding to the next centeroonie could be done like this:

```
30  LET PY=(1.08)↑7
40  LET PZ=INT(PY*100+0.5)/100
50  PRINT PZ
```

See how the payoff PY was scaled up by 100 before the integer part was taken, and then down again.

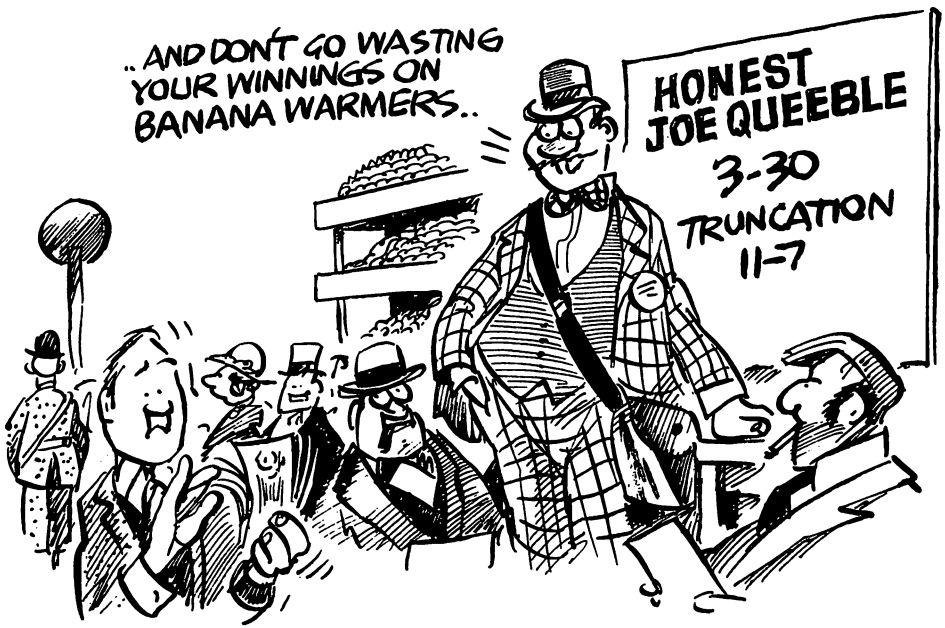
(ii) Banana warmers come in cases of 50. A customer orders WB banana warmers. You have to push this up (good for business, that) to the next 50 to get the number of cases to send, called CS.

```
80  LET CS=INT((WB+49)/50)
```

This is not quite the opposite of the centeroonie procedure. I have added 49 to be sure that the customer who orders 51 banana warmers is sent 2 cases.

(iii) Actually inflation has spoiled the value of the interplanetary credit to the extent that the smallest coin issued by the Pan Galactic Council is 5 centeroonies. If you bet a quarp (25 centeroonies) on a geegee at 11-7, the bookies will cut your prize to the next lowest 5 centeroonies. This is truncation again, with our clever scaling applied. So you get

```
140  PRINT INT((25*11/7)/5)*5
```



centeroonies. The scaling is the other way - reduced by five times before the integer part is taken. A more gererous bookie would round:

```
140 PRINT INT((25*11/7)/5+0.5)*5
```

Already we see great possibilities for the INT function!

(b) Truncation - your actual chop!

When the decimal places are stripped away from a number, it is called 'truncation'. This is what the bookies did to you above. The INT function is not quite a truncation because of its operation with negative numbers. However, the expression

```
SGN(X)*INT(ABS(X))
```

is a truncation for any number X.

(c) Converting units

EXAMPLE:

We can use INT very nicely to break numbers into parts. If we know the distance to San Jose in kilometres (unlikely, I know) we may wish to convert this to whole miles plus yards, feet, and inches, rounding to the nearest 0.1 inch. We need to know that

1 mile = 1.60934 kilometres

and also that

1 mile = 1760 yards
1 yard = 3 feet
1 foot = 12 inches.

The easy part is converting the kilometres to miles:

```
 10  REM  DO YOU KNOW THE
 20  REM  WAY TO SAN JOSE?
 30  PRINT "HOW MANY KILOMETRES"
 40  PRINT "TO SAN JOSE"
 50  INPUT KM
 60  LET ML=KM/1.60934
 70  PRINT "THAT'S "ML" MILES"
```

Now we start using INT to rip off decimal places. First of all, whole miles:

```
 80  REM  NOW BREAK IT DOWN
 90  REM  FIRST WHOLE MILES
100  LET WM=INT(ML)
```

and the bit that is left over can be converted to yards:

```
110  REM  YARDS LEFT OVER
120  LET YD=1760*(ML-WM)
```

If you see how that works, the rest is easy:

```
130  REM  WHOLE YARDS
140  LET WY=INT(YD)
150  REM  FEET LEFT OVER
160  LET FT=3*(YD-WY)
170  REM  WHOLE FEET
180  LET WF=INT(FT)
190  REM  INCHES LEFT OVER
200  LET IN=12*(FT-WF)
210  LET IN=INT(IN*10+0.5)/10
220  REM  PRINT THE ANSWERS
230  PRINT "BROKEN DOWN, IT'S"
240  PRINT WM" MILES "WY" YARDS"
250  PRINT WF" FEET "IN" INCHES"
```

(d) Remainders are useful too

When you divide 22 by 7 the integer part of the answer is 3 and the remainder is 1. Suppose we were dividing N by D and both N and D are positive.The integer part of the answer would be

```
INT(N/D)
```

and the remainder would be

```
N-INT(N/D)*D
```

if N and D were positive. Do you see how this works?

EXAMPLE:

If you don't want to upset your customers in the hot banana trade, perhaps it would be better not to send them all those extra banana warmers. There are 50 to a case, remember. This program tells you how many full cases to send and how many odd ones are needed in addition. See the remainder?

```
10  REM ARE HOT BANANAS MUSHY
20  PRINT "HOW MANY BANANA"
30  PRINT "WARMERS ARE ORDERED"
40  INPUT WB
50  REM FIRST THE QUOTIENT
60  LET CS=INT(WB/50)
70  PRINT "SEND "CS" CASES"
80  REM THEN THE REMAINDER
90  LET RM=WB-CS*50
100 PRINT "AND "RM" ODD ONES"
```

Eight

ROUND AND ROUND WE GO

1 Loop the loop

From modest beginnings, we have come a long way in learning to drive VIC in BASIC. One of the really useful things learned is making loops. First we found out how to make a loop repeat forever, and how to use a variable to count. Then we learned how to stop the counting at a particular value with an IF statement.

2 The hard way

Actually we have learned to count the hard way. Before discovering the easy way, let's be sure we know what we did. First of all, a starting value was set up. Each time around the inside of the loop the counter was incremented and tested to see if the loop should be repeated again. It is easy to see the importance of these steps: initialise, increment, and test. In the BASIC that we know so far, to repeat something 10 times, we would need to write something like:

```
50  LET C=1                 (initialise)
     :
     :
80  LET C=C+1              (increment)
90  IF C<=10 THEN 50       (test)
```

3 The easy way

In BASIC there is a special pair of statements for loop control. These are the FOR and NEXT statements. All you have to do is write

```
50  FOR C=1 TO 10
        :
        :
90  NEXT C
```

The FOR statement is responsible for initialising some variable - C in this case. The FOR is matched by a NEXT statement naming the same variable.

EXAMPLE:

Type this program as one line, i.e. without pushing RETURN until the very end. The colons squeeze several statements on one line:

```
10  FOR J=1 TO 5: PRINT J: NEXT J
```

Now try it without the line number. All these statements work in 'direct mode'.Using the colon you can make a complete little program.

EXAMPLE:

If you want to, you can use a FOR...NEXT loop to deliberately waste time. This program counts forever. Instead of pushing CTRL to slow it down, a FOR...NEXT loop at line 50 wastes a bit of time.

```
10  REM  COUNT FOREVER
20  LET I=1
30  PRINT I
40  REM  WASTE TIME
50  FOR J=1 TO 500:NEXT J
60  LET I=I+1
70  GO TO 30
```

What would happen if I had been used as the FOR...NEXT variable?

4 Adding up again

You know the song about the Twelve Days of Christmas. On the first day your truelove gives you a partridge in a pear tree. On the second day you get another partridge plus two new gifts (turtle doves), and so on. How many gifts do you receive on day 12? Here is a program to add the numbers from 1 to 12. Notice that the variable used for summing, GF, is given a starting value before the FOR...NEXT loop.

```
10  REM  ADD NUMBERS TO 12
20  LET GF=0
30  FOR I=1 TO 12
40  LET GF=GF+I
50  NEXT I
60  PRINT "SUM IS"GF
```

5 Mind your STEP

There is another useful facility in the FOR...NEXT loop. Everything up until now has counted forward by ones. You may want to do something different. Using the word STEP, you can have such things as

```
10  FOR J=1 TO 21 STEP 3
```

or

```
20  FOR X=0.25 TO 1.50 STEP 0.25
```

or

```
30  FOR P=10 TO 1 STEP -1
```

Don't forget the NEXT in each case. There are a number of rules about FOR...NEXT loops - mostly common sense - which will be given a bit later, after some examples.

6 Multiplying up - factorials

Here is something else interesting. The product of all the whole numbers from 1 to n is called the factorial of n. Mathematicians write this as n!.

```
n! = 1*2*3...*n
```

Now n! is pretty useful. Think of how many ways you have of arranging five different coloured balls in a row. You can choose any one as the first one. You then have a choice of four as the second. Furthermore, for each of these you can choose any of the remaining colours for the third one. You have 5*4*3*2*1 possible ways of arranging the colours, which is 5!. These statements would find N! for you by multiplying:

```
 70  LET F=1
 80  FOR I=1 TO N
 90  LET F=F*I
100  NEXT I
```

This is a bit like adding up, only we're multiplying up! Here is a super-duper N! program:

```
 10  REM  GET FACTORIALS
 20  PRINT "HI THERE"
 30  PRINT "SLIP ME A NUMBER"
 40  PRINT "AND I'LL FIND ITS"
 50  PRINT "FACTORIAL IF I CAN"
 60  INPUT N
 70  LET F=1
 80  FOR I=1 TO N
 90  LET F=F*I
100  NEXT I
110  PRINT "FACTORIAL " F
120  GO TO 30
```

EXERCISE:

Factorials get pretty big. Run this program and find out how big N can be before VIC gets N! wrong. Do you remember how VIC represents large numbers?

7 Nesting

You've seen those little Russian dolls - one goes inside another inside another and so on. Nested loops are the same. You can put one loop inside another as long as it uses a different variable to count and as long as the loops do not cross. Here are correct and incorrect examples:

Correct

The loops are 'nested'

```
 20  FOR X=5 TO 15
       :
 50  FOR Y=7 TO 1 STEP -2
       :
 80  NEXT Y
       :
120  NEXT X
```

Incorrect

The loops cross

```
  5  FOR P=1 TO 5
       :
 45  FOR Q=100 TO 101 STEP 0.1
       :
 95  NEXT P
       :                ouch!
200  NEXT Q
```

EXAMPLE:

After the twelve days of Christmas, how many gifts have you received in total? This would work it out and tells you the sum at the end of each day.

```
10  REM  PURE GREED
20  LET GF=0
30  FOR I=1 TO 12
40  FOR J=1 TO I
50  LET GF=GF+J
60  NEXT J
70  PRINT "DAY "I" TOTAL"GF
80  NEXT I
```

Notice that the upper limit of the inner loop is I. Very sly, that!

8 Is yours prime?

A number which cannot be broken down into the product of smaller ones except 1 is called a prime number. For example, 21 is 3x7 so it is not prime, but both 3 and 7 are primes; the first few are 1, 2, 3, 5, 7, 11... How can we tell if a number N is prime? We can try all integers I from 2 to SQR(N) to see if they are factors. How will we know? Because if I is a factor, the remainder is zero when we divide N by I. We did remainders in the last chapter. If we find any zero remainder, then N is not a prime. Here is the bit that does this:

```
100  REM  IS N A PRIME?
110  FOR I=2 TO SQR(N)
120  LET R=N-INT(N/I)*I
130  IF R=0 THEN 170
140  NEXT I
150  REM  IF GET HERE N IS A PRIME
160  PRINT N" IS PRIME"
```

Now let's use this to find all the primes less than M, a number that you type in:

```
10  REM  PRIME SEARCHER
20  PRINT "THIS IS YOUR"
30  PRINT "PRIME FINDER"
40  PRINT "SPEAKING GIVE"
50  PRINT "ME MX>2 AND"
60  PRINT "I'LL FIND EVERY"
70  PRINT "PRIME UP TO MX"
80  INPUT MX
90  FOR N=3 TO MX
        :
        :
the rest of the program as above
        :
        :
170 NEXT N
```

This program has a nested loop.

EXERCISE:

Run this program. Make it count the primes, and find out how many there are less than 500 - note that 1 and 2 are not found.

9 More uses for factorials

The factorial n! was the number of ways of arranging n different things in a row. There are some more complicated things that factorials can do. The number of ways of arranging n different things where you can only take r of them at a time is called the number of 'permutations' of n things taken r at a time, or

$$
\begin{aligned}
{}_nP_r &= n(n-1)\ldots(n-r+1) \\
&= \frac{n!}{(n-r)!}
\end{aligned}
$$

To find this you could take n! and divide it by r! Better still, you could take the product of all numbers from n-r+1 to n:

```
 10  REM  PERMUTATIONS
 20  PRINT "CALCULATE NO"
 30  PRINT "OF PERMUTATIONS"
 40  PRINT "HOW MANY OBJECTS?"
 50  INPUT N
 60  PRINT "HOW MANY TAKEN"
 70  INPUT R
 80  LET PR=1
 90  FOR I=N-R+1 TO N
100  LET PR=PR*I
110  NEXT I
120  PRINT "PERM IS "PR
130  END
```

EXERCISE:
Try it. What is nPn? What is nP ? What does this program do if r>n? Is that right? What does this program do if r<1? Is that right?

Finally, you may wish to know how many combinations there are if you can select r things from a choice of n, but you are not interested in the order. This is the same as taking the r! reorderings out of nPr.This is called nCr:

$$
{}_nC_r = \frac{n!}{r!(n-r)!}
$$

EXERCISE:
Make a program to do nCr.

10 The rules

line number FOR variable = expression TO expression STEP expression

(i) The FOR statement gives the initial, final, and step values. These can be any expressions.

(ii) A variable name must be used as the counter. There is no need to actually use it, as in:

```
10  FOR K=1 TO 5
20  PRINT "OVER AND"
30  NEXT K
```

(iii) The STEP part is optional. If you leave it out, the step size is 1. You can make the step size fractional or negative.

(iv) Regardless of the initial, final, and step values, the loop will be done at least once, as in this silly loop:

```
40 FOR BB=5 TO 1 STEP 3
60 PRINT"DID IT?"
80  NEXT BB
```

which is done once.

(v) The initial, final, and step values are considered only when the loop is first entered. They cannot be altered within the loop. For example, in

```
50  FOR I=J TO K STEP L
60  LET L=L*2
70  LET K=K-1
80  NEXT I
```

all that messing around with L or K has no effect. If J, K, and L were 1, 10, and 1 the loop would be repeated 10 times.

(vi) The counter itself can be changed and this will affect the loop. For example

```
50  FOR I=1 TO 10
60  LET I=I+1
70  NEXT I
```

is repeated only 5 times. Be sure you understand the difference between this and rule (v).

(vii) You must match every FOR statement with a NEXT statement which names the same counter.

line number NEXT variable

Actually on the VIC you can just have

line number NEXT

and it is obvious to the computer which loop should be closed. However, it is not always so obvious to you. You could save yourself a lot of trouble a lot of trouble if you always use the full version.

(viii) Loops must be nested properly. Nested loops cannot use the same counter.

Nine

POKING AROUND

1 Remember, remember

Your computer has a memory. You probably know already that your BASIC programs and the values they work with are kept in the memory, and that VIC promptly forgets them when you turn it off. Some parts of the memory hold special values that you may want to change. One part of memory is a map of the television screen. If you want to make pictures, you will have to get at this part directly, and Chapters 11 and 12 are devoted to this. VIC is also equipped to make sounds through your television, and this is also done through special parts of memory. That is what Chapter 10 is about. Here we get a bit of a preview of this, because we have to learn how to get at memory before we can do much with sound or vision.

2 Something to POKE

Try this:

```
10  REM  POKE SOMETHING
20  POKE 36878,15
30  FOR I=128 TO 254
40  POKE 36877,I
50  FOR J=1 TO 50
60  NEXT J
70  NEXT I
80  POKE 36878,0
```

If it does nothing for you, turn up the volume on the television and try again.

What you are doing is changing the note produced by the growler. How does this work? Each chunk of VIC's memory has an address, just like you do. VIC's addresses are numbers, and the address of the growler is 36877. The statement

```
40  POKE 36877,I
```

puts the value of I into memory address 36877. You can see that the FOR...NEXT loop using I runs the value of I from 128 to 254, and this changing value of I is responsible for the changing pitch that you hear. Memory address 36878 is the volume control; at the beginning of the program we turn it up full and at the end we switch it off. The other FOR...NEXT loop is just to slow things down. You could change the speed by changing the limit on this loop.

So POKE is a statement that does this:

```
line number  POKE  memory address, value
```

The value 'value' is put into the memory address called, ironically, 'memory address'.

You can use POKE as a direct instruction. Type in the command

```
POKE 36878,15
```

Ah! The growler is still on if you haven't turned VIC off in the meantime. Do this:

```
POKE 36877,128
```

and finally

```
POKE 36877,0
POKE 36878,0
```

Using POKE, you can 'crash' your computer, but you can't damage it. All the vital functions of VIC are in a 'read-only memory' which you can't spoil using POKE. If VIC should crash as a result of poking around, switch it off and on again.

Now try this:

```
10  FOR I=0 TO 15          50  FOR K=1 TO 100
20  FOR J=0 TO 15          60  NEXT K
30  LET IC=J+16*I          70  NEXT J
40  POKE 36879,IC          80  NEXT I
```

Clearly, you can have a lot of fun with the screen of VIC.

3 Take a PEEK

PEEK is the opposite of POKE. Using it, you can look in any memory address to see what is there. It gives you back the value, and for that reason it is a function.

Run this:

```
10  IK=PEEK(162)
20  PRINT IK
30  GO TO 10
```

Guess what? VIC has a clock inside! Much, much later (Chapter 20) we will learn how to use it. I suppose that I am cheating here, because as well as being at memory addresses 160-162, the clock has a name, TI. I could have put

```
10  PRINT TI
20  GO TO 10
```

but then we wouldn't be PEEKing, would we?

4 Setting up DATA inside VIC

It can be crashingly boring to have to type a lot of values in through the keyboard or to have a long list of assignment statements. So BASIC has a method of defining a list of values in advance. You do this with a DATA statement, and you get at the values using a READ statement.

The DATA statement is

```
line number  DATA  constant, constant, ...
```

as for example

```
40  DATA 223,225,219
```

The READ statement is used to transfer values from the DATA list to variables:

```
line number  READ  variable, variable, ...
```

The values are taken from the DATA list and assigned to the variables one by one. For example, in

```
10  DATA 38.2,10.5,-9.6
20  READ A,B,C
30  PRINT A,B,C
```

A will be assigned the value 38.2, B the value 10.5, and C will be -9.6. In this example the number of items was exactly right.

If there is data left over after a READ statement, another READ will continue through the data list. It is as if a pointer moves through the DATA list. Look at this program:

```
10  DATA 38.2,10.5,-9.6
20  FOR I=1 TO 3
30  READ Z
40  PRINT Z
50  NEXT I
```

At the beginning, the pointer is at the beginning:

```
next value
   ↓
38.2          10.5          -9.6
```

Then with I=1, the statement READ Z assigns 38.2 to Z and moves the pointer along:

```
              next value
                 ↓
38.2          10.5          -9.6
```

so that when I=2, 10.5 is assigned to Z and the pointer moves again:

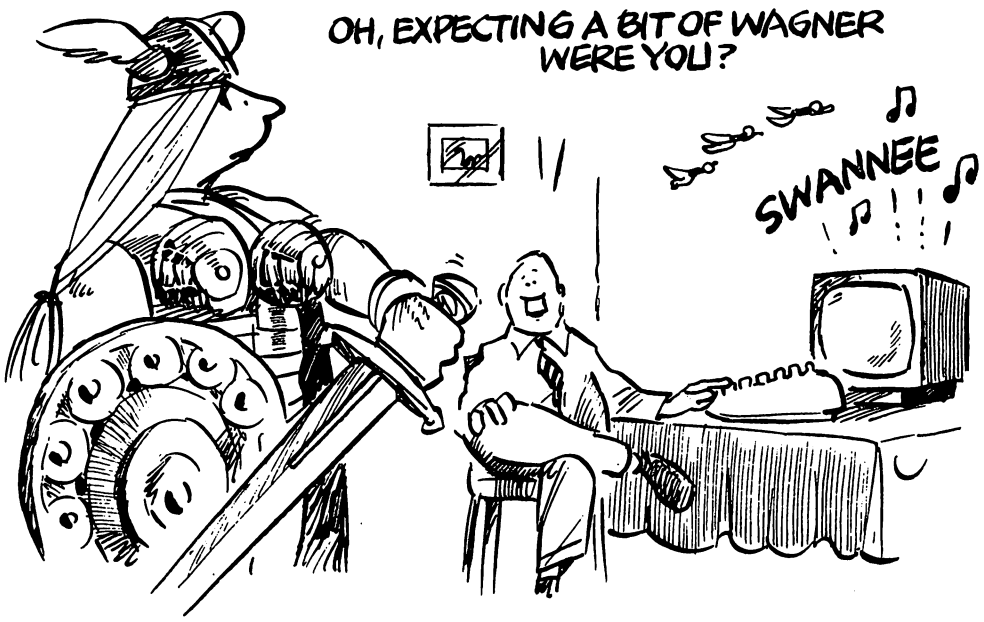
```
                            next value
                               ↓
38.2          10.5          -9.6
```

so that the final READ gives Z the value -9.6.

You can also have several DATA statements, and this extends the list, still in the order that the data originally appears, as in

```
10  DATA 209,209,209
20  DATA 209,215,215
30  DATA 215,215,219
40  DATA 223,225,219
50  DATA 215,215,215
```

```
 60  POKE 36878,15
 70  FOR I=1 TO 15
 80  READ N1
 90  POKE 36874,N1
100  FOR J=1 TO 100: NEXT J
110  NEXT I
120  FOR J=1 TO 200:NEXT J
130  POKE 36878,0
140  POKE 36874,0
```



If you run this, it will play a tune for you.

It is wrong in READ statements to ask for more values than are available in all the DATA statements.

The RESTORE statement returns the pointer to the beginning of all the DATA statements, so that all the values can be used again. There is no way of getting back to the middle of the list, except for going to the beginning using RESTORE and reading through the list, for example in a FOR...NEXT loop.

EXAMPLE:

Change the following lines in the little tune, and try it:

```
130  RESTORE
140  GO TO 70
```

Ten

SOUNDS INTERESTING

1 Resources

As you will have discovered in the last chapter, VIC can be very noisy. Inside it are four tone generators and a volume control which can be used with the POKE statement to make all manner of wonderful sounds. They are:

	POKE Address	Possible values
Tone 1 (High Whistle)	36874	128-254
Tone 2 (Medium Whistle)	36875	128-254
Tone 3 (Low Whistle)	36876	128-254
Tone 4 (Growler)	36877	128-254
Volume Control	36878	0-15

VIC is quiet if the volume is at 0, and each tone is also quiet if its value is less than 128. For each tone, 128 is a low pitch and 254 a high pitch. Tones 1 to 3 are much more musical than growly tone 4. A bit later we will make music together. As we are really here to learn BASIC, we'll cover a few principles but leave the really fancy sound effects to you.

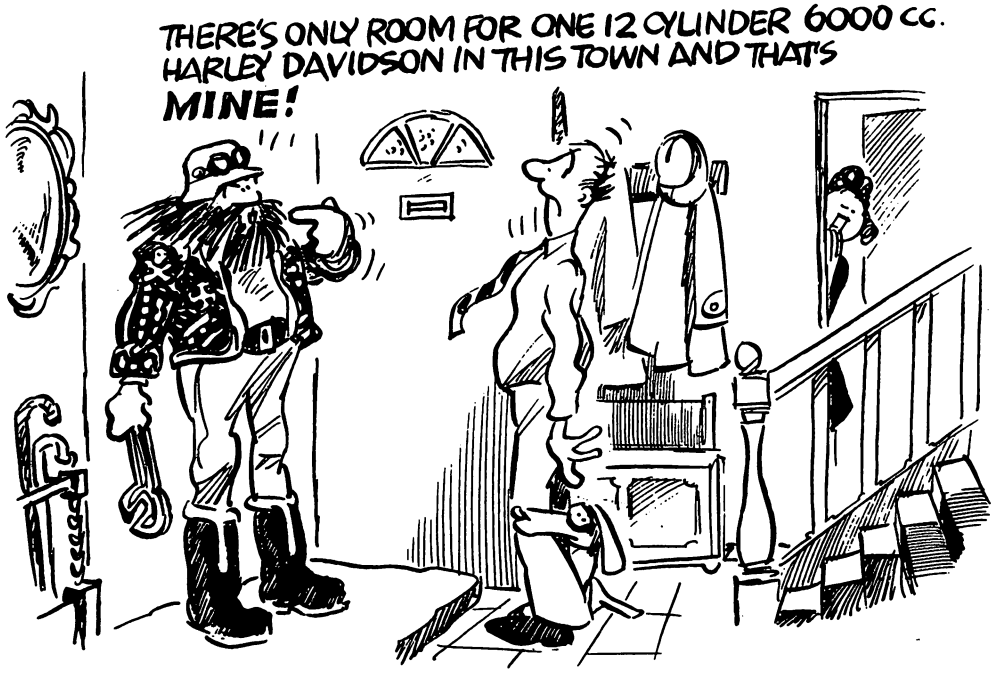
2 Volume only

A number of sound effects can be made by holding the tones at a fixed pitch and varying the volume only. To do this, set up the noise you want and then program the volume only. Here is a blast from a horn: push RETURN to make it honk.

```
10  REM  GET OUTTA MY WAY
20  POKE 36874,225
30  POKE 36875,232
40  POKE 36876,238
50  INPUT
60  POKE 36878,15
70  FOR I=1 to 1000
80  NEXT I
90  POKE 36878,0
100 GO TO 50
```

This is somewhat unsubtle. Let's have a motorcycle approach us at high speed, pass by and disappear. Because of the 'Doppler effect', we have to adjust the pitch at the moment of passing. Notice how the volume is wound up faster and faster as it approaches, and more slowly as it fades into the distance. This is more realistic.

```
10  REM  HELL'S ANGELS
20  REM  WHISTLES OFF
30  POKE 36876,0
40  POKE 36875,0
50  REM  MOTOR ON
60  POKE 36874,210
70  POKE 36877,210
80  REM  HERE HE COMES
90  FOR I=1 TO 15
100 POKE  36878,I
110 FOR K=1 TO 900/I:NEXT K
120 NEXT I
130 REM  THERE HE GOES
140 POKE 36874,180
150 POKE 36877,180
160 FOR I=15 TO 1 STEP -1
170 POKE 36878,I
180 FOR K=1 TO 900/I:NEXT K
190 NEXT I
200 REM  SILENCE
210 POKE 36874,0
220 POKE 36877,0
230 POKE 36878,0
```



And finally, a bit of nostalgia. A choo-choo train. Here the growler is used alone. Its volume is wound up and down quickly.

```
10  REM  I THINK I CAN
20  POKE 36877,242
30  REM  WIND UP FAST
40  FOR I=1 TO 15
50  POKE 36878,I
60  NEXT I
70  REM  AND DOWN AGAIN
80  FOR I=15 TO 0 STEP -1
90  POKE 36878,I
100 NEXT I
110 GO TO 40
```

After you have run this program, you should turn off the growler with

```
POKE 36878,0
```

EXERCISE:

If you do the same thing using tones it can sound like a bell. Wind it up even faster, and down more slowly. Try to choose a combination of tones to make a good bell-like sound.

3 Pitch only

You can wind the pitch up and down with the volume constant. With apologies to the girls who may feel sexually exploited, here is a wolf whistle:

```
10  REM  MALE CHAUVINIST
20  POKE  36878,15
30  FOR P=180 TO 240
40  POKE 36876,P
50  NEXT P
60  POKE 36876,0
70  FOR P=1 TO 100:NEXT P
80  FOR P=200 TO 240
90  POKE 36876,P
100 NEXT P
110 FOR P=240 TO 180 STEP -0.5
120 POKE 36876,P
130 NEXT P
140 POKE 36876,0
150 POKE 36878,0
```

4 More complex sounds

The most sophisticated effects will vary the pitch and volume together. In this blast of fresh air, the pitch and volume of the generators rise and fall together.

```
10  REM  CHILLY ISN'T IT
20  FOR V=5 TO 15
30  POKE 36874,172+V*2
40  POKE 36877,172+V*4
50  POKE 36878,V
60  FOR L=1 TO 100:NEXT L
70  NEXT V
80  FOR V=15 TO 1 STEP -1
90  POKE 36874,172+V*2
100 POKE 36877,172+V*4
110 POKE 36878,V
120 FOR L=1 TO 100:NEXT L
130 NEXT V
140 POKE 36874,0
150 POKE 36877,0
160 POKE 36878,0
```

5 Music be the food of life

We can write a program to play a tune. Let us suppose that the notes are tucked away in a series of DATA statements. We can make the program slightly more sophisticated, by having the first item in the DATA statements tell us the tempo. As in the previous chapter, we read the notes one at a time and play them. So that we can make tunes of any length, we will signal the end with an impossible negative note.

```
10  REM  PLAY ANY TUNE
20  REM  FIRST READ TEMPO
30  READ TM
40  REM  PLAY A NOTE
50  READ N1
60  REM  QUIT IF NEGATIVE
70  IF N1<0 THEN 160
80  POKE 36878,15
90  POKE 36876,N1
100 FOR J=1 TO TM
110 NEXT J
120 REM  A LITTLE GAP
130 POKE 36878,0
140 GO TO 50
150 REM  THE QUIT BIT
160 POKE 36876,0
170 END
```

Here is a table telling us the approximate values which will produce each note; they are not perfect, particularly near the top.

Note	Bottom Octave	Middle Octave	High Octave	Top Octave
C	135	195	225	240
C#	143	199	227	241
D	147	201	228	.
D#	151	203	229	.
E	159	207	231	.
F	163	209	232	not
F#	167	212	233	useful
G	175	215	235	.
G#	179	217	236	.
A	183	219	237	.
A#	187	221	238	.
B	191	223	239	.

Here is a little hornpipe:

```
200  REM  A SAILORS HORNPIPE
210  REM  THE TEMPO
220  DATA 75
230  REM  THE TUNE
240  DATA 215
250  DATA 207,195,195
260  DATA 175,195,195
270  DATA 207,195,207
280  DATA 215,209,207
290  DATA 209,201,201
300  DATA 183,201,201
310  DATA 209,201,209
320  DATA 219,215,209
330  DATA 207,195,195
340  DATA 175,195,195
350  DATA 207,195,207
360  DATA 215,209,207
370  DATA 209,207,209
380  DATA 201,215,209
390  DATA 207,195,195
400  DATA 195,0,-1
```

EXERCISE:

Try it. Here is a program which is quite a bit too long for your screen. Look at the LIST command in the Appendix to see how you can LIST any part of the program you want.

Since VIC has four voices, let's make the program even better. Although it is going to make our DATA list rather long, we will read a chord rather than a note using all three tones.

```
 10  REM  PLAY CHORDS
 20  REM  READ TEMPO
 30  READ TM
 40  REM  AND START PLAYING
 50  READ N1,N2,N3
 60  REM  QUIT IF NEGATIVE
 70  IF N1<0 THEN 170
 80  POKE 36878,15
 90  POKE 36874,N1
 92  POKE 36875,N2
 94  POKE 36876,N3
100  FOR J=1 TO TM
 :       :
 :       :
```

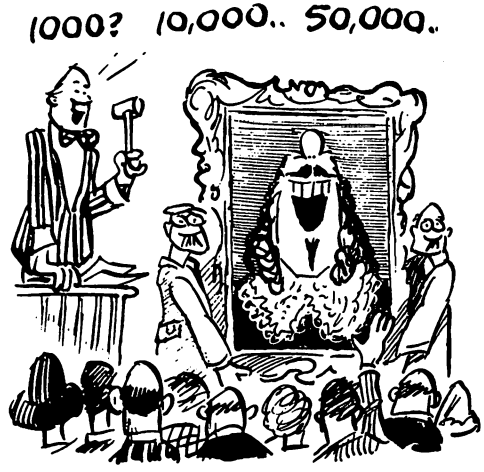
You can see that you need only change line 50, and add lines 92 and 94 to the earlier program. The DATA of course is different. Here is a famous chorale by J S Guesswho. I have left out the line number and the word DATA to squash it all in:

```
Tempo
100

First phrase    Third phrase    Fifth phrase    Seventh phrase
203,187,215     203,187,215     215,195,229      219,195,229
217,203,225     217,203,225     217,209,228      221,209,228
215,203,221     215,203,221     215,201,221      201,187,232
195,203,217     195,203,217     217,203,225      215,187,229
203,187,215     203,187,215     209,217,228      221,209,228
179,203,209     179,203,209     195,217,229      203,175,225
187,201,209     187,201,209     179,209,229      209,183,225
203,187,215     203,187,215     175,203,229      187,209,228
203,187,215     203,187,215     175,203,229      187,209,228
0,0,0           0,0,0           0,0,0            0,0,0

Second phrase   Fourth phrase   Sixth phrase    Eighth phrase
201,191,228     201,191,228     203,215,221      203,187,215
195,175,229     195,175,229     217,203,225      195,203,217
203,195,229     203,195,229     203,215,221      187,203,215
215,195,228     215,195,228     209,195,217      179,203,209
215,209,228     215,209,228     217,209,217      187,209,221
195,203,225     195,203,225     225,207,215      203,187,215
195,203,225     195,203,225     225,207,215      203,187,215
195,203,225     195,203,225     225,207,215      203,187,215
195,203,225     195,203,225     225,207,215      203,187,215
0,0,0           0,0,0           0,0,0            0,0,0
                                                 -1,0,0
```

When you run this program you may feel that the little breaks between chords spoil the flow. These breaks are caused by line number 130, which you could remove.

Eleven

A PICTURE IS WORTH A THOUSAND WHAT?



1 Or was it something to do with Fiona of Troy?

Anyway, one of the delightful features of VIC is its ablity to launch ships, sorry, I mean make pictures. There are two ways of doing this. In this chapter we are going to operate through PRINT statements, but in the next one you will see that it can all be done using POKE.

2 All about PRINT

You have probably noticed that when you use the PRINT statement to put values on the screen, they mostly appear in two 'fields'. The screen used by VIC is 22 columns wide, and the first field used by numbers is made up of columns 1-11 and the second by columns 12-22. If you don't believe it, use this example to convince yourself:

```
10  INPUT A,B
20  PRINT A,B
30  GO TO 10
```

If it is short enough, you can also get a message to appear in one column and numbers in the other:

```
10  LET I=10
20  PRINT "MESSAGE",I
30  LET I=I*10
40  FOR J=1 TO 500
50  NEXT J
60  GO TO 20
```

When you try his example you will see that the columns continue to line up even after VIC switches to exponential form for large numbers.

You can force VIC not to use these two fields. The secret is the semicolon. Try these:

```
10  PRINT "ONE","TWO"          10  PRINT "ONE";"TWO"
20  GO TO 10                   20  GO TO 10
```

If there is a semicolon in a PRINT statement, VIC squashes the output together.

Another great semicolon feature allows you to continue printing on the same line. Normally a PRINT statement starts a new line on the screen. But if you leave a semicolon dangling at the end of a PRINT statement, the next PRINT continues on the same line. You can do something similar with commas, but it isn't as useful. Try these and see the difference. The effect is shattering.

```
10  FOR I=1 TO 5      10  FOR I=1 TO 5      10  FOR I=1 TO 5
20  PRINT I           20  PRINT I,          20  PRINT I;
30  NEXT I            30  NEXT I           30  NEXT I
```

This has many great possibilities. In Chapter 16 I will show the geniuses how to make graphs of mathematical functions. Here is a sneak preview which everyone should enjoy:

```
10  FOR I=1 TO 20
20  FOR J=1 TO I
30  PRINT "*";
40  NEXT J
50  PRINT
60  NEXT I
```

I call that one 'Climb Every Mountain'. See how the clever inner loop prints the right number of stars! See how mild mannered line 50 puts us on a new line just in the nick of time! Overwhelming.

One more little thing you may need to know. When you PRINT a message and follow it with a value, or another message in the same PRINT statement, you don't need any punctuation. VIC will assume you mean a semicolon:

```
10  FOR I=1 TO 10
20  PRINT "S" "E" "E" I
30  NEXT I
```

3 Free Fantastic Functions - TAB, SPC, POS

Well, TAB is fantastic anyway. TAB can only be used in a PRINT statement, and makes your output jump immediately to the column you want. This is very useful in making pretty pictures. Here's an example:

```
10  FOR I=0 TO 21
20  PRINT TAB(I); "*"
30  NEXT I
```

Notice the semicolon to squash the star into the column we want.

The SPC function has a similar effect. SPC(expression) makes the PRINT statement skip forward by the number of spaces that would have been given by INT(expression).

If you want to know where your PRINT statement is on the screen at the moment, use POS(X) like any ordinary function to tell you. The X can have any value and has to be there, but it isn't used by the POS function. POS tells you the column number you are about to use. Can you predict the value of POS in this example? If you can, your understanding is profound.

```
10  PRINT "FIRST"
20  FOR I=1 TO 5
30  PRINT SPC(I);
40  NEXT I
50  PRINT POS(X)
```

4 Roll over, Rembrandt

Now we know all about the PRINT statement. But we have lots of things that we can print that we don't know about yet. You will have noticed a lot of peculiar hieroglyphics along the front of the keys on your VIC keyboard. All of these are shapes that you can use on the screen to paint pictures. If your VIC is in graphics mode, as it is when you turn it on, you get the shapes on the right by holding the SHIFT key down while you press the graphics key you want. You get the shape on the left by holding down the key with the Commodore trademark on it while you press the key you want. Try this and see.

What is more important is that you can get the shapes into a message that you use in a PRINT statement. Try this:

```
10  PRINT "♣♦♥♠";
20  GO TO 10
```

To draw a picture, first we have to clear the screen and then, using combinations of PRINTs with TABs and selected graphics characters from the keyfronts, we draw whatever we like. The range of shapes allows us to make up just about any outline or solid object we want. In colour too, as we will see in the next chapter. How do we clear the screen? Like this:

```
PRINT "Ω"
```

special character consisting of SHIFT and CLR/HOME. It appears on your screen something like SHIFT S.

Without the shift, you move the cursor to the home position (upper left corner) without clearing the screen.

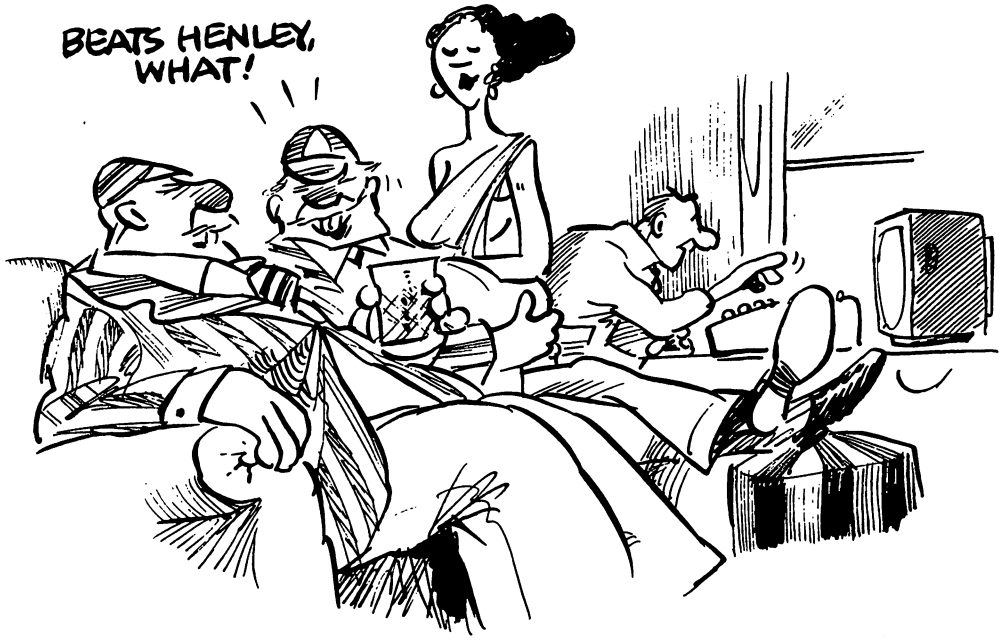
Referring back to our discussion of the Trojan Wars, here is how to draw the navigable vessel known as the millihelene:

```
10  PRINT "Ω"          i.e. SHIFT and CLR/HOME
20  PRINT "  \\\\"
30  PRINT "  ┌──\"
40  PRINT "─┤    ●"
50  PRINT "  └────/"
60  PRINT "  ////"
```

In the next section I'm going to row it across the screen!

5 Animation

If you make your picture move, then you have animation! To make Fiona's boat row across the screen, I am going to move the oars back and forth and move the boat across. As our boat is six symbols wide it can move 16 spaces before it gets to the other side. The way I have organised this, our oarsmen get in four strokes, during which the oars take up four positions. I have to begin my boat on the same row each time. To do this I use the CLR/HOME key without the shift, and a series of PRINT statements to get to row 8 which is where I have put the boat. I then use a TAB to get to the present position of the boat. Notice the ON...GO TO statement to select which set of oars to use. It's too easy, really! Once again, the extra FOR...NEXT loop controls the speed. Do you like the little whirlpools left behind by the oars? Why don't they disappear each time?




```
 10  PRINT "□"            i.e. SHIFT and CLR/HOME
 20  FOR SK=0 TO 3
 30  FOR OA=1 TO 4
 40  PRINT "■"            i.e. CLR/HOME on its own
 50  REM  GET TO ROW 8
 60  FOR RO=1 TO 8
 70  PRINT
 80  NEXT RO
 90  REM  WHERE TO TAB?
100  LET TA=OA+4*SK-1
105  IF TA=15 THEN 320
110  ON OA GOTO 120,140,160,180
120  PRINT TAB(TA); "  ||||" (G key)
130  GO TO 190
140  PRINT TAB(TA); " \\\\"
150  GO TO 190
160  PRINT TAB(TA); "o ||||" (M key)
170  GO TO 190
180  PRINT TAB(TA); "  ////"
190  PRINT TAB(TA); " ┌──\"
200  PRINT TAB(TA); "┤      •"
210  PRINT TAB(TA); " └────/"
220  ON OA GOTO 230,250,270,290
230  PRINT TAB(TA); "  ||||"(G key)
240  GO TO 300
250  PRINT TAB(TA); " ////"
260  GO TO 300
270  PRINT TAB(TA); "o ||||" (M key)
280  GO TO 300
290  PRINT TAB(TA); "  \\\\"
300  FOR I=1 TO 250:NEXT I
310  NEXT OA
320  NEXT SK
```

6 Push the cursor around

Just to be confusing, there is another way of doing graphics using the PRINT statement. The keys that you use to control the cursor position when you edit programs can also be put into PRINT statements. Unfortunately they produce confusing things when you LIST a program containing them - they look like things which they aren't.

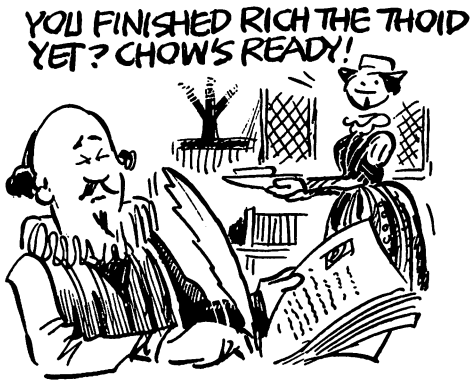
CRSR DOWN	looks like Q reversed
CRSR UP	looks like ▢
CRSR LEFT	looks like ▮▮
CRSR RIGHT	looks like ▮

Here we make a program to do a spiral - a sort of square spiral - on the screen, all by cursor control. The first FOR...NEXT loop pushes the cursor to the centre of the screen - notice the semicolon. Then starting at the centre we move the cursor right, up, left, down, printing all the time.

```
 10  PRINT "▢"                  Not the graphics key, but SHIFT CLR/HOME
 20  REM  CRSR TO CENTRE
 30  FOR I=1 TO 11
 40  PRINT "▮●";        (CRSR right and CRSR down)
 50  NEXT I
 60  REM  NOW SPIRAL
 70  FOR MV=1 TO 21 STEP 2
 80  REM  RIGHT
 90  FOR I=1 TO MV
100  PRINT "●";         (graphics character)
110  NEXT I
120  REM  UP
130  FOR I=1 TO MV
140  PRINT "▮▮▢";       (CRSR left and CRSR up)
150  NEXT I
160  REM  LEFT
170  FOR I=1 TO MV+1
180  PRINT "▮▮▮▮";      (CRSR left twice)
190  NEXT I
200  REM  DOWN
210  FOR I=1 TO MV+1
220  PRINT "▮▮●";       (CRSR left and CRSR down)
230  NEXT I
240  NEXT MV
```

Twelve

IS IT COLOUR OR COLOR?



1 Who cares?

It is a fact that Shakespeare was inconsistent in the spelling of his own name. If that is so, why should we worry? However you spell it, VIC can do it.

2 Colour the screen

When you switch VIC on, you will find that everything prints in blue, on a white screen with a cyan border. Then do this:

```
POKE 36879,44
```

Wow! A red screen with a purple border! The colours available to VIC you will see on a row of keys across the top. You will learn to use those in a moment. First of all, the screen. Give those colours the following values:

BLACK	0	ORANGE	8
WHITE	1	LIGHT ORANGE	9
RED	2	PINK	10
CYAN	3	LIGHT CYAN	11
PURPLE	4	LIGHT PURPLE	12
GREEN	5	LIGHT GREEN	13
BLUE	6	LIGHT BLUE	14
YELLOW	7	LIGHT YELLOW	15

Just to confuse you, the numbers in the first column are one less than the numbers on the tops of the keys. You can have your border any colour from 0-7 and your screen any colour from 0-15. Take

8+16*(Screen Colour) + Border Colour

and POKE it into memory address 36879. Try this:

```
10  REM  PRETTY, PRETTY
20  FOR I=0 TO 7
30  FOR J=0 TO 15
40  POKE 36879,8+16*J+I
50  FOR K=1 TO 100
60  NEXT K
70  NEXT J
80  NEXT I
```

The normal screen and border combination is POKE 36879,27.

3 Now the characters

Characters are normally printed in blue. You can change this by holding down the CTRL key and pressing any colour key. When you put this in a PRINT message it changes the colour of everything that follows it. Once again, when you type one of these in quotes you get a confusing display. For example, CTRL RED will come out as the reverse of F. You can get used to this.

The other thing you can do to characters is reverse their image. You put CTRL RVS ON or CTRL RVS OFF to turn reversal on and off. When you turn it on in this way, it only stays on for the rest of one PRINT statement unless you keep it on with a semicolon. To understand reversal, try this example:

```
10  REM  REVERSE, REVERSE
20  PRINT "□"                    i.e. SHIFT and CLR/HOME
30  PRINT "■THIS IS REVERSE"     CTRL and RVS ON
40  PRINT "THIS IS NORMAL"
50  PRINT "■REVERSE STAYS";      CTRL and RVS ON
60  PRINT " ON THIS TIME"
```

These methods are of limited usefulness in getting characters and symbols to change colour. A better way follows.

4 The screen is a map!

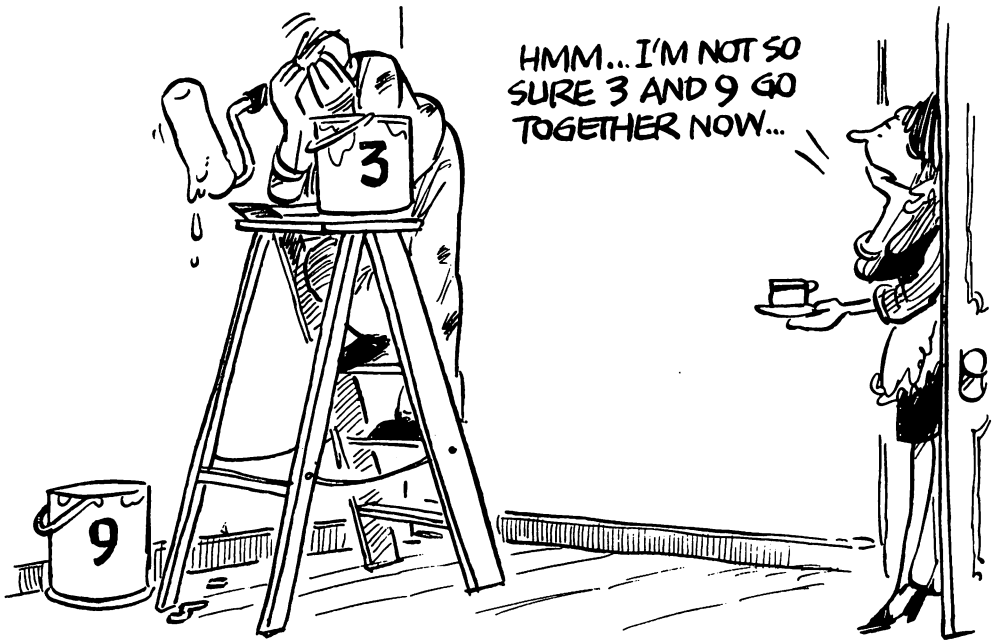
Actually two maps. In the memory of the VIC there is a map of every spot on the screen. As there are 23 rows and 22 columns, that makes 506 addresses. The screen map which holds the symbols begins at address 7680 and is always an exact map of the screen. The other screen map holds the colours and begins at 38400. To put something on the screen you need to POKE both a screen position and a colour. Try this:

```
10  PRINT "□"              i.e. SHIFT and CLR/HOME
20  POKE 7933,160
30  POKE 38653,2
```

Wow! Character number 160 is a solid block and colour number 2, as before, is red.

Address 7680 is the top left of the screen. Address 7681 is the second space in the top row, and so on. When you finish the top row, you appear in the second row. Do this:

```
10  REM  FILL SCREEN
20  FOR I=1 TO 506
30  POKE 7679+I,102
40  POKE 38400+I,I-INT(I/7)*7
50  FOR J=1 TO 100:NEXT J
60  NEXT I
```



Fantastic! See the remainder at line 40? It always gives a colour number between 0 and 7.

To find the address of a particular screen position, do this:

7680 + Column Number + 22*(Row Number) for the symbol

0 to 21 0 to 22

and

38400 + Column Number + 22*(Row Number) for the colour

0 to 21 0 to 22

To find what value to POKE to get a particular symbol, look in the Appendix. If you want the 'reverse' of a given symbol, add 128 to its POKE value.

EXAMPLE:

Run this to make the flag of St Andrew. The remarks tell you how it works. Look up the graphics symbols to see what they are.

```
 10  REM  SCOTLAND THE BRAVE
 20  REM  WHITE ON BLUE
 30  POKE 36879,105
 40  REM  CLEAR SCREEN
 50  PRINT "Ω"                    i.e. SHIFT and CLR/HOME
 60  REM  I IS ROW NO.
 70  FOR I=0 TO 22
 80  REM  LEFT TO RIGHT          200  POKE WS,160
 90  LET WS=7680+23*I            210  POKE WS+1,105
100  LET WC=38400+23*I           220  POKE WS-1,233
110  POKE WS,160                 230  POKE WC,1
120  POKE WS-1,95                240  POKE WC+1,1
130  POKE WS+1,223               250  POKE WC-1,1
140  POKE WC,1                   260  NEXT I
150  POKE WC-1,1                 270  REM  TIDY UP FLAWS
160  POKE WC+1,1                 280  POKE 7910,160
170  REM  RIGHT TO LEFT          290  POKE 7933,160
180  LET WS=7701+21*I            300  POKE 8162,95
190  LET WC=38421+21*I           310  POKE 8141,223
                                 320  DELAY A BIT
                                 330  FOR I=1 TO 10000:NEXT I
```

5 Come fly with me

This, of course, opens up a whole new way of making pictures, by POKEing values directly into the screen colour map. Most games work this way. You actually have three ways of making pictures: by PRINT statements, by cursor movement, and by POKE. You can use them all if you want; suit yourself.

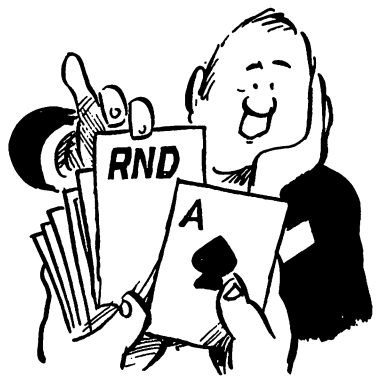
In this program a squadron of jets fly overhead, each one lower than the one before it. I do this entirely by PEEK and POKE. Notice how the jets move along the screen. This is done by shifting the fuselage along and POKEing a new tail. Eventually one flies too low and hits the hill. The explosion is a modification of the program that I used in the last chapter, only PEEKing and POKEing rather than using the cursor. IS is the screen position of the spiral which gets pushed around by a series of FOR...NEXT loops. We'll use this again.

```
 10  REM  CRUISE MISSILES
 20  PRINT "■■"              i.e. SHIFT and CLR/HOME, CTRL GRN
 30  REM  BLUE SKY
 40  POKE 36879,105
 50  REM  GET TO ROW 12
 60  FOR I=1 TO 11
 70  PRINT
 80  NEXT I
 90  REM  DRAW THE HILL
100  FOR I=1 TO 11
110  PRINT
120  PRINT "■";               CTRL and RVS ON
130  PRINT TAB(11-I);         Don't forget the semicolons
140  FOR J=1 TO 2*I-1
150  PRINT " ";
160  NEXT J
170  NEXT I
180  REM  MAKE THE JET
190  PRINT "■■";             CLR/HOME (without SHIFT), CTRL RED
200  PRINT "■■"                  CTRL RVS ON
210  PRINT "■      ■"            CTRL RVS ON
220  REM  ENGINE NOISE
230  POKE 36878,5
240  POKE 36877,240
250  REM  SET SCREEN RED
260  FOR I=0 TO 273
270  POKE 38400+I,2
280  NEXT I
```

```
290  REM  NOW FLY IT
300  REM  NS IS NOSE POSITION
310  FOR NS=26 TO 273
320  REM  COPY FUSELAGE
330  FOR J=NS TO NS-5 STEP -1
340  LET JC=7680+J
350  POKE JC+1,PEEK(JC)
360  NEXT J
370  REM  POKE TAIL
380  POKE 7680+NS-26,32
390  POKE 7680+NS-25,223
400  NEXT NS
600  REM  GRAND EXPLOSION
610  REM  BOOMING SOUND
620  POKE 36877,130
630  POKE 36878,15
640  REM  CENTRE OF SCREEN
650  LET IS=274
660  POKE 38400+IS,0
670  POKE 7680+IS,160
680  REM  N CONTROLS SPIRAL
690  FOR N=1 TO 11 STEP 2
700  REM  CL IS COLOUR
710  LET CL=INT(N/2)
720  REM  GO RIGHT
730  FOR L=1 TO N
740  LET IS=IS+1
750  POKE 7680+IS,160
760  POKE 38400+IS,CL
770  NEXT L
780  REM  GO UP
790  FOR L=1 TO N
800  LET IS=IS-22
810  POKE 7680+IS,160
820  POKE 38400+IS,CL
830  NEXT L
840  REM  GO LEFT
850  FOR L=1 TO N+1
860  LET IS=IS-1
870  POKE 7680+IS,160
880  POKE 38400+IS,CL
890  NEXT L
900  REM  AND DOWN
910  FOR L=1 TO N+1
920  LET IS=IS+22
930  POKE 7680+IS,160
940  POKE 38400+IS,CL
950  NEXT L
960  REM  WIND DOWN NOISE
970  POKE 36878,16-N
980  NEXT N
990  REM  TIDY UP
1000 POKE 36877,0
1010 POKE 36878,0
1020 POKE 36879,27
```

Thirteen

PURELY BY CHANCE



1 Introducing Randy Random

Randy Random is a silly fellow. He never knows where he is going next. But he can be useful, and fun. He is actually a function, called RND. If you write something like

```
10  FOR I=1 TO 200
20  PRINT RND(1);
30  NEXT I
```

your screen will fill up with apparently unrelated numbers. This is because our friend RND (for short) is a 'random number generator'. The result is always a number between 0 and 1 that you can't predict. You can use this for all kinds of interesting things.

2 Scaling our friend Randy

Chances are that you don't want a number between 0 and 1. You probably want one between some minimum value and some maximum value. Call the minimum MN (Minnie Minimum?) and the maximum MX (Maximillian Maximum!). To get a random number RN which is always between Minnie and Maximillian, write

```
line number  LET RN = MN + RND(X) * (MX - MN)
```

The X in RND(X) controls how the random number generator gets started, and how it goes on after that. To start it, or 'seed it', use RND(X) with a negative value of X. If you use it with the same seed each time, you will get the same random numbers out - evidently they aren't really

random, they just look it. So a good way to start Randy going is with a value from the VIC clock - we haven't said too much about that yet - whose name is TI. Try these:

```
10  REM  USE -3
20  LET I=RND(-3)
30  FOR I=1 TO 20
40  PRINT RND(1)
50  NEXT I
```

```
10  REM  USE CLOCK
20  LET I=RND(-TI)
30  FOR I=1 TO 20
40  PRINT RND(1)
50  NEXT I
```

You will always get the same result when you run the one on the left. You will always get a different result when you use the one on the right.

Notice that both these programs call RND(1) once it gets started. This is the usual thing - although any positive X in RND(X) will do.

3 Do you like modern music? Neither do I!

Using the RND function you can do all sorts of enjoyable things. Let's write some modern music - I heard a concert on the radio while I was writing Chapter 12 that has inspired me to have a go myself - roll over Rachmaninov! We can do even better (worse?) than Hans Werner Henze because our orchestra is not limited to the notes of a conventional scale. Here I ask Randy and his friends to give me a chord duration in the range 0-200, four notes in the range 128-254, and a loudness all the way from pianissimo to horribly loud (1-15). Play on. Give me excess of it that, surfeiting, the appetite may sicken so and die...

```
 10  REM  WELCOME TO THE
 20  REM  VIC CONCERT HALL
 30  LET T=RND(1)*200
 40  POKE 36874,128+127*RND(1)
 50  POKE 36875,128+127*RND(1)
 60  POKE 36876,128+127*RND(1)
 70  POKE 36877,128+127*RND(1)
 80  POKE 36878,1+14*RND(1)
 90  FOR I=1 TO T
100  NEXT I
110  GO TO 30
```

4 Do you like modern art? At least it's quiet.

In this program, I am going to poke a random character with a random colour at a random place on the screen, over and over again.

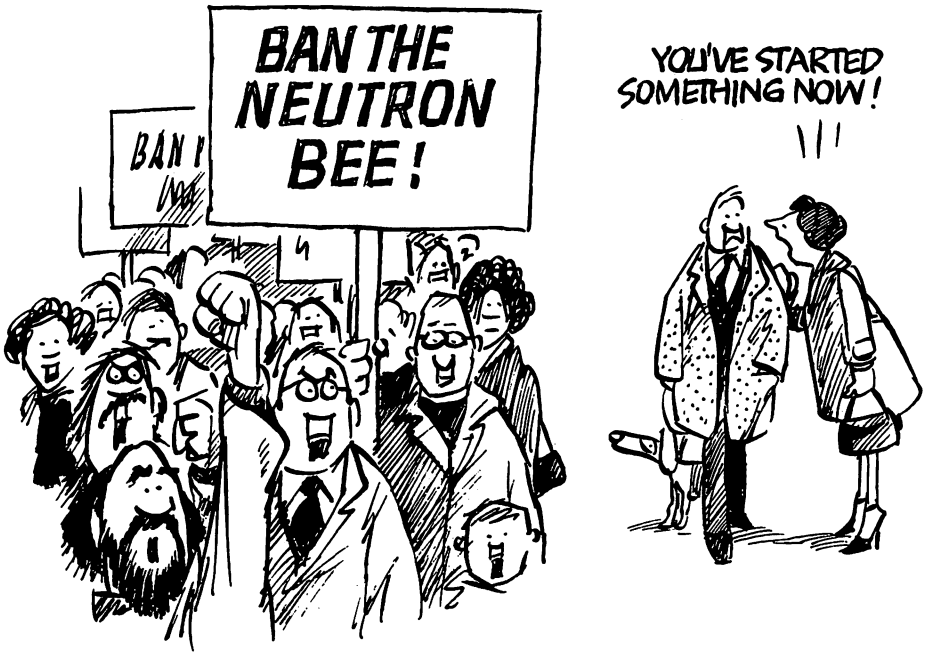
```
10  REM  A VIVID DREAM
20  PRINT "□"  SHIFT and CLR/HOME
30  LET PL=506*RND(1)
40  POKE 7680+PL,256*RND(1)
50  POKE 38400+PL,8*RND(1)
60  GO TO 30
```

EXERCISE:

I think these last two programs would go rather well together. Combine them.

5 The fable of the Bomb and the Buzzing Bee

There is a neutron bomb at column 11 of row 11 on your screen. A bee is flying around on your screen. Eventually...



In this program, I use two random numbers to tell me where the bee is going next. It can move one place to the left or right each time which is by one address in the screen map. It can also move one place up or

down each time, which is 22 addresses in the screen map. I'm going to leave a row of dots where the bee has been. I have to be careful not to let it fly outside the screen, and you will hear it buzzing louder as it flies near the bomb and softer as it flies away. You use the grand explosion from the last chapter when it eventually gets there.

```
 10  REM  UNFAIR TO BUGS
 20  PRINT "□"  SHIFT and CLR/HOME
 30  REM  BEE, BOMB, BUZZ
 40  POKE 7795,42
 50  POKE 38515,4
 60  POKE 7933,81
 70  POKE 38653,2
 80  POKE 36875,229
 85  POKE 36876,200
 90  POKE 36878,1
100  REM  BEE IS AT SC
110  LET SC=115
120  REM  BEE WILL MOVE
130  REM  BY X AND Y
140  LET X=INT(-1+3*RND(1))
150  LET Y=INT(-1+3*RND(1))
160  REM  TRAIL OF BEE
170  POKE 7680+SC,46
180  POKE 38400+SC,0
190  LET SC=SC+X+22*Y
200  REM  KEEP ON SCREEN
210  IF SC>=0 THEN 230
220  LET SC=0
230  IF SC<506 THEN 250
240  LET SC=505
250  POKE 7680+SC,42
260  POKE 38400+SC,4
270  REM  HOW FAR AWAY
280  LET RO=INT(SC/22)
290  LET CO=SC-RO*22
300  LET DS=(RO-11)↑2+(CO-11)↑2
310  LET BZ=15/(1+DS/18)
320  POKE 36878,BZ
330  IF INT(SC)<>253 THEN 140
```

This program continues with exactly the same grand explosion as in the airplane crash of Chapter 12. Well, if all the great composers re-used material, why shouldn't we?

Fourteen

MAKE A LIST

1 Lists and subscripts

In BASIC there is what is called an 'array' facility by which a list of values can be organised under one variable name. You can then get at any member of the list using a subscript. The statement

```
60  LET X=SQR(X1 ↑ 2+Y1 ↑ 2)
```

uses variable names X, X1 and Y1 which stand for only one value each. However, if you write

```
60  LET X(J)=SQR(X1 ↑ 2+Y1 ↑ 2)
```

the variable X represents a list of values and you are assigning a value to member number J of the list. BASIC automatically recognises when a variable is a list or array variable simply by the use of a subscript like (I) with it. Therefore

X(9) refers to the ninth entry in the list called X
AB(I+J) refers to the (I+J)th value in the list called AB

A subscript can be any expression, but it is obvious that it has to be interpreted as a positive integer. In BASIC the lowest subscript is 0. If the value of a subscript is not an integer, it is chopped off to make one. This means that

PQ(4.66) refers to the fourth entry in the list called PQ

VIC will check every subscript as it is used, and if it is less than zero or too large, the program will not be allowed to continue. You should

not use the same variable name to represent both an ordinary value and a list in a program. We'll come to the question of how large a list can be a bit later.

2 Using a list

Suppose Narrowdale Prison has 4 cells - a bit small, but this one is for long stay VIPs such as heads of state who inflict savage monetarist policies on the people. Each prisoner is given a number by which we know them. A list could be used to represent the prisoner numbers. This list could be described as

```
The prisoner in cell 1 is number 728
The prisoner in cell 2 is number 213
The prisoner in cell 3 is number 900
The prisoner in cell 4 is number 463
```

Let's call the list of prisoner numbers PN. There are four cells, and so we need to define PN(1), PN(2), PN(3), and PN(4). Remember the DATA statement from Chapter 9? Here is a little program which reads the prisoner numbers from a DATA list, and prints them out again. In a FOR...NEXT loop the READ statement at line 40 gets the prisoner number for cell number I:

```
10  REM  JAILHOUSE ROCK
20  DATA 728,213,900,463
30  FOR I=1 TO 4
40  READ PN(I)
50  NEXT I

60  REM  NOW PRINT 'EM
70  FOR J=1 TO 4
80  PRINT PN(J)
90  NEXT J
```

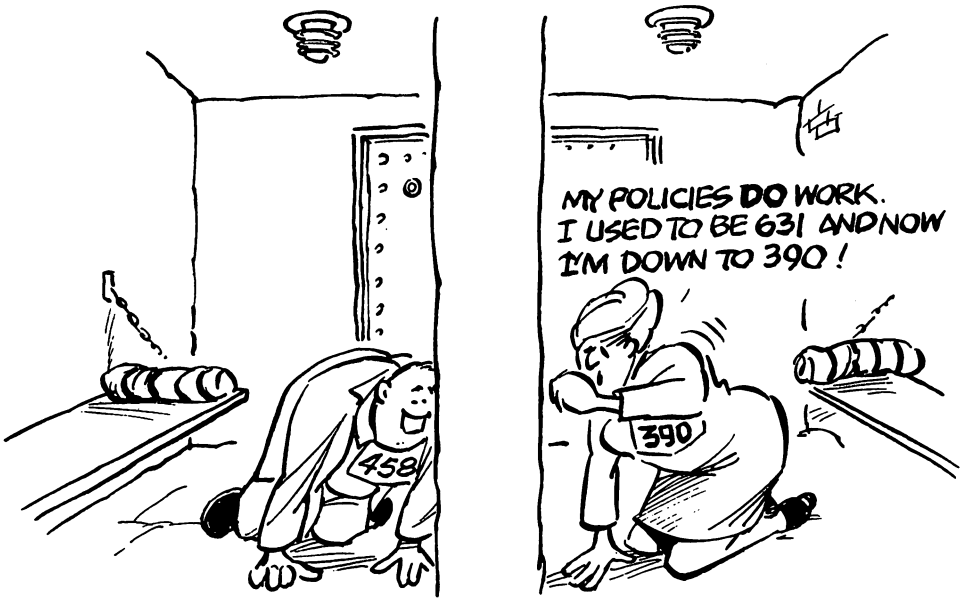
If you wanted to type the numbers in instead, you could have put

```
40  INPUT PN(I)
```

Now let's look for the highest prisoner number. Searching a list is a very common thing in computing - and there are some special ways of doing it. There is nothing special, however, about the way I'm doing it here. It is called a linear search. To do the search, we need to set aside a variable for the answer - HI in this case. We could give it as an initial value a lower prisoner value than we ever expect to find. It would be clearer, however, to use PN(1) as the first value of HI. We then search along the list for a higher number, and put into HI if we find one. Graft

this onto the end of the previous program:

```
100  REM  FIND HIGHEST
110  LET HI=PN(1)
120  FOR J=2 TO 4
130  IF PN(J)<=HI THEN 150
140  HI=PN(J)
150  NEXT J
160  PRINT "LARGEST IS "HI
```



Easy? You should be able to see how this works. Finding the largest or smallest value from a list is something you often have to do.

3 99 years is almost for life

It is common for programs to deal with several lists at once, which are related; for example we might also wish to know the age and remaining sentence of each prisoner:

Cell	1	2	3	4
Prisoner Number	728	213	900	463
Remaining Sentence	37	99	61	12
Age	28	41	16	24

Call the age AG, and the sentence SN. In this little fragment of program, one data statement is used for each prisoner:

```
10  REM  PRISON RECORDS
20  DATA 728,37,28
30  DATA 213,99,14
40  DATA 900,61,16
50  DATA 463,12,24
60  REM  READ 'EM
70  FOR I=1 TO 4
80  READ PN(I),AG(I),SN(I)
90  NEXT I
100 PRINT " NO. AGE SENT."
110 FOR J=1 TO 4
120 PRINT PN(J);AG(J);SN(J)
130 NEXT J
```

EXERCISE:

Now you do some work. Search this data here to find the prisoner who will be the youngest when released. Show on the screen the cell number, prisoner number, present age, remaining sentence, and age when released.

4 So how long is a list? - DIM

We have ignored so far the space taken up by a list. BASIC assumes that any list you mention contains 11 values, with subscripts from 0 to 10, and this is quite often suitable as you are unlikely to miss the wasted space if your list is shorter. However, you can make a list longer or shorter than 11 by using the DIM statement. No, it's not an insult, it stands for DIMension. You simply put

line number DIM name(size), name(size), ...

to tell BASIC the size of any number of lists. You can have any number of DIM statements as long as you define the size of a given list before you first use it - otherwise you're stuck with 11 and VIC will object anyway. It is usual to put the DIM statements for a program at the beginning. If you mention a particular name more than once in your DIM statements, VIC won't like it.

EXAMPLES:

```
5  DIM PN(4),AG(4),SN(4)
```

reserves exactly four spaces for the lists that make up the Narrowdale prison database. You could put this in the previous example to save a bit of memory space.

```
69  DIM XR(200)
```

reserves 200 spaces for a biggish list called XR.

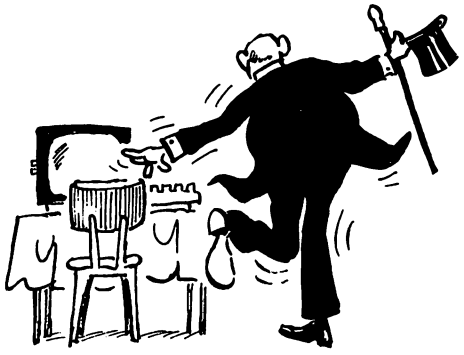
5 Westminster Chimes

Another musical program. The notes that make up the Westminster Chimes are defined in DATA statements and we read them into an array called WC. We then pick them out and play them one at a time. To make a bell-like sound the volume is turned on suddenly and then wound down slowly for each note. In Chapter 19 we're going to make VIC into a chiming clock, VIC BEN.

```
 10  REM  DING DONG
 20  DATA 239,237,235,228
 30  DATA 228,237,239,235
 40  DATA 239,235,237,228
 50  DATA 228,237,239,235
 60  DIM WC(16)
 70  REM  READ DONGS
 80  FOR I=1 TO 16
 90  READ WC(I)
100  NEXT I
110  REM  NOW DO DINGING
120  POKE 36878,0
130  POKE 36877,0
140  POKE 36876,0
150  FOR N=1 TO 16
160  POKE 36874,WC(N)
170  LET P=255-(255-WC(N))*2 ↑ (5/3)
180  POKE 36875,P
190  FOR I=15 TO 1 STEP -1
200  POKE 36878,I
210  FOR J=1 TO 240/I
220  NEXT J
230  NEXT I
240  POKE 36878,0
250  NEXT N
```

There are two interesting things about this program. First of all I use 36874 as the bell note, and 36875 as an overtone a musical third above it. Line 170 calculates the POKE value for the overtone. This took a bit of figuring out, and it turns out that the program can only do a realistic bell noise for DATA values from 215 and up. Secondly, observe how the volume is turned down more slowly by the FOR statement at line 210. I did the same thing in Chapter 10 with a motorcycle noise.

Fifteen



SORT IT OUT

1 Do the VIC shuffle

With all this music around, I suppose you think that I am referring to some kind of dance. Sorry. Once you have got data into lists you are probably going to want to move it around. You have to be careful how you do this.

Back in the jailhouse, suppose you decide you want to rearrange your prisoners, so that each moves down one cell number except for the prisoner in cell 1, who moves to cell 4. In a computer program, you can only move one value at a time. In the prison, this would be a bit like having only one guard to move the prisoners. Here is what you would have to do:

The prisoner in cell 1 goes in temporary accommodation
The former prisoner in cell 2 becomes the new prisoner in cell 1
The former prisoner in cell 3 becomes the new prisoner in cell 2
The former prisoner in cell 4 becomes the new prisoner in cell 3
and finally
The former prisoner from cell 1 becomes the new prisoner in cell 4

If we have associated arrays, we have to shuffle them at the same time. Some prisoners would not be too happy if your prison computer caused them to inhabit the previous inmate's sentence. Others, no doubt, would be delighted! This program does it properly; add it to the bit from the previous chapter which defines the lists and see what happens:

```
140  REM  MOVE NO 1 OUT
150  LET P1=PN(1)
160  LET A1=AG(1)
170  LET S1=SN(1)
180  REM  MOVE OTHERS DOWN
190  FOR I=1 TO 3
200  LET PN(I)=PN(I+1)
210  LET AG(I)=AG(I+1)
220  LET SN(I)=SN(I+1)
230  NEXT I
240  REM  NO 1 COMES BACK
250  LET PN(4)=P1
260  LET AG(4)=A1
270  LET SN(4)=S1
```

```
280  REM  SHOW THE RESULT
290  PRINT " MOVED TO"
300  FOR J=1 TO 4
310  PRINT PN(J);AG(J);SN(J)
320  NEXT J
```

Notice how the prisoners have been moved down one place. This has been done between lines 190 and 230 in ascending order of cells. On the other hand, if you wanted to move them up, you should go backwards. Why?

To shuffle up
    save the top one
    move others in descending order
To shuffle down
    save the bottom one
    move others in ascending order

2 The great sorting problem

Arranging things in order is one of the great computer problems keeping computer geniuses occupied. It is very easy to do this, but it is hard to do it efficiently. I am going to show you two simple (but inefficient) methods.

Putting things in order is called sorting. The things to be sorted are called 'keys'. In Chapter 19 we'll put words in alphabetical order. Here we do it with numbers.

Bubble sorting is the simple one. We have a list LI of, say, 8 values, and we want LI(1) to be the smallest, LI(8) the largest. To do this we simply compare LI(1) with LI(2), and switch them if we need to. Now look at LI(2) and LI(3) and do the same, and continue along until we have dealt with LI(7) and LI(8). After we have done this, the numbers will be more ordered than they were, and we know that LI(8) has the largest value for sure - because it will have been carried along like the largest bubble rising more quickly. (Is this physically true? Falling objects go at the same speed in a vacuum. Do large bubbles rise faster?) Anyway, we now make another pass, until we know that LI(7) has the second largest number. And so on. Here it is; you can sort any number of keys from 2 to 10 using this program. For a larger number, put in a DIM statement.

```
 10  REM  BUBBLE POWER
 20  PRINT "HOW MANY KEYS"
 30  INPUT N
 40  PRINT "TYPE IN"N"KEYS"
 50  PRINT "ONE AT A TIME"
 60  FOR J=1 TO N
 70  INPUT LI(J)
 80  NEXT J
 90  PRINT "HERE GOES"
 95  PRINT "■"  SHIFT and CLR/HOME
100  FOR J=1 TO N
110  PRINT LI(J);
120  NEXT J
130  PRINT
140  REM  SORT STARTS HERE
150  FOR I=N-1 TO 1 STEP -1
160  FOR J=1 TO I
170  IF LI(J)<=LI(J+1) THEN 260
180  REM  SWITCH 'EM
190  LET LT=LI(J)
200  LET LI(J)=LI(J+1)
210  LET LI(J+1)=LT
215  PRINT "■"  CLR/HOME
220  FOR K=1 TO N
230  PRINT LI(K);
240  NEXT K
250  FOR K=1 TO 500:NEXT K
260  NEXT J
270  NEXT I
```

Do you see how the values are switched? Again a temporary store has been used. If you run it, you will see it all happen on the screen.

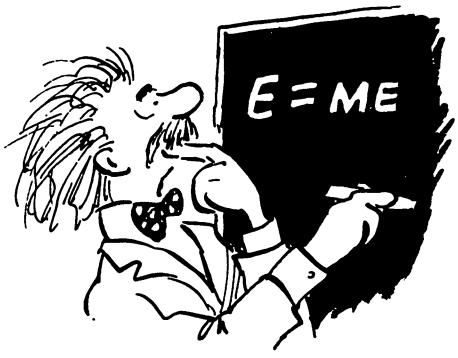
This next one is more efficient (slightly). We take each value LI(2), ... LI(N) in turn and insert it in the right place. This works because when we get to a particular place in the list, all the ones before it are in order because of the sorting we did before. To do this we need our searching expertise from Chapter 14 and our shuffle knowhow from this chapter:

```
140  REM  INSERTION SORT
150  FOR I=2 TO N
160  REM  INSERT LI(I)
170  FOR J=1 TO I-1
180  IF LI(I)>=LI(J) THEN 270
190  REM  HERE COMES
200  REM  THE SHUFFLE
210  LET LT=LI(I)
220  FOR K=I TO J+1 STEP -1
230  LET LI(K)=LI(K-1)
240  NEXT K
250  LET LI(J)=LT
260  GO TO 280
270  NEXT J
280  NEXT I
```

You may find this program a bit difficult; it certainly tests one's ability to think about subscripts. But that is why it is here. Work at it! And while you're at it, don't forget to try it. Put in lots of PRINT statements to help you follow it through. If you tried the bubble sort, then see if you can get a similar display of the sorting as it happens.

Sixteen

ANYONE FOR EINSTEIN?



1 You wouldn't want me to leave something out!

Perhaps this chapter caters for a minority interest, so skip it if you want to. But no mathematician can be without the mathematical functions of BASIC, and I wouldn't be earning my pitiful royalties if I didn't tell you how to use them. Also, as I promised earlier, I'm going to show you how to plot graphs of them. Let's jump straight in.

First the graphs. In Chapter 11 I did this:

```
10  FOR I=1 TO 20
20  FOR J=1 TO I
30  PRINT "*";
40  NEXT J
50  PRINT
60  NEXT I
```

and this:

```
10  FOR I=0 TO 21
20  PRINT TAB(I);"*"
30  NEXT I
```

These are both graphs. Look at them sideways. The first one is a solid or bar graph and the second is an ordinary graph, both of a straight line. Here is a bar graph showing some random numbers:

```
10  LET X=RND(-TI)
20  FOR I=1 TO 20
30  LET T=21*RND(1)
40  FOR J=1 TO T
50  PRINT "*";
60  NEXT J
70  PRINT
80  NEXT I
```

You should remember how these work. The semicolon causes printing to continue on the same line and so we can print T consecutive stars. T is a scaled output from the random number generator.

Now let's look at some functions.

2 EXP and LOG

These two mathematical functions make a pair and are based on e, or 2.71828182845, the magic number which is the base of natural logarithms. The EXP function provides e raised to a power, and LOG finds the logarithm to the base e. If you're good at that sort of thing, you will realise that

```
if Y = EXP(X)  then  X = LOG(Y)
```

These programs make graphs of these functions:

```
10  REM  PLOT EXP(X)
20  FOR X=0 TO 3.0 STEP 0.15
30  PRINT TAB(EXP(X));"*"
40  NEXT X
```

```
10  REM  PLOT LOG(Y)
20  FOR Y=1 TO EXP(19) STEP EXP(19)/50
30  PRINT TAB(LOG(Y));"*"
40  NEXT Y
```

The log function can easily be used to base 10 or any other base. You may know that

$$\log_a X = \log_e X \log_a e = \frac{\log_e X}{\log_e a}$$

so that if you could find the log of X to base 10 by a statement like

110 LET L=LOG(X)/LOG(10) or to base A by 110 LET L=LOG(X)/LOG(A)

3 SIN, COS, and TAN

These are the trigonometric functions of an angle, and are often used. In BASIC the angle is in radians and this may not always be convenient, although conversion is easy since π radians is the same as 180°. The value of π is available on the keyboard - just type it. Here is a program which will plot either SIN or COS for you:

```
10  REM  PLOT A TRIG THING
20  FOR X=- π TO π STEP π/9
30  REM  COULD BE COS OR SIN
40  PRINT TAB(10.5+10*SIN(X));"*"
50  NEXT X
```

TAN is not as pretty. Use

```
40  PRINT TAB(10.5+TAN(X));"*"
```

If you ever want to use TAN in a real program, remember that it doesn't like to evaluate tan(90°) - which is tan($\pi/2$) - or any similar angle. Do you know why?

4 ATN

The ATN or 'arc tangent' function is one that a lot of people have trouble understanding. You tell this function the value of a tangent, and the result is the angle that goes with it. For example, the tangent of 45° is 1. Therefore the statement

```
50  P4=ATN(1)
```

gives P4 the value of 45°; unfortunately in radians. However, 180° is π radians, and so 45° is $\pi/4$ radians.

You can convert any angle given in degrees, called D, to one in radians, called R, by

```
660  LET R=D* π/180
```

Similarly you could calculate D in degrees from R in radians by

```
700  LET D=R*180/π
```

Sometimes all of this can be very useful.

Seventeen

INVENT SOME FUNCTIONS

1 How to do it

We have been introduced to most of the built-in functions of BASIC already, although in Chapter 19 we will see that there are some more to come which deal with characters. Although VIC provides functions to cover most important things we might wish to do, it is often useful to make up our own functions for special requirements. In BASIC you can define a function as long as you can pack it all into one line. You do this with the DEF FN statement:

```
line number  DEF FNxx(variable) = expression
```

The name of your function is FNxx, where xx is a combination of letters and numbers very similar to the combinations allowed for variable names: one letter, two letters, or a letter plus a number. When you use it, you give it value to work on, just like you do with most of the other functions. Normally you would use this in working out the result, although you don't need to. When the function is used, the expression on the right hand side is worked out, substituting the value you gave for the variable in the DEF FN statement. This variable is called the 'argument' of the function. I'm sorry if this sounds a bit complicated. Some examples will help to sort you out.

2 The Buzz of the Bee

In the example of the bee and the neutron bomb a few chapters back, I had to work out from the POKE value just where on the screen I was. To be more precise, I had a number from 0 to 505 which represented the screen address, and I wanted to work out the row and column numbers from this. The row number is just

```
Row Number = INT(Screen Position/22)
```

but the column number was a bit more complicated:

Column Number = Screen Position - Row Number*22

Now this is actually our old remainder problem again. Let's define a function or two. First FNRO to find the row number which is the integer quotient of a screen position:

```
15  DEF FNRO(I)=INT(I/22)
```

and another to give the remainder when a screen position is divided by 22 which is the column number:

```
16  DEF FNCO(I)=I-INT(I/22)*22
```

Now if I want to know the row and column number of a screen position ISC, I can put

```
280  LET RO=FNRO(SC)
290  LET CO=FNCO(SC)
```

in the bee and bomb problem. But I can do even better than that. I want to know how far a screen position SC is away from the bomb at row 11, column 11. Define another function:

```
17  DEF FND(I)=(FNRO(I)-11)↑2+(FNCO(I)-11)↑2
```

Then in the bee program, to get the distance I could put

```
300  LET DS=FND(SC)
```

When I do this, the function FND refers to my functions FNRO and FNCO to give me the distance. But the plot thickens still further! The entire purpose was to work out how loud to make the buzz:

```
310  LET BZ=15/(1+DS/18)
```

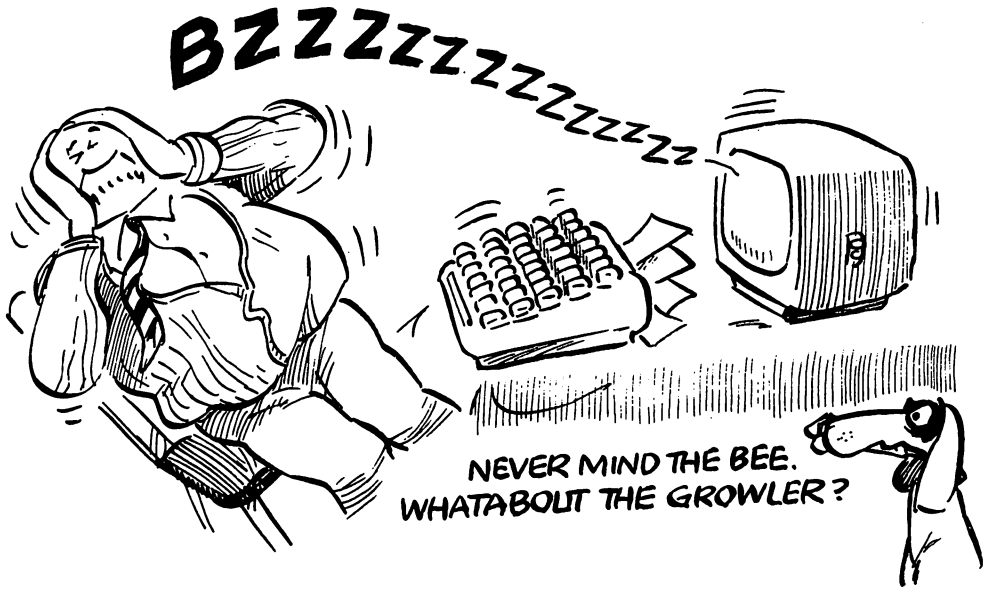
This too could be a function:

```
18  DEF FNB(I)=15/(1+FND(I)/18)
```

and I could then write:

```
310  LET BZ=FNB(SC)
```

The value of SC works its way all the way back to FNRO and FNCO where



it is used to give first of all the row and column numbers, then the distance from row 11, column 11, and finally the value of BZ used to set the volume control.

If you want to try this, take the second bomb program, add the function definitions as lines 15-18, remove 280, 290, and 300, and replace 310 with the latest version which uses FNB.

3 Some more examples

We can find the area of a circle of radius R. The area is πR^2. This is the function:

```
30  DEF FNCR(R)= π*R*R
```

Here is a little program to print the area of a number of circles:

```
10  REM  CIRCLE AREAS
20  DEF FNCR(R)= π*R*R
30  REM  NOW DO IT
40  FOR X=1 TO 2 STEP 0.1
50  PRINT "RADIUS ",X
60  PRINT "AREA ",FNCR(X)
70  NEXT X
```

You can see how the actual value X is substituted for R when the program uses the function FNCR.

In Chapter 7, I talked about rounding and truncation or 'chopping' of numbers to get integer results. Recall that the INT function of BASIC takes the next lowest integer. Here is a function to chop or truncate to an integer value:

```
60  DEF FNT(X)=SGN(X)*INT(ABS(X))
```

For comparison:

```
FNT(1.5) would be 1.0     INT(1.5) would be 1.0
FNT(-2.3) would be -2     INT(-2.3) would be -3
```

If you want to round instead to the nearest whole number, use this:

```
70  DEF FNRN(X)=INT(X+0.5)
```

4 Involving other variables

The functions used as examples so far used only the value of their argument. You may want to use other variable names. If this is so, there is no way of having substitutions made for the extra variables - this is a slight limitation of BASIC. You can only have one dummy argument. Apart from the one name you use as a function argument, any other variables you mention in the function definition refer to their actual values.

For example, consider another kind of rounding as first discussed in Chapter 7. The function FNRS rounds its argument after using a scale factor S, so that the result of the function is rounded to the nearest S:

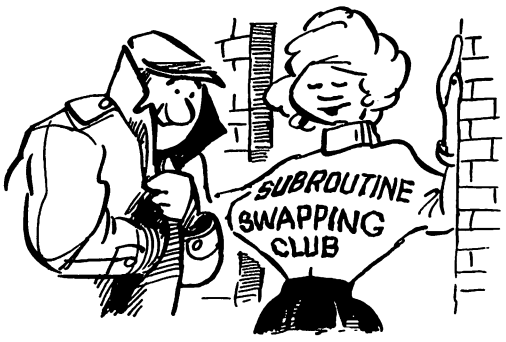
```
20  DEF FNRS(X)=INT(X/S+0.5)*S
```

Here, the value that X is given when the function is used is substituted in the right hand side when it is evaluated, but the value used for S is the actual value of the variable S. Here is a program to try this out:

```
10  REM  A ROUNDING THING
20  DEF FNRS(X)=INT(X/S+0.5)*S
30  LET S=0.1
40  PRINT "TYPE IN A NUMBER"
50  PRINT "TO GET IT ROUNDED"
60  PRINT "TO THE NEAREST 0.1"
70  INPUT N
80  PRINT "ROUNDED ",FNRS(N)
90  GO TO 40
100 END
```

Eighteen

SUBROUTINES



1 Pass the buck

A subroutine is a separate little program within a program. This can be very helpful. If you have an easily isolated little task inside a program that you wish to use over and over again to do the same thing but perhaps under slightly different conditions, you can make it a subroutine. This makes it easier to use in different programs (if you have a cassette tape or disk store), and if you have a printer you can trade subroutines in plain brown wrappers with other VIC owners who are similarly inclined.

All you do is write the bit of program out separately, with its own line numbers. It is usual to start a subroutine at a large line number, like 1000 or 2000. You end your subroutine with the RETURN statement

```
line number  RETURN
```

When you want to use your subroutine, you write

```
line number  GOSUB subroutine's line number
```

The program jumps to your subroutine, does what the subroutine says, and when it hits the RETURN statement it jumps back to the line after GOSUB.

2 Ring dem bells

Now we are going to make a subroutine to ring the bells. First of all, here is a subroutine simply to go "DONG" with the note BL:

```
3000  REM  GO DONG ON
3010  REM  NOTE BL
3020  REM  WORK OUT HARMONICS
3030  DEF FNFT(I)=255-(255-I)*2↑(5/3)
3040  POKE 36874,BL
3050  POKE 36875,FNFT(BL)
3060  POKE 36876,0
3070  POKE 36877,0
3080  FOR IB=15 TO 1 STEP -1
3090  POKE 36878,IB
3100  FOR KB=1 TO 240/IB
3110  NEXT KB
3120  NEXT IB
3130  POKE 36878,0
3140  POKE 36874,0
3150  POKE 36875,0
3160  RETURN
```

It would be useful to be able to play any tune we want on these bells. Here is a subroutine which calls this subroutine - why not? We put any tune in the array WC of length NC notes. This subroutine will play it:

```
2000  REM  PRIVATE CARILLON
2010  REM  PLAYS NC NOTES
2020  REM  FROM LIST WC
2030  REM  ON THE VIC BELFRY
2040  FOR LB=1 TO NC
2050  LET BL=WC(LB)
2060  GOSUB 3000
2070  NEXT LB
2080  RETURN
```

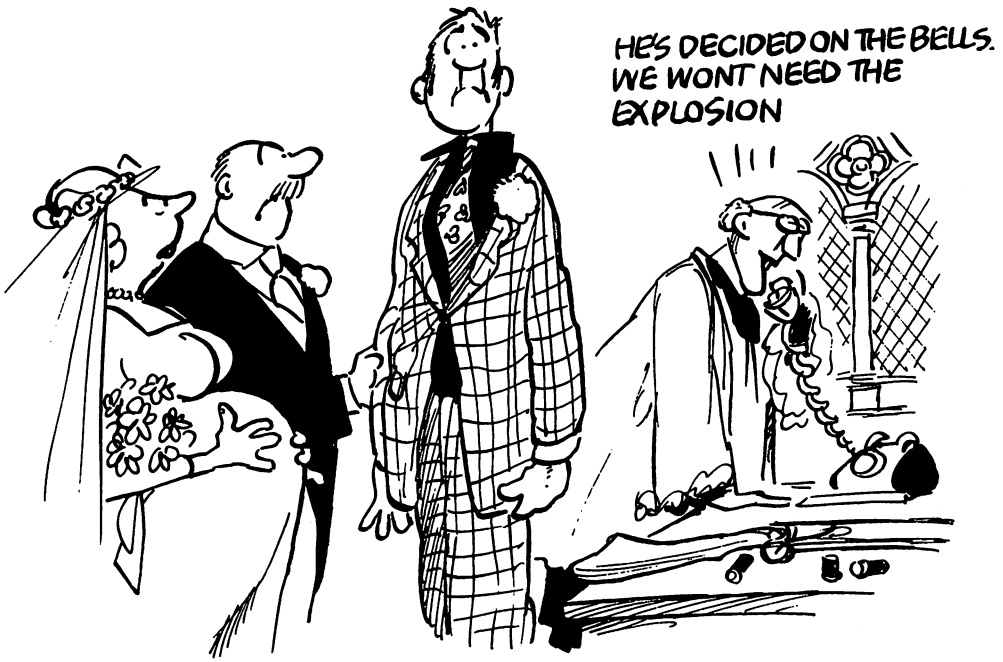
And here's a program to play the good old Westminster Chimes:

```
10  REM  BIG BEN STRIKES
20  DATA 239,237,235,228
30  DATA 228,237,239,235
40  DATA 239,235,237,228
50  DATA 228,237,239,235
60  DIM WC(16)
70  REM  READ DONGS
80  FOR I=1 TO 16
90  READ WC(I)
100 NEXT I
110 LET NC=16
120 GOSUB 2000
130 END
```

In the next chapter, we will use these subroutines to make a fantastic chiming clock!

3 The royal fireworks

We used a big, noisy, colourful explosion in two different programs. So let's make it into a subroutine. While we're at it, we will find that we can make the program a lot shorter because it contained more or less the same thing repeated four times. If you look back at the airplane crash in



Chapter 12, you will see that we do essentially the following, four times:

```
730  FOR L=1 TO N
740  LET IS=IS+1
750  POKE 7860+IS,160
760  POKE 38400+IS,CL
770  NEXT L
```

This new explosion is going to be of any size, and placed anywhere on the screen. In the above, IS was the screen position. As long as I check that I don't go off the screen, I can use almost the same statements as a subroutine:

```
3000  REM  SPIRAL THING
3010  FOR LX=1 TO MX
3020  LET IS=IS+DX
3030  IF IS>505 THEN 3070
3040  IF IS<0 THEN 3070
3050  POKE 38400+IS,CL
3060  POKE 7680+IS,42
3070  NEXT LX
3080  RETURN
```

In the grand explosion master subroutine, you use this to make an explosion starting at screen position SC, and SZ positions wide. Perhaps a square doesn't make a very pretty explosion, but you might like to improve on it.

```
2000  REM  GRAND BOOM
2010  REM  AT SCREEN ADDR
2020  REM  SC, SPIRALS
2030  REM  TO WIDTH SZ
2040  REM  FIRST THE NOISE
2050  POKE 36877,130
2060  POKE 36878,15
2070  POKE 7680+SC,42
2080  LET IS=SC
2090  REM AND NOW SPIRAL
2100  FOR NX=1 TO SZ STEP 2
2110  LET CL=7*RND(1)+1
2120  LET MX=NX
2130  LET DX=1
2140  GOSUB 3000
2150  LET DX=22
2160  GOSUB 3000
2170  LET MX=NX+1
2180  LET DX=-1
2190  GOSUB 3000
2200  LET DX=-22
2210  GOSUB 3000
2220  POKE 36878,PEEK(36878)-3
2230  NEXT NX
2240  POKE 36878,0
2250  RETURN
```

Now the royal fireworks. Perhaps you guessed it - we use RND again. One hundred blasts at random screen positions and random sizes.

```
  10  REM  FOURTH OF JULY
  20  REM  OR IS IT 5 NOV?
  30  POKE 36879,8
  40  PRINT "□"  SHIFT and CLR/HOME
  50  LET X=RND(-TI)
  60  FOR I=1 TO 100
  70  LET SC=505*RND(1)
  80  LET SZ=8*RND(1)
  90  IF SZ>0.9 THEN 120
 100  PRINT "□"  SHIFT and CLR/HOME
 110  FOR P=1 TO 200:NEXT P
 120  GOSUB 2000
 130  NEXT I
 140  POKE 36879,27
 150  PRINT"□"  SHIFT and CLR/HOME
 160  END
```

4 And let us not forget...

...to tell you that you can use the GOSUB in the ON statement:

line number ON expression GOSUB line number

but you can't use an ordinary IF...THEN to get to a subroutine, because VIC won't be able to get back. That is what is so special about GOSUB - VIC remembers where you came from so that the RETURN statement can get you back.

Nineteen

QUITE IN CHARACTER

1 String along with me

As nearly the last stage of our roundup of BASIC on the VIC 20, we find out here how letters and numbers and other symbols can be handled just as if they were variables. We have already met character string constants, without knowing it. Any series of symbols that you put into quotation marks is a character string constant:

```
10 PRINT "■"                    SHIFT and CLR/HOME
20 PRINT " \\\\ "
30 PRINT " ┌──┐ "
40 PRINT "┤    ● "
50 PRINT " └──┘ "
60 PRINT " //// "
```

and quite a large range of symbols is available. I have already described how you can get the graphics symbols on the keyfronts. When you switch VIC on, it is automatically placed in full graphics mode. If you are in this mode, you can type the right side graphics using the SHIFT key and the left side graphics using the Commodore trademark key. There is also another mode, which you could call 'text mode', in which the SHIFT key gives you upper and lower case letters, just like an ordinary typewriter. The left side graphics are still available through the Commodore key, but the right side graphics are not. There are two ways of switching modes:

(i) Push SHIFT and Commodore together

or

(ii) POKE 36869,240 to get graphics mode
POKE 36869,242 to get text mode

Character string variables are given these variable names with the symbol $ added on:

```
A$
N9$        are all names of string variables
CH$
```

A string variable can be subscripted, as for example

```
10  DIM TT$(60)
```

defines a string list or array with 60 entries, and

```
20  DIM X$(20,20)
```

defines a table of string variables. Tables or arrays with two or more subscripts are introduced in Chapter 20.

2 What can you do with a string?

Many of the statements of BASIC can manipulate character strings. We have seen string constants in PRINT statements. Here are the other things you can do.

(a) LET

The LET statement may contain a string variable on the left hand side and a string variable or constant on the right hand side. You cannot assign a character value to an ordinary integer variable, nor can you assign a numeric value to a string variable. Sorry about that. There are, however, some special functions to use with strings which are described a bit later. This is what you can do:

line number LET string variable = string variable or string constant

Of course this includes string variables that have subscripts.

(b) PRINT

In a PRINT statement you can use string constants or variables, or the special string functions.

```
10  LET TH$="CENSORED"
20  PRINT "THE THOUGHT FOR"
30  PRINT "TODAY IS"
40  PRINT "■■";TH$                CTRL RED and CTRL RVS ON
```

(c) INPUT

An INPUT statement can ask for character strings. To respond, you simply type the message you want. You can put it in quotation marks

if you want. You need to put it in quotes only if the message itself contains a comma or a colon. A bit later you will find out about the GET statement for reading one symbol from the keyboard.

(d) IF...THEN

Two character strings can be compared, as in

line number IF string compared to string THEN line number

as in

```
100  IF BX$(I)="STOP" THEN 66
```

This compares the Ith entry in the string array called BX$ with "STOP" and jumps to line 66 if they are the same.

When you compare strings, you have to know what order the symbols take. The alphabet is always taken in order so that

```
"ABC" is greater than "ABB"
                   or "AB"
"ABC" is equal to "ABC"
"ABC" is less than "ABD"
                or "ABCA"
                or "ABC "      (note the blank)
```

Because of this, you can sort lists into alphabetical order. If you want to know the order of the other symbols, look in the Appendix.

(e) DIM

As already mentioned, you can use string variables with subscripts, and therefore you can give them a size in the DIM statement. If you don't do this, then the subscripts are assumed to vary from 0 to 10.

(f) DATA and READ

You can put character string constants in your data list:

```
10  DATA "ONE","TWO","THREE"
20  FOR I=1 TO 3
30  READ ST$
40  NEXT I
50  PRINT ST$
```

This prints THREE. Be careful not to try to read a string constant into a numeric variable or vice versa. You won't like the results!

Actually you don't need the quotation marks unless the string constant contains commas or colons.

3 How to GET

How to GET what? How to get just one character from the keyboard. Particularly when you are devising video games, you need to know what key your player is hitting (sorry, I mean gently pressing) at the moment. Just write

line number GET string variable name

This puts a character that has been typed into the string variable. If there's nothing there, you get a null (not a blank). A null is like "". You can get any symbol, including the RETURN key, so your player doesn't hit RETURN every time he wants to dodge the Martian invaders. To show you how this works, we will use the keys 1 to 8 to play a tune on our bells.

In a data list, I put the note values of a G major scale, and then read them into an array. I then start looking at the keyboard. When I press the "1", what I get is the symbol for 1, not its value. To get its value, I borrow the VAL$ function from the next section, and pick out the note you are asking for. I use the DONG subroutine from Chapter 18:

```
 10  REM  DON'T FORGET THE
 20  REM  CANDELABRA
 30  DATA 215,219,223,225
 40  DATA 228,231,234,235
 50  FOR I=1 TO 8
 60  READ NO(I)
 70  NEXT I
 80  GET KY$
 90  IF KY$="" THEN 80
100  BL=NO(VAL(KY$))
110  GOSUB 3000
120  GO TO 80
```

Do you see why line number 90 is necessary? If you have not pressed a key, this makes the program wait until you do. If you run this, you will see that you can enter a whole lot of notes and then sit back while VIC plays the tune. The keyboard data is 'buffered' so that VIC can keep track of quite a few keys that have already been pressed.

4 Those useful character functions

You met VAL, a valuable function, in the last program. Here is a brief explanation of the functions that VIC provides for you to use with character strings:

ASC(X$) gives you the ASCII code of the first character in X$. This is a number.
CHR$(X) changes a number X which is in the ASCII code into the correct string character.

The ASCII code is not the same as the number used inside VIC to represent a character.

LEFT$(X$,X) This picks off the X leftmost characters of X$.
MID$(X$,S,X) This picks off X characters from the middle of X$, starting at character number S.
RIGHT$(X$,X) Surprise! This picks off the X rightmost characters of X$.
LEN(X$) This gives a number which is the number of characters in X$.

And finally, two functions which can convert a number into a character string and vice versa. Remember that when playing the VIC carillon, the symbol "1" is not the numerical value 1, i.e.

"1" is not equal to 1

However

VAL("1") is 1
STR$(1) is "1"

VAL(X$) This converts X$ into a number. X$ should be a string which makes up a number, for example "12.34". If VIC runs into a symbol that doesn't belong, it quits. This would give you a 0 result if the first character was wrong.
STR$(X) This converts a numeric value X into a character string which you could print.

5 And now BIG VIC - or is it VIC BEN?

I promised you a chiming clock. Actually we have most of the bits and pieces already. In Chapter 18 I gave subroutines to play a tune on the bells. All we need now is to have a clock.

I have said very little so far about the clock ticking away inside VIC. This is because you couldn't set it without knowing about character strings and we had more important things to learn about first.

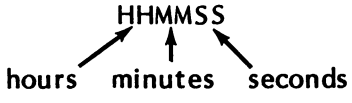
The variable name TI is special because it is reserved for the VIC clock. You will find that if you do this:

```
10  PRINT TI
20  GO TO 10
```

you can see it ticking away merrily. Now try:

```
10  PRINT TI$
20  GO TO 10
```

This is a character version of the clock. TI$ has 6 characters:



You can set the VIC clock by putting a character string into TI$. It is a 24 hour clock. Here is your clock program. You type in a time, and when you push RETURN, the VIC clock is set. It will then chime every quarter hour using the Westminster Chimes. We do this by looking at the time and chiming whenever the remainder of the time in hours, converted to minutes, is zero when we divide it by 15. Do you see how I chime the hours?

```
 10  REM  VIC BEN
 20  DATA 239,237,235,228
 30  DATA 228,237,239,235
 40  DATA 239,235,237,228
 50  DATA 228,237,239,235
 60  DIM WC(16)
 70  FOR I=1 TO 16
 80  READ WC(I)
 90  NEXT I
100  REM  SET CLOCK
110  PRINT "TYPE IN TIME"
120  PRINT "HHMMSS"
130  PRINT "AND PUSH RETURN"
140  PRINT "TO SET CLOCK"
150  INPUT TI$
160  PRINT "■"  SHIFT and CLR/HOME
170  REM  WATCH FOR QUARTERS
180  LET TS=VAL(RIGHT$(TI$,2))
190  LET TM=VAL(MID$(TI$,3,2))
200  LET Q=INT(TM*60+TS)
210  LET NC=INT(Q/900)
220  PRINT"■■■"TI$  CLR/HOME, CTRL RVS ON, CTRL RED
230  IF Q-NC*900<>0 THEN 180
240  REM  TIME TO CHIME
250  LET NC=NC*4
260  IF NC>0 THEN 280
270  NC=16
280  GOSUB 2000
290  IF NC<16 THEN 180
300  LET BL=215
310  LET NB=VAL(LEFT$(TI$,2))
320  IF NB>0 THEN 340
330  LET NB=24
340  FOR I= 1 TO NB
350  FOR J=1 TO 1000
360  NEXT J
370  GOSUB 3000
380  NEXT I
390  GO TO 180
```

To run this you need two bell ringing subroutines from Chapter 18, one beginning at line 2000 and the other at line 3000.

6 Sorting into alphabetical order

Here is a program to take a list of character strings and sort them into alphabetical order using an insertion sort. Each list item IN$(I) has another list item associated with it, PN(I) which is a number. I used a similar idea to make the Index for this book. In a character array I put the items I wanted indexed, and in a numeric array the page numbers. Then I sorted the character array into alphabetical order, dragging the page number alongside:

```
 10  REM  MAKE AN INDEX
 20  DIM IN$(200),PN(200)
 30  PRINT "OK AUTHOR,LET'S"
 40  PRINT "GET STARTED."
 50  PRINT "TYPE IN YOUR"
 60  PRINT "ITEMS AND PAGE"
 70  PRINT "NUMBERS. KEEP"
 80  PRINT "IT UP UNTIL"
 90  PRINT "YOU ARE FINISHED"
100  PRINT "THEN TYPE IN"
110  PRINT "QUIT,0"
120  PRINT "THE ONLY THING"
130  PRINT "YOU CAN'T DO"
140  PRINT "IS HAVE SOMETHING"
150  PRINT "CALLED QUIT"
160  PRINT "ON PAGE 0"
170  FOR I=1 TO 200
180  INPUT XN$,IP
190  IF XN$="QUIT" AND IP=0 THEN 240
200  LET IN$(I)=XN$
210  LET PN(I)=IP
220  NEXT I
230  PRINT "NO MORE ROOM"
240  PRINT "SORTING NOW"
250  GOSUB 2000
260  PRINT "HERE'S YOUR INDEX"
270  FOR J=1 TO I
280  PRINT IN$(J),PN(J)
290  NEXT J
300  END
```

Here's the subroutine:

```
2000  REM  INSERTION SORT
2010  FOR IS=2 TO IN
2020  REM  INSERT IN$(I)
2030  FOR JS=1 TO IS-1
2040  IF IN$(JS)>IN$(IS) THEN 2070
2050  NEXT JS
2060  REM  PUT IT IN JS
2070  CS$=IN$(IS)
2080  PS=PN(IS)
2090  FOR KS=IS TO JS+1 STEP -1
2100  IN$(KS)=IN$(KS-1)
2110  PN(KS)=PN(KS-1)
2120  NEXT KS
2130  IN$(JS)=CS$
2140  PN(JS)=PS
2150  NEXT IS
2160  RETURN
```

With all these lovely long useful programs, we really need that cassette, don't we?

Twenty

TURN THE TABLES

1 A whole new dimension

We know how to use lists - they are arrays with one subscript. I expect we also know that working with a subscript requires careful thought. Now we will find out that arrays with two or more subscripts are possible, although if you use a lot of subscripts you will run out of memory very quickly. You could think of a list, such as A, defined by

```
10  DIM TH(3)
```

as being laid out in a column:

```
TH(0)
TH(1)
TH(2)
TH(3)
```

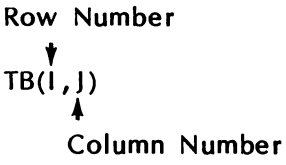
With two subscripts you have a table, like

```
10  DIM  TB(3,3)
```

which specifies

```
TB(0,0)  TB(0,1)  TB(0,2)  TB(0,3)
TB(1,0)  TB(1,1)  TB(1,2)  TB(1,3)
TB(2,0)  TB(2,1)  TB(2,2)  TB(2,3)
TB(3,0)  TB(3,1)  TB(3,2)  TB(3,3)
```

When you use a table, you write two subscripts separated by a comma; the first gives the row number and the second is the column number



If you use a variable name with two subscripts, VIC knows that you are talking about a table. But you have to be consistent. Once a variable is a list, you can't make it into a table and vice versa. As with a list, if you don't have a DIM statement for an array with two subscripts, BASIC will assume that each varies from 0 to 10, and it will therefore reserve 121 spaces in memory for it. As this is quite a lot of space, you should always remember to give a DIM statement. The normal array of three subscripts would use 1331 spaces, and so as the number of subscripts increases it becomes more and more important to have a DIM statement. The DIM statement can be used to specify several arrays:

line number DIM variable sizes with commas , variable sizes with commas , ...

The variables can be ordinary, integer, or string variables.

2 Do your budget

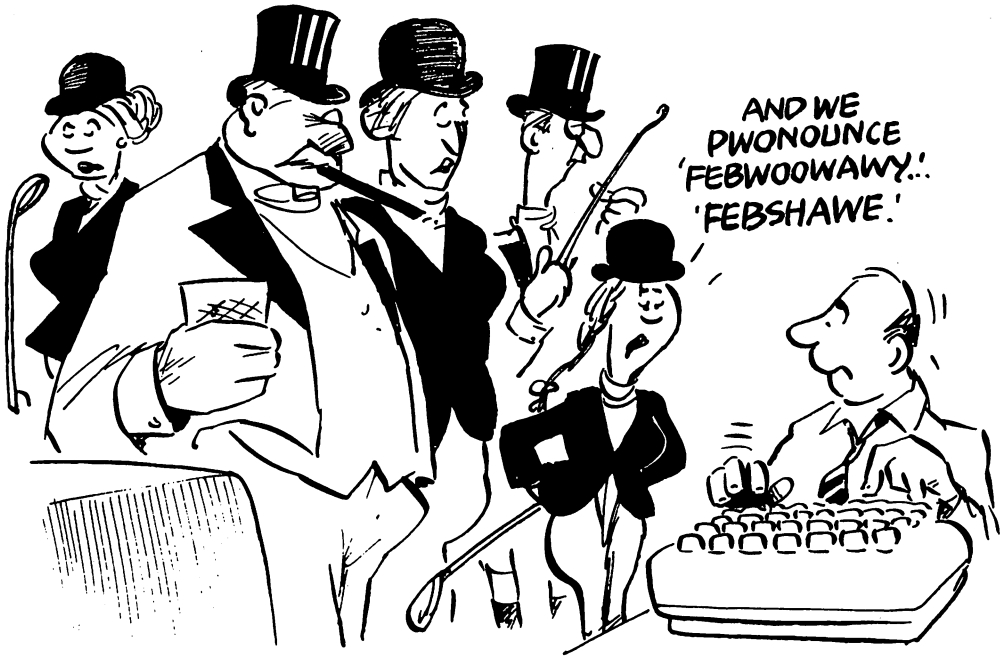
A lot of things in real life turn out to be best represented by tables. Here is a family budget:

The Fairwetherstonehaugh Family Budget

	The Stately Home	The Rolls	Banqueting	Couturier	Opera and Ballet
January	3000	540	900	1500	700
February	3000	560	900	1200	100
March	2000	580	1200	1650	0
April	1500	240	300	1300	300
May	1000	600	400	900	50
June	500	650	400	200	2300

(Glyndebourne and Bayreuth!)

The family name, by the way, is pronounced as if it were 'Fanshawe'. No kidding!



You could set this out in a DATA statement, and read it into a table:

```
10  REM  LIVING HIGH
15  REM  THE HOUSING
20  DATA 3000,3000,2000,1500,1000,500
25  REM  THE DRIVING
30  DATA 540,560,580,240,600,650
35  REM  THE EATING
40  DATA 900,900,1200,300,400,400
45  REM  THE CLOTHING
50  DATA 1500,1200,1650,1300,900,200
55  REM  THE ENTERTAINING
60  DATA 700,100,0,300,50,2300
```

```
70  DIM FB(6,5)
80  FOR J=1 TO 5
90  FOR I=1 TO 6
100 READ FB(I,J)
110 NEXT I
120 NEXT J
```

You can also put in captions for the rows and columns:

```
130 DATA JAN,FEB,MAR,APR,MAY,JUN
140 DATA HOUSE,CAR,FOOD,RAGS,GIGGLES
150 DIM CR$(6),CC$(5)
160 FOR I=1 TO 6
170 READ CR$(I)
```

```
180 NEXT I
190 FOR J=1 TO 5
200 READ CC$(J)
210 NEXT J
```

I have used I for column numbers and J for row numbers for no particular reason except that people usually do. After all of this the budget is in FB, and CR$ has the row captions and CC$ the column captions.

Now let's work on this table. First find out what the half-yearly budget is for each item:

```
220  FOR J=1 TO 5
230  LET SM=0
240  FOR I=1 TO 6
250  LET SM=SM+FB(I,J)
260  NEXT I
270  PRINT "YOUR " CC$(J)
280  PRINT "BUDGET FOR 6"
290  PRINT "MONTHS IS " SM
295  PRINT
300  NEXT J
305  INPUT "PUSH RETURN TO GO ON";A$
```

and the monthly totals:

```
310  FOR I=1 TO 6
320  LET SM=0
330  FOR J=1 TO 5
340  LET SM=SM+FB(I,J)
350  NEXT J
360  PRINT "IN " CR$(I)
370  PRINT "YOU WILL SPEND"
380  PRINT SM
390  NEXT I
395  INPUT "PUSH RETURN TO GO ON";A$
```

and compute the total cash flow for the half year:

```
400  LET SM=0
410  FOR I=1 TO 6
420  FOR J=1 TO 5
430  LET SM=SM+FB(I,J)
440  NEXT J
450  NEXT I
460  PRINT "THIS HALF YOU"
470  PRINT "NEED"SM
480  PRINT "BETTER CALL"
490  PRINT "CHRISTOBYS AND"
500  PRINT "SELL THE TITIAN"
```

Do you see how the nested loops have been used with the subscripts to work out each sum?

Appendix

A SUMMARY OF BASIC ON THE VIC 20

1 BASIC programs

A program has numbered lines, or statements, like

line number statement

such as

```
10  PRINT "HOWDY"
20  END
```

When you use the command RUN, the BASIC program in the VIC is obeyed in the order of line numbers unless the program itself decides otherwise. VIC does not require an END statement, but many computers do. A program stops running when it runs out of lines or an END statement is encountered. It can be restarted by a CONT command.

2 Commands

Anything you type without a line number is a command (except when a program is running). VIC has the following:

NEW Throw everything away and start a new program.

LIST Print the current program on the screen.
LIST entire program
LIST line number- : from there to end, e.g. LIST 20-
LIST line number : only one line, e.g. LIST 50
LIST -line number : from top to there, e.g. LIST -100
LIST line number-line number : between line numbers, e.g. LIST 10-20

RUN Obey the program from the beginning.

RUN line number : start at line number, e.g. RUN 50

CONT Restart a program that has been interrupted or halted by a STOP statement, END statement, or STOP key. If you change nothing in the program, it can carry on.

LOAD Load from cassette. See VIC manual.

SAVE Save on cassette or disk. See VIC manual.

VERIFY Check program against cassette or disk. See VIC manual.

Most of the statements of BASIC can be executed like a command; for example, if you type without a line number

POKE or PRINT PEEK( )

this will be obeyed at once and not become part of your program. These are called 'direct' statements. The statements that cannot be used this way are DEF FN, GET or GET#, and INPUT or INPUT#.

3 Creating BASIC programs

Everything you type that begins with a number is a line of BASIC, except when a program is running.

(a) Entering

Simply type in the program; each line begins with a line number and ends by pressing RETURN. You can put several statements on one line with colons between, as in

```
30 FOR I=1 TO 10 : NEXT I
```

(b) Correcting

Replace a line by typing it in full, or position the cursor over the error and type over it. Press RETURN to make the replacement permanent - changing it on the screen is not enough.

(c) Inserting
A new line is inserted by using a suitable line number.
(d) Deleting
A line is removed by typing the line number and pushing RETURN.

4 Numbers in BASIC

Numbers which are constants in BASIC can take three forms:

(a) Integers - a number without a decimal point, such as 56.
(b) A number with a decimal point, such as 2.71828.
(c) Either of the above with the letter E and an exponent, e.g. 3.61E23 meaning 3.61×10^{23}.

5 Variables in BASIC

(a) Ordinary variables in BASIC represent numerical values. They can be given names which are single letters: A, B, C, ... Z

or a letter plus a digit: A0, A1, ... Z9

or two letters: BL, ZC, ...

The variables TI and TI$ refer to the VIC clock as described in Chapter 20. The variable ST is an indicator used with tape or disk. You cannot use these as variable names: FN, IF, OR, ON, TO.

(b) An integer variable represents an integer value in the range -32768 to 32767 and could save space in practice. These have the ordinary names with a % sign:

Q%, XP%, C3%, ...

An integer variable cannot be used to count in a FOR...NEXT loop.

(c) String variables represent values which are character strings and have ordinary variable names with a $ sign: A$, CH$, P7$, ...

(d) Scalars
A variable is 'scalar' if it is written without a subscript. It represents a single value, e.g. X7.

(e) Arrays
A variable represents an array if it is written with subscripts, such as T(1,2,3), A$(7), WC(I). There is no limit on the number of subscripts. A particular array must always have the same number.

(f) Subscripts
A subscript may be any expression; however, the value it gives must be between 0 and the maximum for that subscript. Non-integer subscripts are truncated to integers.

(g) Array sizes
The maximum size of a subscript for an array is 10 unless a size is set up by a DIM statement.

6 Character strings

(a) A sequence of symbols in quotation marks is a character string constant, e.g. "WHOOPEE". They can be used in PRINT statements, DATA, IF...THEN, and on the right hand side of LET statements. In the DATA statement the quotation marks can be omitted unless the string contains a comma or a colon or has leading or trailing blanks.
(b) String variables have names ending with $, such as A$, CH$, P7$. They can be used in the statements INPUT, GET, PRINT, READ, IF... THEN, LET, DIM (and therefore they can be subscripted).
(c) In comparing strings in the IF...THEN statement, the order is that of the POKE codes tabulated in Section 13 of this Appendix. Strings at the beginning of the alphabet are less than those near the end.
"QR" is less than "QN"
"QR" is less than "QRA"
"QR" is less than "QR "

7 Arithmetic expressions

Operations use the hierarchy:

( )	expressions in brackets	high priority
↑	exponentiation	
* /	multiplication and division	
+ -	addition and subtraction	low priority

Operations of equal priority are done left to right.

Expressions can be written using ordinary or integer variables or constants, e.g. (A+3)/D(1%). A single variable or constant is itself an expression. You cannot do arithmetic with character string variables.

The operators *, ↑ and / normally have to appear between values but the operators + and - can appear in front of any value.

A*/B is illegal A+-B is legal
A*-B is legal A +B is legal

8 Relational expressions

(a) arithmetic expression — relational operator — arithmetic expression e.g. PQ < 3.

character string	relational operator	character string	e.g. "NOW" > L$(K).

The result of a relational expression is TRUE or FALSE. Relational expressions are used only in the IF...THEN statement.

(b) The relational operators are

=	equal to
>	greater than
<	less than
>= or =>	greater than or equal to
<= or =<	less than or equal to
<> or ><	not equal to

(c) The operators NOT, AND, OR can be used in relational expressions. These have hierarchy

AND	highest
(inclusive) OR	
NOT	lowest

e.g. (P=Q ↑ 2) OR (A$>="BB").

9 Library functions

VIC BASIC provides the following functions.

SIN(X)	The sine of X where X is an angle in radians (Chapter 16).
COS(X)	The cosine of X where X is an angle in radians (Chapter 16).
TAN(X)	Tangent of X where X is an angle in radians (Chapter 16).
ATN(X)	Angle in range $-\pi/2$ to $\pi/2$ with tangent X (Chapter 16).
EXP(X)	The value e^X (Chapter 16).
LOG(X)	The natural logarithm $\log_e X$ (Chapter 16).
SQR(X)	The square root of X where X must be positive (Chapter 7).
SGN(X)	Sign of X: 1=positive, 0=zero, -1=negative (Chapter 7).
PEEK(X)	The value contained in memory address X (Chapter 9).
RND(X)	A random number. Start off with RND(-TI), thereafter use RND(1) (Chapter 13).
ASC(X$)	The ASCII code of the first character in X$ (Chapter 19).
CHR$(X)	A string character whose ASCII code is X (Chapter 19).
LEFT$(X$,X)	String using the leftmost X characters of X$ (Chapter 19).
MID$(X$,S,X)	A string of X characters starting from the Sth character in X$ (Chapter 19).
POS(X)	The present column number of the screen cursor. X is not referred to. It is usual to put POS(0).
RIGHT$(X$,X)	String using the rightmost X characters of X$(Chapter 19).
LEN(X$)	The number of characters in X$ (Chapter 19).
VAL(X$)	Convert X$ to number, e.g. "12.34" ➤12.34 (Chapter 19).
STR$(X)	Convert X to a string, e.g. 12.34 ➤ "12.34" (Chapter 19).

Not described in this book:

USR(X) Jump to machine language subroutine.
FRE(X) The number of bytes of free memory.

In the above, X is any expression, which could include references to other functions. X$ is a character string. The X or X$ are called the argument or parameter of the function.

The following are used in PRINT statements and are described in Section 11 of this Appendix: TAB(X), SPC(X).

10 The statements of VIC BASIC

These are given here in alphabetical order, and include some not described in the main text. Most of the statements of BASIC can be executed as 'direct statements', like a command, by typing them without a line number. Those that cannot be treated that way are DEF FN, GET or GET#, and INPUT or INPUT#.

line number CLR
Sets all variables to zero but not your program. The command RUN does this anyway. Not described in the text.

line number CMD value
Send output to file number value instead of to the screen. To get it back to the screen, CLOSE the file. Not described in text.

line number DATA constant, constant, ... (Chapter 9)
The constants are stored in the computer in order. Successive DATA statements add to the DATA list. Information from the list can be assigned to variables by the READ statement.

If the constants are character strings, they can be given with or without quotation marks. The quotation marks are required if the string contains a comma or colon or has leading or trailing blanks.

line number DEF FNxx(variable)=expression (Chapter 17)
Used to define your own functions. See Section 12 of this Appendix.

line number DIM name(sizes), name(sizes), ... (Chapter 14)
The DIM statement specifies the maximum size of arrays. The sizes must be integer numbers. If an array is not mentioned in a DIM statement, then each subscript can vary from 0 to 10. An array can be mentioned in at most one DIM statement, and must always have the same number of subscripts.

line number END
When an END statement is reached, the execution of a program terminates, as it does when it runs out of lines. It can be restarted with the CONT command.

line number FOR variable=expression a TO expression b STEP expression c (Chapter 8)

This statement begins a FOR...NEXT loop, which is repeated with the variable starting at the value given by expression a and stepping by expression c until it reaches expression b. The STEP is optional and if it is not given the step used is 1. The variable can be adjusted during the loop but the initial, final, and step values are set when the loop first begins and cannot be altered from inside the loop. A loop ends on a NEXT statement which must be present. FOR...NEXT loops using different variables may be nested.

line number GET variable (Chapter 19)

The GET statement returns in the variable values from single keys pressed on the keyboard. Normally the variable would be a string variable; if it were an ordinary or integer variable then the key would have to be a number. If a series of keys has been pressed, a series of GET statements returns them in order. If no key has been pressed the value "", i.e. nothing at all, is returned.

line number GET# value, variable

Obtains one character from the unit number given by the value, which must have been OPENed. The variable is normally a character string variable as with GET above. Not described in the text.

line number GOSUB line number (Chapter 18)

Used to call a subroutine. See Section 12 of this Appendix.

line number a GO TO line number b (Chapter 5)

This causes an immediate jump to line number b.

line number a IF relational expression THEN line number b (Chapter 6)

When the IF statement occurs, the relational expression is evaluated and if it is TRUE the program jumps to line number b. If FALSE, the program carries on with the next line after line number a. Relational expressions are described in Section 8 of this Appendix.

line number INPUT variable, variable, ... (Chapter 4)

This causes the computer to request information from the keyboard. It will prompt with a "?" and wait for the information to be entered. The user should enter the correct number of values with commas between. If too many items are given, VIC will ignore the extra and tell you so. If too few are given it will keep asking for more. You can write:

line number INPUT "message"; variable, variable, ...

to prompt yourself with the message. You have to use a semicolon in this case.

line number INPUT# value, variable, variable, ...

The same as INPUT but takes input from the file given by value, which must have been OPENed. This is not described in the text.

line number LET variable=expression (Chapter 4)

The expression on the right hand side is worked out, and its value replaces the value of the variable. The word LET is not required.

line number NEXT variable (Chapter 8)
The NEXT statement indicates the end of a FOR...NEXT loop. You can leave out the variable name, and you can give several variable names separated by commas which are taken from left to right. This does not enable you to violate the proper nesting of loops.

line number a ON expression GO TO line number b, line number c, ... (Chapter 6)

or

line number a ON expression GOSUB line number b, line number c, ... (Chapter 19)
This allows a multiple choice of destinations. The expression is evaluated and truncated to an integer. If the result is 1, the branch is to line number b, if 2, to line number c, and so on. If the result is negative, zero, or too great for the number of destinations, the program continues with the next line after line number a. The statement is in two forms; one is a choice of destinations for an ordinary jump and the other is a choice of subroutines.

line number OPEN value 1, value 2, value 3, string
This statement opens the unit whose number for CLOSE, CMND, GET#, INPUT# and PRINT# statement is value 1. Value 2 is a standard device number: 0 = screen; 1 = cassette; 4 = printer; 8 = disk. Value 3 is a secondary address which depends on the device. The string also could mean various things to the device being used. This facility is not described in this text.

line number POKE address, value (Chapter 9)
The value, from 0 to 255, is put in memory at the given address.

line number PRINT value punctuation value ... (Chapters 1, 11)
This produces printed output. See Section 11 of this Appendix.

line number PRINT# unit, value punctuation value ...
Produces output on the unit whose number is given, which must have been OPENed. It behaves like the PRINT statement described in Section 11 of this Appendix. Not described in this text.

line number READ variable, variable, ... (Chapter 9)
This assigns values to the variables in order from the list of values established by one or more DATA statements. Successive READ statements continue through the list. You have to be careful not to read numbers where character strings are expected, and vice versa.

line number REM any old comment (Chapter 4)
This does nothing in a running BASIC program. It is provided to allow explanations to be inserted in programs.

line number RESTORE (Chapter 9)
Returns later READ statements to the beginning of the DATA list.

line number RETURN (Chapter 19)
Used in subroutines. See Section 12 of this Appendix.

line number STOP
Your program will halt and tell you what line number it has reached. You can carry on with the CONT command.

line number SYS address
Jumps to the memory address and executes a machine language program there. Not described in the main body of this text.
line number WAIT address, number 1, number 2, number 3
The program halts until the value in the given address, exclusive ORed with number 3 if it is given and ANDed with number 2, is not zero. The use of this is not described in the body of this text.

11 Printing

All output is done by the PRINT statement:

line number PRINT value punctuation value

The values can be

(a) expressions giving a numerical result
(b) a string variable which is printed as a message
(c) a character string in quotation marks
(d) the TAB or SPC functions.

The punctuation can be

(a) a comma to jump to the next field on the screen. A row on the screen is divided into two columns, each 11 spaces wide.
(b) a semicolon to squeeze the values together
(c) omitted before or after a character string in quotation marks, in which case the effect is like a semicolon. If you end a PRINT list with a comma or semicolon, the next PRINT statement continues on the same line. Otherwise all PRINT statements start a new line.

The TAB function
TAB(X) causes the output line to jump to column number X. It cannot, however, move backwards, and will use the next available space if the print position is already beyond column X. This is used only in a PRINT statement.

The SPC function
SPC(X) prints X spaces, i.e. moves forward by X spaces of output, without blanking out everything it moves past. This is used only in a PRINT statement.

The POS function
POS(0) returns the present cursor column number in the range 0 to 21. The POS function can be used anywhere that an ordinary function can be used. The argument of POS is not referred to.

12 Functions and subroutines

(a) Functions

You can define a one line function

line number DEF FNxx(variable)=expression

where xx is any ordinary variable name allowed by BASIC, e.g. FNA, FNRO, FNZ7.

When a program uses a function, the expression on the right hand side is evaluated using the given value of the function variable, but using the true values of any other variables. A given function name can only be defined once, and must be defined before it is used. Functions may use other functions if they were defined earlier.

(b) Subroutines

Subroutines are called by the GOSUB statement:

line number a GOSUB line number b

or by

line number a ON expression GOSUB line number b, ...

The program jumps to the subroutine at line number b (or other in the ON...GOSUB version) and executes it until a RETURN is given. It then resumes execution at the line after line number a.

Subroutines can call other subroutines, but should not set up an endless loop of calls.

Subroutines are ended by the RETURN statement:

line number RETURN

The running program returns to the line following the latest GOSUB.

13 The VIC 20 screen

The screen of the VIC 20 contains 22 columns numbered 0-21 and 23 rows numbered 0-22. In the memory of the VIC there is a table which always contains the 506 symbols currently on the screen and another containing the colour of each.

Symbol table:

The POKE address 7680 is row 0, column 0. The address of any screen position is

7680 + Column Number + 22*(Row Number)
0 to 21 0 to 22

The codes for the symbols are given in the next section.

Colour table:

The POKE address 38400 is row 0, column 0. The address of any screen position is:

38400 + Column Number + 22*(Row Number)
0 to 21 0 to 22

14 Character and symbol codes

Here is a list of the POKE values of all the symbols in the VIC 20. To get the reverse of a symbol, add 128 to its POKE value. For example, POKE 160 is a solid blob which is the reverse of POKE 32, a space.

POKE Value	Graphic Symbol	Character Symbol	POKE Value	Graphic Symbol	Character Symbol
0	@	@	21	U	u
1	A	a	22	V	v
2	B	b	23	W	w
3	C	c	24	X	x
4	D	d	25	Y	y
5	E	e	26	Z	z
6	F	f	27	[	[
7	G	g	28	£	£
8	H	h	29	]	]
9	I	i	30	↑	↑
10	J	j	31	←	←
11	K	k	32	space	space
12	L	l	33	!	!
13	M	m	34	"	"
14	N	n	35	#	#
15	O	o	36	$	$
16	P	p	37	%	%
17	Q	q	38	&	&
18	R	r	39	'	'
19	S	s	40	(	(
20	T	t	41	)	)

POKE Value	Graphic Symbol	Character Symbol	POKE Value	Graphic Symbol	Character Symbol
42	*	*	85		U
43	+	+	86		V
44	,	,	87		W
45	—	—	88		X
46	.	.	89		Y
47	/	/	90		Z
48	0	0	91		
49	1	1	92		
50	2	2	93		
51	3	3	94	π	
52	4	4	95		
53	5	5	96	space	
54	6	6	97		
55	7	7	98		
56	8	8	99		
57	9	9	100		
58	:	:	101		
59	;	;	102		
60	<	<	103		
61	=	-	104		
62	>	>	105		
63	?	?	106		
64			107		
65		A	108		
66		B	109		
67		C	110		
68		D	111		
69		E	112		
70		F	113		
71		G	114		
72		H	115		
73		I	116		
74		J	117		
75		K	118		
76		L	119		
77		M	120		
78		N	121		
79		O	122		
80		P	123		
81		Q	124		
82		R	125		
83		S	126		
84		T	127		

The graphic mode is used when VIC is first turned on. To switch to text mode, press SHIFT and the Commodore trademark key. Alternatively, to get text mode, use POKE 36869,242. To get graphics mode use POKE 36869,240. BASIC programs can be typed in either mode.

The right side graphics symbols from the keyfronts are always available by pressing the Commodore trademark key and the derived symbol key.

The left side graphics are only available in graphics mode, by pressing SHIFT and the key. In text mode, the keyboard behaves like a typewriter keyboard; SHIFT and a key gives upper case while the same key alone gives the lower case symbol.

15 Colours

The colours available to VIC are numbered

0	BLACK	8	ORANGE
1	WHITE	9	LIGHT ORANGE
2	RED	10	PINK
3	CYAN	11	LIGHT CYAN
4	PURPLE	12	LIGHT PURPLE
5	GREEN	13	LIGHT GREEN
6	BLUE	14	LIGHT BLUE
7	YUCK!	15	LIGHT YUCK!

Colours 0-15 can be used to set the border and colours 0-7 can be used for the screen background. Use

```
POKE 36879, 8 + Border Colour + 16*(Screen Colour)
```

The usual combination is POKE 36879,27.

The cursor colour is set by typing CTRL and the colour key. It can be set in a program as well, as in

```
PRINT "■"          CTRL and RED
```

Similarly CTRL and RVS ON causes symbols to be reversed. In a PRINT statement this lasts for one line of output, which can be extended over several PRINT statements by the semicolon.

16 Sound and music

VIC contains four tone generators:

```
POKE 36874,N    a low tone
POKE 36875,N    an octave higher
POKE 36876,N    another octave higher
POKE 36877,N    a growler
```

N can have a value from 0 to 255. Any value between 0 and 127 produces silence. 128 is the lowest note of each generator, and 254 the highest. Don't use 255 - I think the designer made a mistake.

The volume is set by:

```
POKE 36878,N
```

where N can be from 0 to 15 to give a volume from off to loud.

The approximate musical notes are:

Note	Bottom Octave	Middle Octave	High Octave	Top Octave
C	135	195	225	240
C#	143	199	227	241
D	147	201	228	.
D#	151	203	229	.
E	159	207	231	.
F	163	209	232	not
F#	167	212	233	useful
G	175	215	235	.
G#	179	217	236	.
A	183	219	237	.
A#	187	221	238	.
B	191	223	239	.

INDEX

START WITH BASIC ON THE COMMODORE VIC 20

Don Monro

Illustrated by Bill Tidy

JOIN THE COMPUTER REVOLUTION AND BECOME A COMPUTER EXPERT!

Let Don Monro's latest book, START WITH BASIC FOR THE COMMODORE VIC 20, teach you to program the Commodore VIC 20. Using BASIC, this dynamic new book shows how easy it is to do your own programming utilizing all the graphic functions of the VIC 20. The helpful exercises and step-by-step instructions put the full power of this amazing machine at your fingertips.

Table of Contents:

Reston Publishing Company
A Prentice-Hall Company
Reston, Virginia

0-8359-7070-1