# Back Door Into BASIC

## Phil Cohen

In this part of the series, Phil Cohen expands the arithmetic functions he covered last month to include arrays, and introduces strings and string manipulation functions — which form the basis of word processing.

AN ARRAY is a method of storing a large number of values in different places in memory in such a way that they are easy to handle.

For example, say you own twenty chooks (that's 'chickens', for the English-speaking audience) and want to keep track of how many eggs they lay. I realise this is hardly the sort of task you would buy a computer for — but it's only an example (eggsample? — Ed.)

Every time one of your chooks lays an egg, you go to your computer and add 1 to a number stored in it somewhere. Simple.

For chook number 1, you would input at the start of the week: 'C1 = 0'. This would cause the computer to set aside an area in memory for variable C1, and to put the value 0 into it. Then you would input 'C2 = 0', which would do the same for chook number 2. And so on up to C9.

Then you come to a problem. You can't put in C10 — BASIC only allows one digit after the letter. The next variable name you use has to be something like H1 (for 'hen' number 1). You then have H1 to H9 — but you're still stuck. So you use R1 and R2 — roosters 1 and 2. I know they're not roosters, but it's as close as you can get.

Now you're set. Every time one of the chooks lays an egg, you input something like 'C5 = C5 + 1', which, if you think about it for a minute, will add 1 to the present value of C5. At the end of the week you can 'PRINT C5' to find out how many it's laid.

This is all a bit messy. After all, you have to remember that chook 11 is called H2 in the computer, and then there are chooks 19 and 20 which are getting neurotic because you keep calling them roosters 1 and 2 . . .

This is where arrays come in. If you input 'DIM C(20)', the computer will set aside space for 20 variables and call them C(1), C(2), . . . C(20). DIM is a BASIC word which is short for 'dimension'.

Now when chook number 15 lays an egg, you input 'C(15) = C(15) + 1'. At the end of the week, you can 'PRINT C(15)' to get the total.

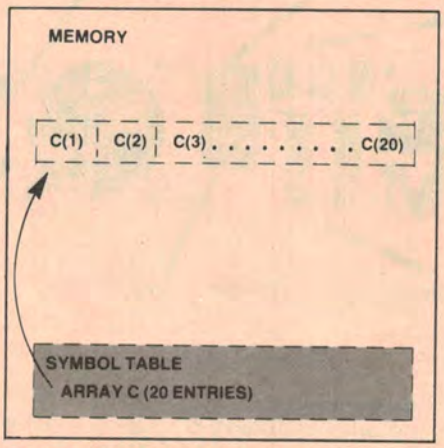



*Figure 1.* **An array could be stored in the computer in this way. The symbol table holds information on how big the array is, what its name is and where it begins.**

Figure 1 shows what this looks like in memory. All the computer needs to remember is the name of the array and the position in memory of the first item. When asked to find C(15), the computer will figure out the distance in memory of C(15) from C(1), then add this to the position of C(1). This will give it the position of C(15). The computer will, of course, know how much space each of the items takes up.

By the way, an item of an array takes up less space than a normal variable — this is because of reduced symbol table usage, amongst other things.

In fact, you can save even more space by using C% as an array — an array of integer variables. The dimension statement would then be 'DIM C%(20)'.

Any valid BASIC variable name can be used (in most versions of BASIC) as a variable name. D3(120) is quite acceptable. Even T8%(9) is OK.

## Multi-dimension arrays

What if we complicate the problem? Say we want to find out how many eggs each chook has laid — and that we want to know how many of them were brown, how many white and how many green (well, like I said, it's only an example . . . ).

We could dimension three arrays, one B(20), one W(20) and one (G(20). The way to do this would either be to input 'DIM B(20)', then 'DIM W(20)', then 'DIM G(20)', or, more simply, to input 'DIM B(20), W(20), G(20)' (which means the same in BASIC).

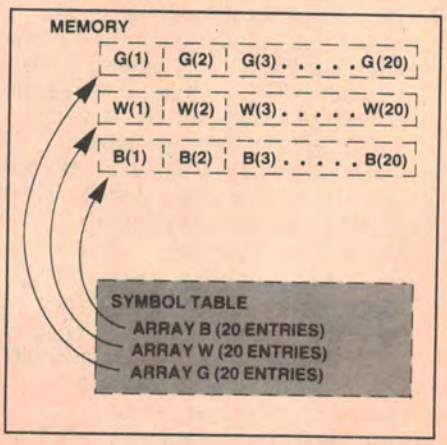



*Figure 2.* **Three arrays — but there's an easier way . . .**

Either way will set aside three lots of 20 variables (see Figure 2). This works in exactly the same way as with C(20) in Figure 1.

Every time chook number 15, say, laid a brown egg, you would input 'B(15) = B(15) + 1', and every time chook 3 laid a green egg you would input 'G(3) = G(3) + 1'.

At the end of the week, you could find out how many green eggs chook 11 had laid by inputting 'PRINT G(11)'.

This is all very well, but say that instead of grading the eggs by colour, you decided to grade them by size, and that you had a machine that told you which of ten sizes the egg was. You could dimension ten arrays called A(20), B(20) . . . J(20), but again this is getting messy. What you need is a two-dimensional array.

This means that you are actually creating an array which instead of taking up a row of memory, takes up a rectangular area. The dimension statement you would use would be something like 'DIM E(20,10)'. This would set aside 200 spaces in memory (ten for each chook). Figure 3 shows what it would look like.

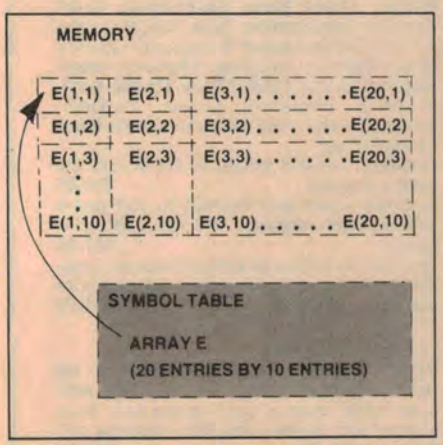



*Figure 3.* **An easier way to arrange data of the type shown in Figure 2.**

The items of the array are laid out in rows and columns. E(5,7), for example, is in row 5, column 7 — or if you like, it's the number of size 7 eggs laid by chook 5.

Whenever chook 15 laid a size 3 egg, you would input 'E(15,3) = E(15,3) + 1'. At the end of the week, to find out how many size ten eggs chook 2 had laid, you would 'PRINT E(2,10)'.

You have to be careful, though, not to confuse the egg size and the chook number — it's up to you to remember that E(3,5) is the number of size 5 eggs by chook 3, and *not* the number of size 3 eggs by chook 5!

Arrays are very useful in programming, because they allow the user to do the same thing to a large number of items automatically. Say, for example, that you wanted to print out the number of size ten eggs laid by all of your chooks. You could input 'PRINT E(1,10)', then 'PRINT E(2,10)', then . . . This would be rather tedious.

Instead, you would be better to write a program (we'll look at this more closely next month) which sets aside a variable, say K, and makes it equal to 1: 'K = 1'. Then you make the program do 'PRINT E(K,10)', then 'K = K + 1', then 'PRINT E(K,10)', then 'K = K + 1', then . . .

This is what's called a 'loop', and it's something we'll cover in some detail later.

## Strings

As I mentioned earlier in the series, one of the advantages which a computer has over a programmable calculator (apart from increased memory capacity) is its ability to handle text — letters as well as numbers.

Say for example you want a program to print a simple message, like HELLO. In BASIC, all you have to do is 'PRINT "HELLO" '. Notice that the message to be printed out must be in double quotes "". Inside the double quotes, anything goes. Even 'reserved' words like RUN and PRINT will be handled simply as messages to be printed.

So, if you type in 'PRINT "THIS IS A PRINTED OUTPUT" ', the computer would respond with 'THIS IS A PRINTED OUTPUT'.

PRINT statements can be made to output several things on one line, simply by separating them by commas in the print statement.

For example, say in a program you wanted to print out the value of K. Let's set 'K = 6'. Now if we input 'PRINT "K IS", K, "AT THIS POINT IN TIME" ', the computer will respond with 'K IS 6 AT THIS POINT IN TIME'.

What we've done is to print on one line three things: a message (a 'string'), a value, then another message.

This facility makes the output of programs readable. In the chook example, where we wanted to print out the number of size 10 eggs each chook had laid, we could have used: 'PRINT "CHOOK NUMBER", K, "LAID", E(K,10), "SIZE 10 EGGS" ' and the computer would respond with something like 'CHOOK NUMBER 15 LAID 197 SIZE 10 EGGS'.

This is one of the many special facilities available using the PRINT function — we'll look at some more of these later.

As far as the computer is concerned, 'PRINT "HELLO" ' is in the same category as 'PRINT 5'. We're presenting the computer with an absolute value to output. It's also possible to have 'string variables' which hold 'values' in the same way as normal variables do. These 'values', however, are messages.

String variables have a special nomenclature in BASIC. In the same way as integer variables end with a %, string variables end in $. So valid string variable names are: A1$, B$ and K9$.

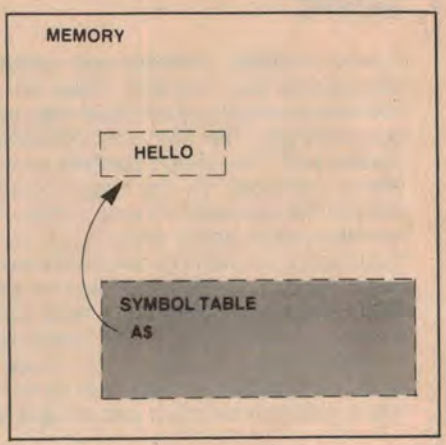



*Figure 4.* **A string variable in memory.**

A statement like 'A$ = "HELLO" ' will produce the situation in memory shown in Figure 4. Notice that the amount of space needed to store a string variable is not always the same — the longer the message, the more space is needed.

'PRINT A$' will cause the computer to output 'HELLO'.

It's even possible to have arrays of string variables. Say for example that each of your 20 chooks had a name, and you wanted to print that name along with the total number of size 10 eggs she had laid. First, you would create space to store the names in: 'DIM N$(20)'. Then you would have to fill it up: 'N$(1)="TAMMY" ', then 'N$(2)="FLO" ', then the rest through to 'N$(20)="MAGGIE" '.

The print statement would look something like this: 'PRINT "THE CHOOK CALLED", N$(K), "LAID", E(K,10), "SIZE 10 EGGS" '. This would give an output like: 'THE CHOOK CALLED MAGGIE LAID 2 SIZE 10 EGGS'. Notice that we're not even bothering to output the value of K; in cases like this, K is called a 'dummy' variable.

Figure 5 shows what N$ would look like in memory. Notice that, because all of the entries in N$ are of different sizes, the symbol table has to keep track of where *each* of them is.

## String functions

What else can you do with string variables, other than just printing them? There are many string functions available in BASIC — in general, the larger the machine and the more ▶

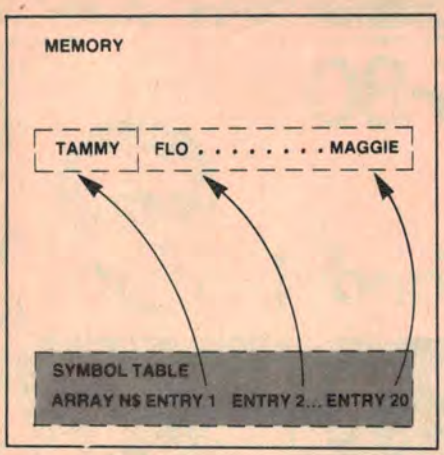



*Figure 5.* **A string array. Notice that, because the entries in the array are different sizes, the symbol table has to hold the position of each of them (there are other ways of doing it, but we won't go into them at this stage).**

comprehensive the version of BASIC, the more string functions will be available. I've listed the more common ones below.

**Concatenation:** This allows two (or more) strings to be joined together to form one long string. Example: 'A$ — "HELLO" ', 'B$ = "GOODBYE" ', 'C$ = A$ + B$'. C$ will now be equal to "HELLOGOODBYE". Typical uses for this function include adding spaces to the end of a string: 'A$ = A$ + " " ' will make A$ equal "HELLO ".

**String Length: LEN:** This function gives the length of a string. If, for example, you wished to know how many characters were in A$, inputting 'D = LEN(A$)' would set D to the number of characters. For example, if A$ was "HELLO", D would be 5.

**Value: VAL:** If a string is in the form of a number, then the VAL function will allow the value of the number to be used in the program. For example, if A$ is equal to "2.3" (and there is absolutely no reason why a string should not include digits), then inputting 'D = A$' would give an error message — the computer does not know that A$ is a representation of a number; it thinks of it only as a string of characters. Inputting 'D = VAL(A$)' would, however, set D to 2.3.

**String Splitting: LEFT$, RIGHT$ AND MID$:** These allow the user to split a string into several other strings (the opposite of concatenation). 'B$ = LEFT$(A$,3)' will make B$ equal to the three left-hand characters of A$. (A$ will be unchanged, by the way.) Similarly, RIGHT$(A$,3) will give the three right-hand characters of A$ and MID$(A$,3,2) will give two characters in A$, starting from the character third from the left. MID$(A$,3) will give the third character from the left in A$.

Some examples:

| input | output |
|---|---|
| A$ = "HELLO" | |
| PRINT LEFT$(A$,2) | HE |
| PRINT RIGHT$(A$,2) | LO |
| PRINT MID$(A$,3) | L |
| PRINT MID$(A$,3,2) | LL |

## PRINT special functions

PRINT is not such a simple function as it first appears. All of the clever output formatting in BASIC computer programming is achieved using the PRINT function.

You already know that 'PRINT A,B' will output the value of A, followed by the value of B. The comma between A and B actually means 'tabulate'. That is, move to the right across the screen until you reach one of the predefined 'columns' built into the computer function.

On a typewriter, you can set the 'tabulations' (or 'tabs', as they are usually called) to any position you like on the typewriter. When you press the TAB button, the carriage will move to the left until it hits one of the tabs you have set. This means that you can type in columns of figures by pressing the TAB key each time to move you to the next column.

A similar system is at work in the computer — but this time the tabs are predefined when you turn the machine on (there is another sort of tab in the PRINT function, but I'll come to that later).

So when you put a comma between the two variable names A and B, the computer will output the value of A, then move to the right until it finds one of the tabs (there are usually four or five, equally spaced across the width of the screen — you can't see them, of course).

This is all very well, but what if you want to 'PRINT "THE VALUE OF A IS", A'. This will mean that the computer will output 'THE VALUE OF A IS', then tabulate to the next tab before outputting A. This could leave a sizable space — very unsightly.

The alternative is to use a semi-colon, ';'. Using one of these between successive variables (or between variables and strings, or whatever) in the PRINT statement will cause the computer to leave no room at all between them on the screen.

This means that if you input 'PRINT "THE VALUE OF A IS";A', and A is 4, then the output will be 'THE VALUE OF A IS4'.

Wait a minute, though, this isn't much use either — we want *some* gap in there.

The answer is to plan your PRINT statements carefully. What if we use 'PRINT "THE VALUE OF A IS ";A' (notice the extra space at the end of the string). The output will now be 'THE VALUE OF A IS 4'. Bingo.

Another nice point to remember is that every PRINT statement will cause the computer to go to the start of the next line when it is finished — unless it ends with a semi-colon. I can't really give an example at this stage because you're still using the machine in the 'calculator' mode, and the machine will take a new line at the end of every output anyway — but remember it later on.

## Tabs

I said earlier that there was another way to tab in BASIC. This is by means of a function called TAB (surprise, surprise).

Inputting 'PRINT TAB(20);"*" ' will cause the machine to move 20 spaces to the right, then output a '*'. Inputting 'PRINT TAB(20);"*";TAB(30);"@" ' will make the machine move 20 spaces to the right, output a '*', then move another 9 spaces to the right (bringing it 30 spaces *from the left-hand edge of the screen),* then output a '@'.

Most machines will ignore a TAB which asks for a move to the left. In other words, if the machine is outputting at a position 30 spaces from the left (call that 'position 30'), and then encounters a TAB(20), it will do nothing.

TAB can be used with a variable — TAB(A) is quite acceptable. This is very useful in plotting applications — wait and see the example program in part five of the series.

Some machines (including the Apple — see this month's ETI) allow *vertical* tabs as well. This means that inputting 'VTAB(10)' will bring the next computer output to the 10th line from the *top* of the screen.

## Graphics

'Graphics' is a word which is used to describe the ability of a computer to put diagrams on the screen.

For example, say I got the computer to 'PRINT "-------" '. This would leave a line of '-'s across the screen.

Now, it doesn't take much imagination to realise that by judicious use of characters like '!' and '-', you could cause the computer to draw horizontal and vertical rulings on the screen, to clarify a table, for example — and in fact this is done quite commonly. ▶

But '-' and '!' are a bit limiting. It would be nice to have characters that were a bit wider, such as '—', so that putting a row of them on the screen would give a '——————————', with no gaps in it.

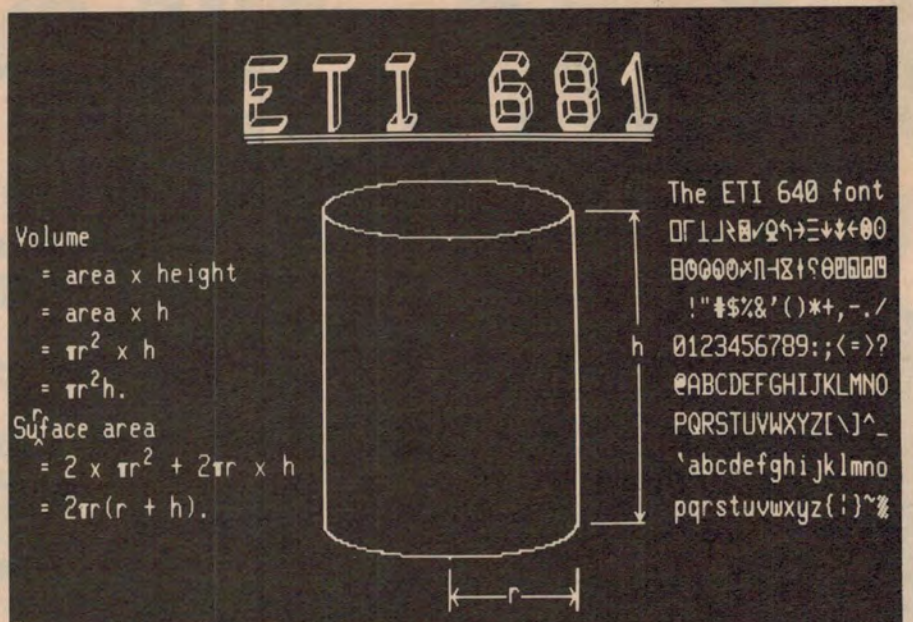
Some machines (notably the PET, from Commodore), have a comprehensive set of graphics symbols which allows not only vertical and horizontal lines, but also areas of tone, 45° lines and a whole host of other shapes.

Some machines, such as the Exidy Sorcerer, allow the user to define some of his *own* characters — which may be anything from the letters of the Greek alphabet to the shape of a duck — for 'duck shoot' games!

The Apple, and other machines, allow the user to switch the machine into various graphics 'modes', in which the screen becomes like a plotting surface, on which lines and curves can be 'drawn' under program control.

In most applications, though, especially where the computer is being used mainly for computation (and that is what the word means, after all), the graphics produced is usually limited to ruling tables and simple graph plotting.●

**Next month, Phil Cohen explains how to write some simple programs, explores the mysteries of 'debugging' and introduces PEEK and POKE.**




This is a photograph taken from a VDU screen which is being 'driven' by our ETI 681/640 VDU with programmable character generator.

The block of characters on the right are the full set which can be produced by the ETI 640 VDU on its own. Notice that there are a small number of characters which can be used to provide various simple 'graphics' outputs — the shaded box at the lower right, for example, and the minus '–' and underline'_' symbols.

The rest of the screen uses the ETI 640 symbols plus a feature which is added by the ETI 681 programmable character generator — user-defined characters.

Although it is not immediately obvious from the photograph, the graphics on the rest of the screen are made up of characters the same size as the ETI 640 characters, but with shapes which make them join up to produce the various patterns shown.