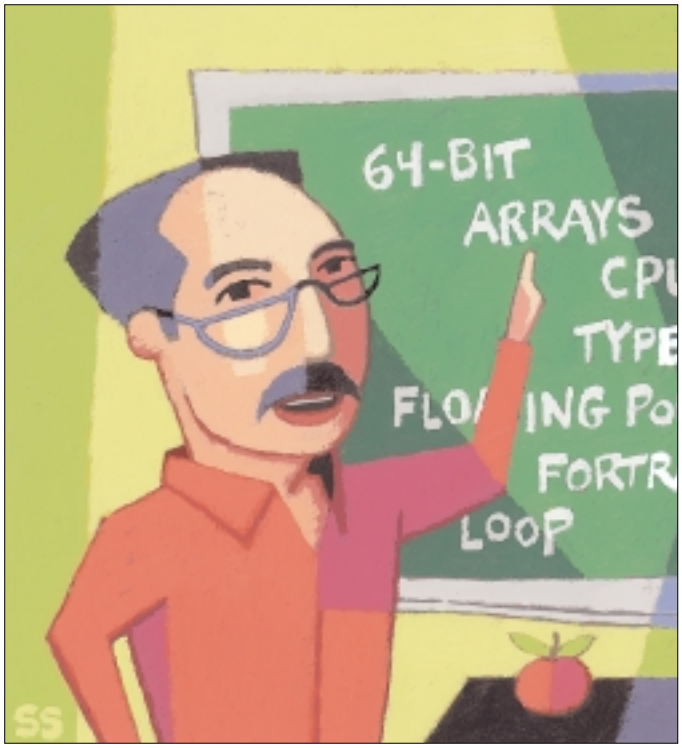
by Peter Collinson, **Hillside Systems**



STEPHEN SCHILDBACH

A Programming Primer

I'm always writing bits of programs for this column, programs that perhaps use the UNIX tools or employ one shell or another. While I was considering what to write this month, it occurred to me that I often make assumptions that underpin the nature of programming. I often don't realize that I am taking these implied concepts for granted, because I've been writing programs for more than 30 years, and they've become second nature.

In many ways in those 30 years, the nature of computing has changed, with each step along the path building on what went before. My son's idea of how his computer actually works is colored by the view that he is presented by the operating system he has grown up with: Windows. When we discussed this, it hadn't really occurred to him that many of the facets of the system he uses were not easily supported by the underlying hardware. He had not comprehended that Windows presented him with an interface that bends the reality of the hardware in a way that is intended to make the system easier to use.

Programmers who were familiar with the way programs worked created the UNIX tools and shells, and many of the concepts used in programming have found their way into the tools that we all use every day. To understand this, let's take a peek at the ideas in programming languages and systems.

I started my programming career by learning FORTRAN IV. FORTRAN was the creation of computer language pioneer, John Backus, whose task in 1954 was to develop a new compiler for the IBM 704 computer. FORTRAN took around 25 man-years to develop. Saul Rosen says in his 1967 book, *Programming Systems and Languages*, "Like many of the early hardware and software systems, FORTRAN was late in delivery, and didn't really work when it was delivered." Somehow, I feel that not much has changed.

FORTRAN is an abbreviation of FORmula TRANslation, and was designed as a scientific language, allowing sets of familiar mathematical expressions to be turned into computer programs. It was groundbreaking. Before it existed, people had tried to create languages, but most programs had to be expressed in machine code. Initially, a human would create the binary codes for the instructions to be executed. Later, the machine helped somewhat. With assembly languages, humans wrote programs using words rather than numbers and the system translated those words into the appropriate binary code. However, in both cases, the programmer wrote code for the machine, and each statement mapped onto something the machine understood as a single instruction.

How Things Work

The way that computers work hasn't changed in any fundamental way since those early days. If we neglect all the peripherals, a basic computer consists of a Central Processing Unit (CPU) and

some memory. The CPU contains *registers* that hold numbers, and all the real work that the CPU performs happens in those registers. The maximum number of binary digits these registers hold gives us the *word length* of the computer. We are now moving toward using 64-bit machines. Nowadays, we are used to thinking in bytes, groups of 8 bits. It was not always so, I did my Ph.D. with a PDP-8 whose word length was 12 bits. For the record, the PDP-8 was manufactured by Digital Equipment Corp. and was the first widely used minicomputer.

The arithmetic unit in the CPU works with whole numbers (usually called *integers*). With a fixed word length, this gives a maximum and minimum number that can be stored. In some ways, I am sure there's an element of magic for many people in the way computers deal with numbers. We all know that numbers are stored as binary patterns, and the computer has the capability to perform a set of fixed operations on the binary patterns. My January column discussed the representation of negative numbers in computers (see "The Time," Page 24, `http://sw.expert.com/C2/SE.C2.JAN.00.pdf`). In fact, what I described was one way to represent positive and negative numbers. The method I talked about is designed so that adding positive and negative integer numbers together "works." The reality is that when we type two numbers into a computer, there are many ways of representing the numbers in binary and we don't really care as long as the result that's printed on the screen is correct, and the result is achieved speedily.

We often want to perform operations on numbers that are not just integers, called *real* numbers. To do this, we adopt a binary representation, enabling the computer hardware to operate on the binary patterns to create the correct answer at the end of the day. If the CPU only supports integers, then we need to use a binary representation that can make use of primitive integer operations. We write a program or a portion of a program that can deal the mapping of our chosen binary representation of real numbers into a set of integer operations and back again for storage.

Consequently, to handle real numbers, we need to use a binary pattern that represents the number in a different way from the way that integers are stored. Perhaps we'll adopt a scheme where some portion of the word will be the integral part of the value, and some portion will be the fractional part. This coding method is often called *fixed-point* number representation.

However, for scientific calculations, we want to deal with very large or very small numbers. This is done with *floating-point* numbers. Floating-point numbers also split the binary pattern into two chunks. Actually, I don't want to get into the actual mathematics behind the representation because it's not easy to explain simply. Suffice it to say that the representation allows a wide range of fractional numbers to be stored and manipulated arithmetically. Floating-point arithmetic is most often done using an additional piece of CPU hardware called the *floating-point unit.*

Incidentally, the representation of floating-point numbers isn't particularly compact. It has been usual to use 64 bits for the storage of the numbers for some time. Another problem is that the way the numbers are stored can sometimes result in imprecise calculations. I didn't say "incorrect," I said "imprecise." Essentially, the representation is prone to "rounding error," where a number cannot be accurately represented and may be rounded to the nearest figure. For this reason, I rarely use floating-point numbers for monetary calculations. It's easier and more accurate to make programs work in pence rather than pounds (or cents rather than dollars on your side of the big pond).

Memory

The memory of the computer contains binary patterns. Some of the memory will hold the program that is executed, so the binary numbers in that part of the memory are values that are understood by the CPU to be instructions. Some of the memory will hold data on which the program operates. Most computers cannot tell the difference between stored instructions and data. So, they cannot object when a programming error attempts to add two instructions or treat some stored data as part of the program.

The best way to visualize memory is to think of it as a set of boxes placed end to end, such that we can say, "put this item in box number 500," or "get me the contents of box number 4506." Each memory location has an address that the CPU uses to identify it. The CPU obtains the value of a memory location by sending the address for the location that it needs to the piece of hardware that controls the memory, and is returned a value. It can also load the memory by sending some data and an address.

When we start the computer running, we load a special CPU register, called the *program counter,* with the first address of the program that's stored in memory and say "Run." The CPU now goes into a loop. First, it fetches the contents of the memory location that's stored in the program counter. Second, the contents are assumed to be an instruction for the CPU. Third, the instruction is decoded and the appropriate action is taken. Finally, the program counter is incremented to automatically point at the next memory location in the program.

The program, then, is a set of instructions that are executed sequentially. This is normally called the *flow of control.* Each instruction is at a primitive level. For example, "load this register from this memory location," or "add these two registers together," or "place the contents of this register in memory location number 272386."

We can affect the flow of control by loading the program counter. It's easy then to create a loop, jumping back to the start of the program. We can also load the program counter conditionally, depending on the result of a test. Often this is expressed as "if the result of the last operation was zero, then jump to this memory location." Of course, we can often test different aspects of the last operation, like "was it nonzero?" or "was it negative?"

High-Level Languages

FORTRAN was born into a computing world where it was the norm to program in fairly low-level machine instructions. Programs were hard to write, and even harder to debug.

The original FORTRAN paper by Backus (and others) is reprinted in Rosen's book and goes to great lengths to justify the time saved by using FORTRAN over previous methods. It seeks to calm the fears of the people who doubted its efficiency. Yes, it really could generate programs automatically from complex statements that ran as fast as those that were hand-coded.

Another feature of FORTRAN was apparent when I started to use it. It allowed programs to be portable from machine to machine. I didn't need an IBM machine to run FORTRAN. By creating what we think of today as a "model" of how a computer worked and supplying a method of mapping the model onto the underlying hardware, you could take a program written for one machine, translate (or compile) it, and run it on another.

Some elements of the FORTRAN model are still in use today. For example, a program is a series of statements that are executed in order from the first line to the last. This idea arises from the way that the computer itself works. Essentially, the flow of control in a program is similar to the flow of control in a computer. This notion is so deeply embedded in almost everything we do that it seems almost a truism to talk about it in any great depth.

Most statements in FORTRAN perform some form of arithmetic and generate a result that is stored in memory. We label the memory address with a name and call it a *variable*. The statement looks like an algebraic equation:

```
A = B*C + D*E
```

The `A` here is the destination of the value of the computation that takes place on the right-hand side of the statement. The right-hand side is written in familiar algebraic form that we all learn at school, so we know that the above example means multiply `B` by `C` and add that result to the product of `D` multiplied by `E`. If we want something different to happen, we use brackets:

```
A = B*(C + D)*E
```

This type of statement is called an *assignment* statement, because we are computing a value and placing it into a variable. There's often a need to include constant numbers on the right-hand side of an assignment statement, and you just write them in as needed.

However, the use of the equals sign for assignments can sometimes be confusing to the mathematically trained mind. It doesn't mean "mathematical equality," it means use the left-hand variable as a destination for the computation.

The problem of the interpretation of the equals sign is made plain by the idea of incrementing a variable, written like

```
I = I + 1
```

which says take the value from the memory location, add one to it and put it back in the same memory location. Programs often contain statements like this. We use `I` to control a loop, counting the number of times that a section of the program is executed.

Types

What about dealing with floating-point numbers? How do we force one calculation to be done using integer arithmetic and another to use the floating-point unit? This is a question about the *type* of variables. Knowing the type has also been important on many machines because a floating-point number occupies twice the number of bits an integer does. Also, the language compiler will wish to generate different instructions to handle a floating-point add operation than it uses to execute an integral one. However, there is nothing in the statements above that overtly indicates the type of the variables that are being used.

FORTRAN adopted a simple policy. If a variable name started with `I`, `J`, `K`, `L`, `M` or `N`, then it holds an integer, otherwise it contains a floating-point number. The choice of letters came from common mathematical practice. We are left with this legacy today. Many programmers use `I`, `J`, `K`, `L`, `M` or `N` for simple counters and other integral values.

As time went on, it was realized that just being able to invent a variable as the program unfolded was a large source of bugs. You could mistype a variable name and nothing would spot that there was a problem. Most modern languages insist that you declare all variable names. You say at the start of the program: "These are the variable names I intend to use," and the compilation process will detect any mistypings and complain loudly.

Arrays

The objects that you did have to declare in FORTRAN were arrays. You used a `DIMENSION` statement to tell the compiler that a particular variable name was to be tied to the start of a contiguous section of memory. The size of the array, essentially the number of variables the array can store, is given by the `DIMENSION` statement. The standard variable typing rules that depend on the initial letter of the array name were also applied to the array, so the array would have a type and elements from the array would behave correctly in arithmetic statements.

An array is essentially a section of memory that contains *n* variables, where *n* is the size of the array. Some way is needed to access each element of an array, because few languages are able to handle arithmetic operations on the whole array. The convention is to use a subscript in brackets:

```
Z = A(1) + A(3)
```

This will add the first variable to the third and place the result in `Z`. Some languages use square brackets for array access rather than round ones, largely because it makes things easier if the compiler can distinguish between the round brackets used to impose precedence on arithmetic expressions and those brackets used to indicate an array access. Also, some languages define array indexes to run from zero to *n*-1.

You can place an array reference anywhere in the code where a variable is written normally. The real power of arrays emerges when you replace the value inside the subscript brackets by a variable. To sum the values of an array, you'd say something like the following:

```
  SUM = 0.0
  DO 5 I = 1,1000
5 SUM = SUM + A(I)
```

The best way to understand this is to walk through it. The first statement sets the accumulator to zero. The `DO` statement sets up a loop whose last statement is marked by the label `5`. Each time around the loop, `I` will take new value. It starts at 1 and is incremented by one until it is equal to 1,000, at this point the loop terminates and control passes to the statement immediately after the end of the loop.

The first time around the loop, `I` will be 1 and the contents of the first array element will be added into `SUM`. The next time, `I` will be 2 and `A(2)` will be added in and so on. Eventually, the sum will be calculated.

In a very few lines, we can achieve a task that not only would be a bore to write out, but doing so would also be prone to errors. Notice also that by replacing the 1 and 1,000 in the `DO` loop by variables, we can calculate different sections of the array using the same basic code. If you think about this idea, what's happening is that we are using data (the contents of the start and stop variable for the loop) to control the program. I often write very general programs that are data-driven in this way.

All the main languages support arrays. Many, including FORTRAN, support two-dimensional arrays aimed at scientists that wish to program operations on matrices. Many scripting languages support *associative arrays*, where the index is not a number but a text string. Again, the idea is to allow the programmer to write code that processes a single element. I use associative arrays an immense amount in Perl, often to allow me to data-drive the general-purpose code I have written.

Evolution

We have always been standing on the shoulders of someone else's efforts as computing has developed. What has happened has been often conditioned by what went before. FORTRAN provided high-level access to a set of underlying facilities provided by a machine. The primitive objects in the FORTRAN world mapped pretty closely onto what the machine could do.

However, programmers have wanted or needed to handle other types of data in a simple fashion. For example, I suspect nearly all the programs I have written in my life have been concerned with handling text and not numbers. In C, which was designed as a high-level assembler, with primitive operations mapping directly onto the hardware, strings are handled as arrays of characters. A set of standard routines is implemented to provide the functionality I need as a programmer. Other programming languages have handled strings by allowing the syntax of the language to cope, so in many languages, you can join strings together:

```
world = "world";
str = "hello " + world;
```

This looks like familiar assignment syntax, but it is doing complex string handling using a set of hidden routines. One of the reasons for doing this is it feels natural for the programmer to extend their assumptions about variables and assignments into the field of string handling.

Further Reading

In this article, I've referred to one of my undergraduate texts that still lives on my working bookshelf. It's called *Programming Systems and Languages,* edited by Saul Rosen and published by the McGraw-Hill Book Co. in 1967. It seems to have predated ISBNs. I suspect the book is out of print, so hit your local library if you are interested. It contains a bunch of early papers on language design and development, the legacy of which we are still living with today.

Peter Collinson *runs his own UNIX consultancy, and is dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.