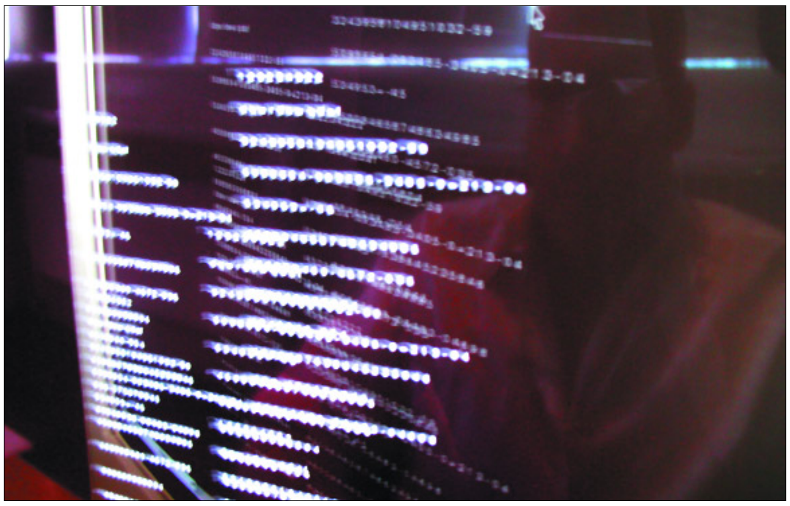
Speaking with tongues



So what is your favourite programming language? It is a question that never loses its appeal during coffee breaks and is sometimes even discussed in more elevated forums – including this magazine (see *Physics World* September 2001 p19). However, there is not really a sensible answer to the question "which language is best?"; most people seem to agree that the choice of language depends on the problem.

FORTRAN is excellent for much scientific work, whereas C and C-based languages such as C++ and C# are a more natural choice for systems programming. FORTRAN's success was largely due to the power of its input/output (I/O) architecture. It lets you use the same READ or WRITE statements regardless of the peripheral involved – be it a printer or monitor – and whether you want formatted or unformatted transfers. When FORTRAN first appeared in 1953, this was an impressive piece of software engineering. It set a standard that has seldom been approached since by other languages.

C, on the other hand, is sometimes described as a high-level assembly language. There is some truth in this point of view because C was developed in the early 1970s to enable the Unix operating system to be written. Among other things, successful C programming demands that you have a detailed grasp of memory layout and address manipulation, otherwise you will come to a stop the moment you want a procedure to deliver values back to your program.

Over the past 40 years I have enjoyed writing in many languages, but my current interests using Windows API (application programming interfaces) demand the use of a C-based language. The fun and enjoyment remain undiminished, but when I am flying back to my local airport I begin to worry a bit, and you might see why in a minute. C is extraordinarily picky and unforgiving in some respects – such as manipulating memory addresses – but equally laid back in others.

To understand why, take the following code fragment, which I hope both C and non-C programmers will be able to understand.

```
int i = 1;
if (i = 2) printf ("stupid");
if (i == 2.0) printf ("even stupider");
```

This compiles on all of my C and C++ compilers; the constructs are therefore legal. But execution is another matter. The program obediently prints "stupid" and "even stupider", but in any half-sane world it should not do either. In fact, a program like this should not by rights compile at all.

Why not? Let's take a moment to examine the code. An integer *i* is first declared and initialized to 1, so no problem here. Then we want to test to see whether *i* equals 2. Because it does not, you should not get the "stupid" message, but you do. Old C hands, if they are alert, will know, of course, that this statement does not test whether *i* equals 2 at all. It actually sets *i* to 2; in any case, the conditional is virtually meaningless in C when it is written in this way.

To test the value of *i* you have to use the double-equals sign in the next statement. But because *i* is now 2 – which we did not intend – this is also satisfied and you get "even stupider" printed out. There is also a subtler problem in the last line, because here we are testing an integer against a floating-point quantity. C does not bother itself much about such trivia, but it should! Thus this simple little program demonstrates three different problems with C.

Just think how easy it is for a tired programmer to make mistakes

Actually, this sort of thing (and there are other problems of this kind in C as well as other current languages) worries me. My airport qualms centre on whether there are hidden bugs of this sort in the air-traffic-control code when my plane is landing. Just think how easy it would be for a tired programmer, working to tight deadlines, to make mistakes such as those above, which might not be detected until a system is operational. How anyone can defend the use of a language like this for the millions of lines of code in a typical safety-critical application, and/or its operating system, defeats me.

Yet other languages have been around for at least 30 years that would rigorously trap errors such as these when the program is compiled. Algol-68 was one of the first, but only a handful of compilers were ever implemented. One reason was that its designers loftily ignored the requirements of real-world programming by not including any I/O at all. It therefore, quite rightly, died almost before it was born. What a pity though – its heart was in the right place.

Where does all this leave us? Do physicists have a role to play? Obviously yes, especially those who work with software. But no matter how good the product, we have to ensure that it will be taken up and used widely, as both FORTRAN and C have been for several decades despite their shortcomings. This requires not only intellectual excellence but an appreciation of what the customers (programmers) want. We also have to appreciate the dynamics of business so that your all-singing, all-dancing ideas for a new compiler will not bust the company's R&D budget while simultaneously appealing to the marketing department.

It's all jolly difficult. Does anybody else share these concerns, or am I just getting paranoid in my old age?

Colin Pykett was a physicist in the UK's Ministry of Defence and now does freelance research. He is currently developing novel digital musical-instrument techniques, and making lots of coding mistakes, e-mail cep@pykett.org.uk

● Readers are invited to submit their own Lateral Thoughts for possible publication. Articles should be between 900 and 950 words long and can be e-mailed to pwld@iop.org