TRICKS OF THE TRADE

Most people teach themselves programming by using the manual that comes with their computer. This is a good enough way to get started, but it often means you never learn to write efficient programs nor discover the tricks of the trade that make programming easier. We introduce a series of articles designed to give you insights into the techniques used by good programmers.

Good programming is developed through experimentation and experience. The novice programmer, often solving problems through enormous enthusiasm and sheer effort, is gradually transformed into a technician with an awareness of short-cuts and rule-of-thumb methods that achieve the desired results. Eventually, the programmer will develop the simple clarity and direct approach of the expert. But there is no reason why the personal progress of a home computer programmer cannot be hastened by learning from the mistakes of others who have taken the same path. The lessons are there for the learning, and everyone's programming can benefit from them. Our course begins with a discussion of some of the more helpful hints that can aid a beginner.

Programming is a problem-solving process, and a great part of it should be carried out in the mind and with a pencil and paper long before a line of code is written. The stages in this process are well-known: a clear comprehensive statement of the problem in practical terms, followed by repeated re-statement of the problem with increasing precision, until it is formulated with as much detail and accuracy as possible. This description nearly always contains or implies the essential solution, which must then be expounded in greater and more practical detail so that it becomes a working method. In programming, only the last stage should involve coding, and that should be a straightforward realisation of the preceding stages. When the coding stage overlaps the real problem-solving, poor solutions and bad code result.

Solutions are often known as *algorithms*, processes of computation analysed in logical stages. The efficiency of a program depends mainly upon that of its algorithm, and this is judged in terms of its 'completeness' and its 'correctness'. These two commonsense qualities refer to the program's theoretical and practical ability to cope with the foreseeable range of input conditions, and to the consistency of its internal logic. Needless to say, it's much easier to recognise their absence than to demonstrate their presence, but every program must be subjected to this judgement, and the earlier in its development the better.

Solutions must be *reliable*, as well as complete and correct. Not only must they handle their prescribed range of problems, but they must also deal predictably and safely with conditions outside their range. This usually means having the ability to recognise potential error conditions, and being able to stop operating with all the data intact, as well as displaying some useful status message. It is difficult to judge whether code is sufficiently reliable, as a program that isn't reliable is easier to recognise than one that is. Experience leads to better judgement.

Making programs reliable and robust is a worthy aim that nearly always conflicts directly with an equally desirable goal — keeping them *economical*. Everything costs money, even if it's only the time you spend writing programs for fun. There always comes a moment when you have to decide between continuing to work on a program that's nearly 'bombproof', and abandoning it to start a fresh project. Even if your time is unlimited, the computer's memory and operating speed are not. It's quite possible to surround the central algorithm with so much precautionary code and error-trapping that protecting against crashes can take more time than solving the original problem.

TESTING AND DEBUGGING

Solving analytical and logical problems in theory is enormously important, but programs are meant to perform a task. Once the first syntax and logical errors have been dealt with it's time to begin *testing*. This is so familiar an idea that it hardly seems to merit statement, never mind emphasis. But it is, in fact, a much misunderstood process. In anything but trivial programs there are usually far too many possible combinations of input conditions for exhaustive trials, so tests must be devised to put as much strain as possible on what are likely to be the most vulnerable (and what are expected to be the strongest) parts of the program. Generating comprehensive test conditions is not a simple matter and takes time and money. The professional approach to testing is that there are no perfect programs, only bad tests.

Successful tests reveal a program's inadequacies, and should do so in a logical fashion so that *debugging* takes as little time as possible. Like testing, debugging is an essential process that regularly fails to be achieved precisely because it embodies the same human failings that make it necessary in the first place. A program bug should be approached as another problem to be solved, exactly as described earlier — statement, analysis, algorithm, testing — but it is most often treated as a casual pest to be swatted, poisoned or crushed, with