FAULT LINES

The detection and correction of errors is an important aspect of program design. Problems may be caused by typing errors, but faulty logic or a misconception of the program's function can have more serious results. We examine potential trouble-spots — at the interfaces between subroutines and between the program and its user.

There are many potential sources of error at each stage of a program's creation, from its specification, through design and coding, to the testing. Errors are often introduced at the specification and design stages if little thought is given to the nature of the problem and insufficient care is taken to ensure that the program does exactly what it is supposed to do. We can reduce the chances of these mistakes occurring by following the structured design methods outlined earlier in the course (see page 476). Further errors are likely to arise as the design is translated into code — poor typing can introduce bugs, as anyone who has ever misspelt a variable name knows only too well! — and even testing and debugging can cause other mistakes when a correction to one fault itself leads to others.

But it is at the interfaces — between routines and between the program and its user — that most errors are to be found. Particular care should be taken to ensure that any values passed across these interfaces are of the correct data type and fall within the range required by the program. Values may be checked either within the routine that passes them or in the routine that accepts them; the process of checking values as they pass between routines is known as 'firewalling'.

To ensure that values output by a routine are in an appropriate range and are of the right data type, checks should be carried out if the output depends on a value entered by a user or read from a file. Values that are entered into a routine should always be checked. Subroutines can be designed to give a well-defined set of outputs, but human beings do not operate so methodically and tend to have a wide range of different responses to any given prompt, so stringent checks must be placed in any routines that accept data from users. Similarly, files of data may be corrupted or misread, so checks should be placed in all file-handling routines.

Errors do not often cause programs to crash. When they do, it is because the program has broken a rule of the language (using an operator illegally, for example, as in RESULT = FIRST$ + SECOND$) or a rule of the operating system (opening too many files at the same time, say). The following code would appear to be a perfectly legitimate program:

```
10 FORCOUNTER = 1 TO 10
20      SUM = SUM +1
30      PRINT COUNTER, SUM
40      GOTO 10
50 NEXT COUNTER
```

However, it is a non-terminating algorithm and will crash the system because of the way the language works. In this case, the language (BASIC) uses the 'stack' to keep track of FOR... NEXT loops, adding to the stack each time a new loop is started. In this program, line 50 (with the NEXT command that would decrement the stack) is never reached, and so the stack gradually fills up until eventually a 'stack overflow' message is generated and the interpreter stops the program. Errors such as this are usually easily spotted, but if they appear in rarely used sections of code thorough testing may be needed to uncover them.

A more insidious type of error is one that allows

Error Checklist

A logical structured approach is the essence of error avoidance and debugging; the following error checklist (from an idea by G J Myers in 'The Art Of Software Testing') is an abbreviated example of such an approach

Variables

1 Are all variable names unique, bearing in mind that many interpreters use only the first two characters of any name?
2 Have any variables (especially loop counters or subroutine parameters) been re-used while their contents are still significant?
3 Are array subscripts within bounds, and are they whole numbers?
4 Do array subscripts start at element zero or element one?

Calculations

1 Do calculations yield string or numeric results, and are the results assigned to string or numeric variables?
2 Does any calculation result in a number too small or too large for the computer to handle? Can this cause a 'divide by zero' error?
3 Can rounding errors be significant?
4 Are all operations in an expression executed in the correct logical order, as opposed to the order imposed by the precedence of arithmetic operators?

Comparisons

1 Are strings always compared only with strings, and numbers with numbers?
2 Does it matter if a test string is wholly or partly upper-or lower-case?
3 Are strings of unequal length being compared, and does the difference in length matter more or less than differences in characters?
4 Are Boolean and comparison operators being mixed properly? A>B OR C is not the same as A>B OR A>C, for example.
5 Does the precedence of Boolean and comparison operators affect the execution of any comparison expression?

Control

1 Do loops and algorithms terminate whatever the state of the variables?
2 Do loops and routines have only one entry and exit point each?
3 When an IF...THEN statement fails, does control pass to the next program statement or the next program line?
4 What happens if none of the test conditions in a multiple branch statement is satisfied?