PIECES OF THE PUZZLE

The most efficient way to create programs in any language is to use 'modular structuring'. Some languages, such as PASCAL, encourage this approach, while BASIC users need to discipline themselves to adopt the technique. We show you how your programming will be greatly enhanced using modules of code as your basic components.

A module is a piece of code that performs a particular function. The points of entry and exit, known as the module's 'interfaces', must be precisely defined, and the processes that occur between these interfaces should be entirely independent of the rest of the program. Once a module has been written, it can be treated as a 'black box'. Data may pass in and out of the module's interfaces, but what goes on inside can be left to itself.

Modules can be joined together to build up a program without the writer having to worry about how they perform their tasks. A stock of modules can be built up by a programmer to be used when needed, and programmers can pass modules on to be used in another writer's programs. But in order to take advantage of modular structured programming, we need to take careful note of the flow of control and the flow of data when we write the modules.

To ensure that all your modules behave in the same way with respect to the flow of control, a very simple rule should be followed: all modules should have a single entry point and a single exit. What this means in practice is that the flow of control within the module has to be carefully designed so that it starts at one place and, no matter how much it loops and branches, it reaches the same exit by all possible routes.

Modules correspond to the algorithms we have been looking at in previous instalments of the course. 'Structured' languages, such as PASCAL, allow the programmer to create subroutines that may be called by name, and which use their own variables. Such languages encourage a programmer to enter or leave a routine (called a 'procedure') by single entry and exit points.

In BASIC, using the GOSUB . . . RETURN combination, a subroutine can be called from the main program and, after the subroutine has been carried out, control will return to the line immediately after the GOSUB command. However, there is no restriction on which line the GOSUB sends control to. Two different GOSUBs may send control to different lines of a subroutine with a single RETURN, and the result might be completely different in each case. Similarly, there is no restriction on how many RETURN statements may be used in a subroutine.

This means that the BASIC programmer must be self-disciplined. You should start by making sure that all GOSUBs to the same subroutine point to the same line number, and that every subroutine has only one RETURN in it. It is best to get in the habit of marking the first line of each subroutine with a REM statement giving it a title, and use that line as the entry point. Make the RETURN the last line of the subroutine. This is not essential but it makes things much clearer.

THE GOTO RULE

Extra care should be taken with the GOTO command, which can play havoc with program structure. The rule here is: only use a GOTO to send control to a line *within* the same subroutine. This avoids the potential danger of skipping over a RETURN or passing control to the wrong RETURN. There are times when it is necessary to leave a routine without executing every line. In this case you should GOTO the line with RETURN on it, and there should be no problems.

Using GOTO within loops is even more dangerous. If control jumps out of a loop, BASIC cannot know this and assumes that the rest of the program is the body of that loop! The safety rule is: when in the body of a loop, never GOTO a line outside the body of that loop. If a loop needs to be terminated early, set the loop counter or test variable to the terminal value and GOTO the test line (the line with NEXT or WHILE in it). As with the RETURN statement, put NEXT or WHILE on a line of its own to make this easier. Keeping track of the structure of a program is a lot simpler if GOTOs are avoided as much as possible.

Branches are the most likely place for control to go astray, so try not to allow any decisions to send control out of a subroutine unless it is with a proper call to another subroutine. Remember that each subroutine has a single exit point, so make sure that it is possible to follow the flow of control through every branch to that point. Drawing a flow chart for the routine makes this easy to check. Setting a flag can often reduce the need for GOTOs in routines involving loops and branches.

We can think of data passing in and out of modules, just as we did for algorithms (see page 386). So that modules can be used independently of each other, you must design them so that the only influence they have on each other is through the data that passes between them. The main program passes data to a module and, when the module has been executed, any result that has