TOP SECRET

Our course on program design has so far shown how programs may be constructed from small, largely independent units called modules. We have looked in detail at how such building blocks are designed, and here we show you how to use them in the development of a complete program.

When building a program, it is a good idea to develop an overall structure, consisting of a base level of general-purpose routines that are used by other routines of increasing specialisation on higher levels, all under the direction of a single control module at the top. This 'pyramid' structure will allow us to use a design method called 'program refinement' or 'top-down design'.

Top-down design, as its name suggests, entails designing the topmost control program first. We describe its functions in terms of calls to 'lower' level routines and, for the time being, we need not worry too much about how these lower-level modules will work. Once this is done, we move down a level and describe the workings of each routine called by the top-level module. Each routine is described in terms of the routines it must call, and this process is repeated level by level until we reach the lowest level. At that stage, the functions performed by the routine we are describing are so simple that they may be defined by using the programming language itself.

As an example, let us look at the design of a 'Hangman' game. Instead of the player trying to guess a word selected by the program, as is the case with most computer versions of the game, we want the program to guess a word that we have chosen. One way of achieving this, without giving the program a long list of English words, is to enter data on the likelihood of particular letter sequences occurring.

```
100 REM Initialise variables and arrays


500 REM *****Control Routine***********
510 REM
520 GOSUB 1000:REM Title & Help screens
530 GOSUB 2000:REM Set up Board
540 GOSUB 4000:REM Find word length
from player
550 GOSUB 8000:REM Select data set and
load it
560 GOSUB 3000:REM Guess a letter
570 GOSUB 4500:REM Check guess with
player
580 GOSUB 5000:REM Update the board
590 IF GAME_NOT_OVER THEN 560: REM
guess again until game is over
600 IF WIN THEN GOSUB 10000 ELSE
GOSUB 11000:REM Give appropriate
ending for win or lose
610 GOSUB 6000:REM ask the player for
another game
620 IF ANOTHER THEN 530:REM if
another then start again
630 GOSUB 7000:REM say goodbye and stop
640 END
```

We know before we star: that certain things must be done: variables need to be initialised, arrays must be dimensioned, the 'board' display has to be set up and updated as necessary, and routines must be written that keep the score, that make guesses, and that end the game.

Our first attempt at designing the control routine has a simple REM statement to indicate that variables and arrays must be initialised – we can fill in all the necessary details at a later stage. The control routine itself is simply a pair of loops. The outer loop (line 620) tests to see whether the user is signalling the end of a session, while the inner loop (line 590) tests to see if the game has ended.

Should we need to test the control routine, we must set up dummy subroutines to match the GOSUBs. Each GOSUB in the control routine should have a REM statement to explain its function and should start at a convenient line number – preferably one that is a round figure, such as 1000 or 5000. It is a good idea to ensure that routines with similar functions are given standardised line numbers; this will make life easier when routines are moved from one program to another. For example, game instructions might be contained in a subroutine that begins at line 1000, while a GOSUB 7000 program line will always end a game by calling a standard routine.

Our initial control routine is kept short and simple. It will fit onto the screen and therefore is easier to understand and debug than a program that extends over several screens. The three variables, GAME NOT OVER, WIN and ANOTHER, are all flags that are set in the various subroutines called by the control routine and are used here to determine whether the control program works in the way we intend. It should be quite easy to spot any errors in logic in this simple control routine.

At this stage it is necessary to look at the program's structure with a critical eye – we need to ensure that the program behaves as it should in all circumstances. We can also start to make improvements in the program design; for example, we might like to make the instructions available at any stage of the game and it might also be a good idea to keep a record of how many games the computer or player has won and a list of words that beat the program. Any or all of these changes can be made at this stage.

The next step is to specify each of the