This part of subroutine Y is reading values into a subscripted array, where the subscript is denoted by the variable L. If subroutine Y is called after subroutine X, and if the test condition in subroutine X has been met (that one of the characters is a " . "), the value of L would be completely unpredictable and so we would not know which element of the array values were being assigned to in subroutine Y. Apart from the error of branching out of a loop, this subroutine also uses a GOTO, and this practice should also be avoided. GOTOs lead to confusion and they should be avoided wherever possible.

To avoid confusion when using variables, it is good practice to make a list of them at the pseudo-language stages of program development, together with notes saying what they are being used for. Some languages (but not BASIC) allow variables to be declared as 'local' or 'global' — that is, they have values that apply either in only part of a program (local) or throughout the whole program (global). Many variables, such as those used in loops (for example, the L in LET L = 1 TO 10), are almost always local, so it is often wise to initialise the value of the variable before it is used (for example, LET L = 0). Some languages, such as PASCAL, insist on this; and although BASIC always assumes the initial value of a variable is 0 (unless otherwise stated), initialising is still recommended.

So far we have formulated a reasonable definition of a name for the purposes of our computerised address book, and developed some routines that can handle names in various ways that we shall use in our complete program. Now let's once again distance ourselves from the details of program coding and consider the structure of the 'records' in our address book 'file'

The terms 'record', 'file' and 'field' have fairly specific meanings in the computer world. A *file* is a whole set of related information. In a computer system it would be an identifiable item stored on a floppy disk or on a cassette tape and it would have its own name, usually referred to as a filename. We can consider our entire address book as a file, and we shall call it ADBOOK.

Within a file we have *records*. These are also sets of related information. If we think of our address book as a card index box, the file would be the whole box full of cards and the records would be the individual cards — each one with its own name, address and telephone number.

Within each record we have *fields*. The fields can be considered as one or more rows of related information within the record. Each of the records in our ADBOOK file will have the following fields: NAME, ADDRESS and PHONENUMBER. A typical record would look like this:

```
Peter Edvadsen
16A Holford Drive
Worsley
Manchester
061-540 2588
```

In this record there are three fields: the name field, which comprises alphabetic letters (and, possibly, the apostrophe in names such as Peter O'Toole); the address field, which comprises a few numbers and many letters; and the telephone number field, which comprises only numbers (ignoring the problem of whether or not to allow hyphens in numbers like 01-258 1191). Before we can begin to write a program to handle complex information such as this with flexibility, we must decide how to represent the data within the computer. One way might be to consider all the information within a record to be just one long character string. The problem with this approach is that extracting specific information is extremely difficult. Let's assume that the following entry is just one long character string:

```
PERCIVAL R. BURTON
1056 AVENUE OF THE AMERICAS
RIO DEL MONTENEGRO
CALIFORNIA
U.S.A.
(415) 884 5100
```

If we were searching the records to find the telephone number of PERCIVAL R. BURTON, would it be safe to assume that the last 14 characters in the record represented the number? What if we had included the international dialling code, like this: 0101 (415) 884 5100? Then the number would have had a total of 19 characters. To overcome this difficulty, the telephone number is assigned a separate field, and the program will give us all the characters (or numbers) in that field when requested.

The difficulty with this approach is that there has to be some way of relating the various separate fields, so that referring to one field (the name field, for example) can give us the other fields on the record, as well. One way this could be tackled is to have a further field associated with the record just for indexing purposes. If a record was, for example, the 15th record in the file, its index field would contain the number 15. This could then be used to point to the elements in a number of arrays. To illustrate this, let us suppose one record looked like this:

Jamie Appleton	NAME field
15 Pantbach Road	STREET field
Llandogo	TOWN field
Gwent	COUNTY field
0594 552303	PHONE NUMBER field
015	INDEX field

If we knew the name of this person and wanted his telephone number, all we would have to do would be to search through the elements of the array holding the names until a match was found. We would then find which element of the array the name was in — in this case, number 15. Then all we would need to do would be to find the 15th element in the PHONE NUMBER array to get the right telephone number.

If we had a number of friends in the Forest of