Discussion and Correspondence
Structured programming and input statements

J. Inglis

Department of Computer Science, Birkbeck College, University of London, Malet Street, London WC1E 7HX

This short paper proposes a simple improvement to the input statements of the commonly used programming languages.

One of the types of program most frequently set as exercises for beginners is the type in which records of an input file (very often a deck of cards) are processed sequentially. The natural form of such a program is shown below as Program 0, written here in an informal ALGOL-like notation; the function *read* (x) obtains the next record from the file whose name is x.

```
Program 0 initialise;
          while not end.of.file (x)
            do [read (x); process.record];
          finish;
```

It is unfortunate that this type of program cannot usually be expressed naturally in current programming languages and that students are compelled, very early in their learning, to resort to an unnatural approach which sows the seeds of bad programming style.

Appendix 1 describes the three common types of input statement. Input statements in some variant of these basic forms are to be found in COBOL, FORTRAN, PL/1 and (implementations of) ALGOL 60. In each case Appendix 1 shows: (*a*) a 'well-structured' program—using **while** but not **go to**—and (*b*) an 'unstructured' program—using **go to**—both corresponding to Program 0, above. It appears that in all three cases the 'well-structured' version is no more natural or comprehensible than the 'unstructured' version.

There are two reasons why so-called 'structuring' does not have the desired effect in these cases. In the first place, they belong to a class of programs which cannot satisfactorily be structured by the use of the **while** statement alone—Knuth (1974) discusses additional constructs which have been suggested or implemented to deal with such cases. In the second place, and more importantly, the programs in Appendix 1 belong to that class of programs only because the input operations provided in so many programming languages are themselves unnatural and compel the programmer to distort the structure of his problem.

An analogy should make this last statement clearer. When I go home tonight, I shall look in the drawer in which my clean shirts are kept. Now, I don't want a clean shirt until tomorrow morning; but, if there are no shirts left in the drawer, I shall have to take some action tonight (such as avoiding putting today's shirt in the laundry basket; or ringing my friends to say that I'll be staying at home tomorrow). If the standard logic of input statements were applied to my shirt-handling, I would be forced to take tomorrow's shirt out of the drawer tonight, and not be able to put it back; I would have to put it somewhere else where it wouldn't be crushed, and I'd have to remember tomorrow morning to obtain it from there rather than from the drawer. I could, of course, avoid all that trouble by not looking in the drawer until morning; but if there were then no shirts in the drawer it would be too late to take the action which is possible tonight.

That is precisely the dilemma which faces the programmer today in so many programs. He cannot test for the end-of-file condition without the danger of receiving a record which he may not want at the time of testing, nor can he ever ask for the next record of a file in the secure knowledge that such a record exists. (Of course, he is so accustomed to these limitations that he doesn't really recognise a dilemma—he just writes messy programs.)

What is needed is a system boolean function *end.of.file* (x), which yields the value **true** at any time after the last record of file x has been obtained by the program, rather than after the first unsuccessful attempt to obtain a record from the file. Most data management software would have no difficulty at all in evaluating such a function; but, in cases where input is direct from the device to the user, one additional buffer would be necessary.

The required end-of-file function can, of course, be simulated in today's languages by preprocessing or insertion of additional data declarations and standard procedures, but such simulations are distracting and have undesirable features. In any case, the added code is usually repeating some of the processes already carried out at a lower level by the data management software.

Clearly, if such a function existed in the popular high-level languages, it would make a large contribution to achieving good structure in programs; perhaps the most significant contribution would be that *beginners* could write natural well-structured programs—even when they had input data! Yet it is unusual in the literature of structured programming to find more than an occasional passing reference to input statements—the die-hard ALGOL 60 attitude of ignoring input and output appears still to be very much with us.

Appendix 1

Programming with the three common types of input statement

(1) *read* (x) is defined: 'If the end of file x has been reached, then abort the program; otherwise, make the next record from file x available to the program.'

Note: (i) In this, the crudest, case, the programmer has to know in advance how many records are in the file *or* rely on information in the file itself (such as a record count or a recognisable final record) *or* stipulate that a final 'dummy' record be inserted at the end of the file. The last of these alternatives is assumed in the programs below. Clearly, all of these practices are vulnerable to errors in the data.

(ii) In program 1(*b*) (and in program 2(*b*)), it appears more 'natural' to regard the call of the procedure '*finish*' as a top-level statement, rather than to use **else**. '*Finish*' may in many cases constitute the main part of the program.

```
Program 1(a)        initialise;
                    read (x)
                    while not last.record
                      do [process.record; read (x)];
                    finish;
Program 1(b)        initialise;
             again: read (x);
```

```
        if not last.record
          then [process.record; go to again];
        finish;
```

(2) *read* (*x*) is defined: 'If the end of file *x* has been reached, then assign the value **true** to a system variable *end.of.file*; otherwise, make the next record from file *x* available to the program.'

```
Program 2(a)        initialise;
                    read (x);
                    while not end.of.file
                      do [process.record; read (x)];
                    finish;
Program 2(b)        initialise;
             again: read (x);
                    if not end.of.file
                      then [process.record; go to again];
                    finish;
```

(3) *read* (*x*), **at end** *s*; is defined: 'If the end of file *x* has been reached, then execute the statement-sequence *s*; otherwise, make the next record from file *x* available to the program and do not execute the statement-sequence *s*.'

```
Program 3(a)        initialise;
                    flag ← 0;
                    read (x), at end flag ← 1;
                    while flag = 0
                      do [process.record;
                        read (x), at end flag ← 1];
                    finish;
Program 3(b)        initialise;
             again: read (x), at end go to end;
                    process.record;
                    go to again;
               end: finish;
```

Reference

KNUTH, D. E. (1974). Structured programming with go to statements. *A.C.M. Computing Surveys*, Vol. 6, pp. 261-301.

Blind programmers—their manager's experience

D. L. Fisher

Director, Computer Laboratory, University of Leicester, Formerly Programming Manager, Computer Services Division, Bank of Scotland

This paper does not subscribe to the view that good braille facilities are necessarily of prime consideration in expanding the role of the blind programmer. Perhaps it is wiser to limit his dependence on braille by developing better access to media that can be shared with sighted colleagues.

Introduction

The May 1975 issue of *The Computer Journal* included a paper by P. W. F. Coleman entitled 'Integrated training for the blind programmer'. Coleman ended his paper by highlighting the problems of providing documentation in braille as if this was the right and proper way to provide blind programmers with the information they require. This paper challenges that supposition and offers instead the approach born of the experience of the Computer Services Division of the Bank of Scotland.

By the end of the sixties, the Bank's complement of approximately 40 programmers included two who were blind. One is quiet and restrained, and the other a rugged extravert. The extravert has been totally blind since early boyhood, whereas the other slowly went blind and is still able to detect light. Both are graduates, so that both had registered notable achievements, despite their handicap, prior to entering computing.

This is important because both are proud. Both are happy to accept that they are different from sighted people, perhaps at a disadvantage, but neither is prepared to accept that they are inferior to their sighted colleagues. Indeed they gain considerable satisfaction from being treated as their equals, although they can enjoy a little fuss when their lack of sight commands some special attention.

Education

The extravert was the first to be recruited and initially his computer education followed the conventional lines for a blind person. His companion was recruited about a year later and gained so much from the experience of the first, that most of his early education came by way of their man-to-man communication.

Their main problem was the lack of braille versions of the necessary manuals, and those they did hold were usually several years out of date by the time of acquisition. Fortunately the two did appear to compensate by making considerable gains from discussions and lectures they attended.

If one eliminated the conventional courses for sighted programmers, courses appropriate to their programming needs were almost impossible to find, so it was decided to turn to the conventional courses instead of eliminating them. Eventually it was decided to send the 'quiet' one on a conventional IBM closed-circuit TV COBOL course! He went with three other programmers from the installation, the idea being that they could spread the load of filling in the gaps for him whilst studying the course themselves.

The exercise was extremely successful, so much so that the 'robust' one was sent on a similar Assembler course, this time with two colleagues although three were originally planned. This particular exercise took a little longer to bear fruit. The material of the course was more difficult as is well recognised, but the blind programmer also had greater difficulty preparing himself for the course due to the lack of suitable braille reading material.

Nonetheless the approach was considered to be basically sound. Formal education was no longer considered to be a major problem, and when the installation moved towards audio-visual education given in-house, the two blind programmers were not considered to be seriously disadvantaged.

Lecturers performing live at the installation had to be persuaded not to use 'this' and 'that' but describe or name