A Case for Teaching Multi-exit Loops to Beginning Programmers

P. A. Buhr
Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

Abstract

While programming using the WHILE and REPEAT constructs, as in Pascal, is well established and taught regularly, programming using a LOOP construct with multiple exits is not. I feel that this much more general construct is a more useful programming construct and that it should be taught in introductory programming courses.

1 Introduction

Ever since the structured programming revolution in the early 70's and the elimination of the GOTO from the vocabulary of most programmers, programs have been written using the IF and the WHILE statements to specify control flow. While these control structures are complete in the sense that they allow any program to be written, I do not believe they provide satisfactory tools for either teaching or learning one of the fundamental concepts of programming: how to control the execution sequence in a program. This concept must be understood by students as the first stage in learning programming, and as with most learning experiences a student imprints on the first method he is shown and will follow that method regardless of the existence of better ways.

It is important to understand at the outset that I will be discussing how control structures are taught to beginning programmers, and how the types of control structures affect the ability to learn the art of computer programming. Hence, most of the material I am going to cover is not new, and should be considered as both a review and some of my own personal comments on the material. These comments arise from my experiences teaching beginning (as well as advanced) programmers using a variety of control structures in several different programming languages.

2 Background

Böhm and Jacopini [BJ66] demonstrated that any program written with GOTOs could be transformed into an equivalent program which uses only IF and WHILE control structures (actually only WHILE is necessary). The transformation requires the following modifications to the original GOTO program:

1. copying of fragments of the original program
2. introducing switches that will be used only to affect control flow (and, thus, do not contain data from the original program)

While these are necessary as part of the transformation from a GOTO program to a "structured" program, they are frequently required when programming with IF and WHILE even if the program is not initially written with GOTOs. The copying of code introduces problems such as keeping the copies consistent with one another when changes are made. Although this problem can be mitigated by the introduction of subprograms, using subprograms introduces additional declarations and executable statements (CALL's) into the program. Similarly, introducing switches requires extra declarations and executable statements to set, test, and reset the switches. In both these cases additional complexity is introduced into the program; this complexity being largely a consequence of the simple control structures in the programming language.

SIGPLAN Notices, vol. 20, no. 11, November 1985, pp. 14-22

It has been shown by [PKT73] that the switches required by the Böhm and Jacopini transformation can be avoided by using a new control structure. This control structure involves a LEAVE statement and requires that some of the loops in the program be labelled. The LEAVE statement exits all containing control structures until one is encountered that has the label specified in the LEAVE statement; for example, in:

```
A: LOOP
  . . .
  B: LOOP
    . . .
    LEAVE A
    . . .
    LEAVE B
    . . .
  END LOOP
  . . .
END LOOP
```

LEAVE A transfers control to the statement after (the END LOOP of) the loop labelled A and LEAVE B transfers control to the statement after the loop labelled B. LEAVE is essentially a GOTO, restricted in the following ways:

- It cannot be used to create a loop (i.e. cause a backward branch in the program). This means that only the looping construct can be used to create a loop. This is important to the programmer since all the situations that can result in repeated execution of statements in a program are clearly delineated.
- Since it always transfers out of containing control structures, it cannot be used to branch into a control structure.

While some languages do implement just such a control structure, one may identify three different capabilities here: one in which control leaves only the current loop control structure at possibly several different locations, called a **multi-exit loop**; one in which control leaves multiple levels but to a statically determinable level, called **static multi-level exit**; and one in which control leaves multiple levels but to a level that is determined dynamically, called **dynamic multi-level exit**. In the case of multi-exit loop, the level of exit is determined implicitly, and hence it is not necessary to name program levels nor to use that name to exit the loop. However, terminating multiple control structures either statically or dynamically requires such naming. I feel there is a need for a separate multi-exit loop and multi-level exit construct in a programming language, because multi-exit loops will occur more frequently than either type of multi-level exit, and absence of names reduces program complexity. As well, different constructs will likely be required for static and dynamic multi-level exits, as in Ada[1] [Uni83]; however, the Ada designers grouped multi-exit loop and static multi-level exit together into a single construct.

Programs written in languages without such control structures are often littered with logical variables that are used to determine flow of control at different times and stages during execution. It is my opinion that these switches are undesirable because they detract from readability, must be continually set and reset, and often must be tested repeatedly in a number of places. For most beginning programmers (and many professional programmers) programming using switches is in fact the norm. I feel that a switch used strictly for control flow purposes and not containing information about actual data being manipulated is the variable equivalent of the GOTO statement. This is because modification of it can be buried in a loop body or even appear in remote parts of the program.

In this discussion I will deal only with the multi-exit loop concept and not multi-level exits. This is because I want to discuss only concepts that are taught in an introductory programming course. If multi-

[1] Ada is a Registered Trademark of the U.S. Department of Defense

level exits were taught in an introductory course, they would be taught only after a good foundation in control flow had been laid. While it is true that both multi-exit loop and multi-level exits are necessary to get the greatest reduction in the number of logical variables needed when using just IF and WHILE constructs, it has been my experience that the problems that are encountered in an introductory programing course are not sufficiently complex as to require multi-level exits.

3 Comments on the Syntax of a Multi-Exit Loop

Since a multi-exit loop exits from the body of the loop, no conditions need to be specified at the beginning or end of the loop. I think the basic construct should be:

```
LOOP
  ...
END LOOP
```

Some languages have a loop termination facility (C-language [KR78], Pascal/VS [IBM81b], PL/1 [IBM81a] but require that the loop itself be created using one of the existing looping constructs. This may require specifying a condition at either loop beginning or end. Some examples are:

Pascal/VS	PL/1	C-language
WHILE TRUE DO	DO WHILE('1'B);	FOR(;;)

I do not find these acceptable because they are not aesthetically pleasing, and because they may require a knowledge of logical values.

The loop-exit itself should not be treated as a separate statement as in Ada, C-language, MODULA-2 [Wir85], and PL/1. In these languages, the exit statement must be placed in some conditional control structure to sensibly exit a loop, as in the Ada program:

```
LOOP
    ...
    IF ... THEN
        EXIT;
    END IF;
    ...
END LOOP;
```

The problem here is that students are particularly confused by an IF-statement that does not continue execution after the END IF; the usual flow of control associated with an IF-statement is not followed in this case because of the exit. This point was recognized in Ada which allows a conditional on the EXIT statement. However, this does not prevent a student from writing such a program nor the confusion that results from having to decide when to use an exit in an IF-statement or a conditional on an EXIT, as in:

```
LOOP              ?    LOOP
    ...           ?        ...
    IF ... THEN   ?        ...
        EXIT;     ?      EXIT WHEN ...;
    END IF;       ?        ...
    ...           ?        ...
END LOOP          ?    END LOOP
```

As well, a separate EXIT statement allows the exit to be buried in the body of a loop, thereby making it difficult to determine where and how many exits appear in the loop. This means that the entire body of the loop must be examined to find the situation(s) that cause loop exit, for example, the Ada program:

```
LOOP
    ...
    IF ... THEN
        ...
        IF ... THEN
            EXIT;            ←exit buried in the
        END IF;               body of the loop
       ...
    END IF;
    ...
END LOOP;
```

There is also the possibility that statements are put after the EXIT where they can never be reached, as in:

```
LOOP
    ...
    IF ... THEN
        EXIT;
        ...                  ←statements cannot be reached
    END IF;
    ...
END LOOP;
```

Therefore, I suggest a loop construct that includes exits as part of its definition as in Modula [Wir77], such as:

```
LOOP
    ...
  WHEN condition EXIT
    ...
END LOOP
```

The keyword WHEN emphasizes that the loop-exit clause performs a quite different function from the IF statement. As well, the WHEN clause forms part of the syntactic structure of the loop and therefore cannot be buried within other control structures; the following is syntactically incorrect:

```
LOOP
    ...
    IF ... THEN
        WHEN ... EXIT    ←cannot be contained in
    END IF                another control structure
    ...
END LOOP
```

A loop construct can have any number of WHEN clauses in its body.

While the loop-exit clause helps to identify exit points, they can be made most obvious by aligning them at the same level as the LOOP and END LOOP, for example:

```
LOOP
    ...
  WHEN ... EXIT
    ...
  WHEN ... EXIT
    ...
END LOOP
```

This allows the reader to find all loop-exit points easily. Their conditions can then be examined without having to examine the code that comprises the loop body. This sort of alignment is the same convention used for the ELSE clause of an IF-THEN-ELSE control structure. If the ELSE clause is not properly aligned the entire IF statement must be searched to locate the ELSE clause. This may seem like a simple point, but many programmers and certainly students would not do this without being told.

As well, the exit conditions should be allowed to include code that is pertinent to that exit, again as in Modula. This might be specified as:

```
LOOP
    ...
  WHEN ... THEN
        ...          ←second level of indentation
    EXIT              to emphasize the exit-code
    ...
END LOOP
```

Here the statements between the THEN and the EXIT are executed when the WHEN condition is true and before exit from the loop. This avoids having to duplicate the code to re-evaluate the conditions that caused one of several loop exits, so that appropriate action can occur for each. I prefer this form to the equivalent Zahn control structure [Zah74] for two reasons. First, I believe that an exit and its exit-code are closely related, and hence they should be located as close together as possible. Second, the Zahn construct requires the introduction of additional names which I believe increases program complexity. The equivalent program using a slightly modified version of the Zahn control structure is:

```
LOOP UNTIL exitname
    ...
    IF ... THEN
        exitname
    END IF
    ...
END LOOP
EXIT
    exitname => ...
END EXIT
```

The loop exit is specified by using the exit name anywhere in the body of the loop. When the exit name is reached during execution of the loop, control transfers to the corresponding exit name at the end of the loop, its exit-code is executed and control transfers to the statement after the END EXIT.

The advantage of the Zahn construct is when the same exit code is required for several exits, as a single exit name can appear in several locations in the loop body. To achieve the same thing, I would be forced to create a procedure to prevent duplication of code at each exit point; thus, ending up with a procedure call and a new name which is essentially the same as in the Zahn construct. However, by using a procedure, the procedure code is not associated with the loop body as in the Zahn construct. Fortunately, the situation where several exits have the same exit code does not occurs that frequently. And of those that do occur, many could have the exit conditions joined with the logical operator AND and execute the same exit-code. Hence, I have optimized my control structure to deal with the common situation in the simplest way, while still providing all the features of the Zahn construct.

When writing a loop that has only a single exit, code that should follow the loop can easily be put just before the EXIT and vice-versa. Thus, there is a slight problem in deciding exactly what code should be placed prior to the EXIT and what should follow the loop. Students must be told clearly that only actions that are directly related to the loop termination for that particular condition should appear in the loop-exit clause.

4 Primary Justification for the Multi-Exit Loop

Currently, most students learn to program using a conventional IF-WHILE programming language. In these programming languages, it is necessary to introduce logical variables and logical operators at an early stage in the teaching of a programming language. They are usually needed to create the compound conditions for loop termination that are necessary in many simple algorithms, for example, in the linear search:

```
found ← FALSE
i ← 0
WHILE i < size AND NOT found DO
    i +← 1
    found ← key = list[i]
END WHILE
IF found THEN
    . . .
```

The program must be written this way to avoid using an invalid subscript with variable list; this would happen if the list is full and the key is not in the list. Many students do not realize this point and great pain must be taken to explain it. As well, if the loop index is to be used after the search to reference other fields associated with the item found, care must be taken to not increment the index after the key is found.

While short-circuit AND/OR can simplify this program by eliminating the logical variable, as in:

```
i ← 1
WHILE i <= size AND key ~= list[i] DO
    i +← 1                    ↑
END WHILE                     executed only if i ≤ size
IF i <= size THEN
    . . .
```

it relies both on the compiler treating AND and OR in this manner and on an understanding of logical operators. The same linear search coded using a multi-exit loop looks like:

```
i ← 1
LOOP
  WHEN i > size EXIT
  WHEN key = list[i] EXIT
    i +← 1
END LOOP
IF i <= size THEN
    . . .
```

Here both the logical variable and logical operator have been eliminated and it is much clearer that the subscript problem will not occur. As well, after loop termination the loop index is pointing at the item in the list where the key was found. Since this construct eliminates the switches that are frequently needed in introductory programs, the multi-exit loop allows the discussion of logical variables and operators to be postponed to a much later time in the teaching of the programming language. Students can then learn and understand control structures and control flow before tackling logical variables and operators. As well, there is less chance that students will mistakenly think that logical switches are a fundamental component of looping and not an independent issue.

5 Secondary Justifications for the Multi-exit Loop

While my primary motivation for adopting a multiple exit loop was to reduce the need for logical switches and their manipulation, there are several other secondary benefits that result from using loop-exits. One is that copied code which arises from using only IF-WHILE programming constructs can be eliminated. There are many algorithms that are best written using an exit from the middle of a loop. If not written in this fashion, techniques such as the traditional "priming" of a loop must be used. For example, in reading a file:

```
read(a, b, c)                        LOOP
WHILE NOT eof DO                        read(a, b, c)
   { process a, b, & c }              WHEN eof EXIT
   read(a, b, c)                        { process a, b, & c }
END WHILE                            END LOOP
```

The example on the left shows the priming of a loop. This requires duplication of the READ statement or, in other cases, creation of a subprogram to eliminate the duplication, both of which are undesirable. As well, loop priming can be done an arbitrary distance before the beginning of the loop making the program difficult to read as the priming statement(s) is critical to the understanding of the loop.

Pascal [JW75] attempted to solve this particular problem by introducing a "trick" to eliminate priming reads. The first line of the file is read when the file is opened in order that a subsequent WHILE NOT eof DO loop can be used. Unfortunately, this "trick" has set back understanding of how to read a file for an entire generation of programmers. Situations such as reading from an interactive terminal cannot use this trick, and problems such as merging the contents of two files using end-of-file to control the loops does not work either. While Pascal GET and PUT mitigate these problems somewhat, their use relies on an understanding of pointer variables and file buffers. These concepts are usually not taught in an introductory course.

However, the reader may have noticed that both examples still have to rely on the (builtin) logical function eof. To eliminate it would require a dynamic multi-level exit and a named program level; so the logical function is not eliminated but merely replaced by a different construct. In any case, the program structure read(...) WHEN eof EXIT makes the action easily understood [SBE83].

The multi-exit loop allows the stopping condition to be stated in terms of why the loop should exit and not why the loop should continue. It has been my experience that students and most programmers talk about, document, and think about the situations that cause a loop to stop, not about the situations that cause the loop to continue. As a result, I think that this is a more natural way of expressing what the programmer wants. For example, when merging two files using high-values, the loop that performs the merge exits when both keys become high-value or conversely the loop executes while either key is not equal to high-value.

```
LOOP
  WHEN f1.key = high AND f2.key = high EXIT
    ...
```

(It is assumed by the time file merging is taught, logical variables and operators will also have been taught.) The student must translate this stopping condition into one used in a WHILE statement. Without a good understanding of de Morgan's law many students will not make this transformation properly for compound conditions. They will negate the relational operators but not the logical, producing:

```
WHILE f1.key ~= high AND f2.key ~= high DO
                      ↑
                      should be OR
```

Thus, the specification of loop termination by stopping conditions instead of continuation conditions produces programs that are easier for a student to write and understand.

The LOOP statement can replace several looping constructs as it subsumes both the WHILE and REPEAT. When a student has a choice between different control structures that perform very similar tasks, he is often not sure of the correct construct for a particular situation; the LOOP construct obviates this problem.

For the same reason, the iterative loop should not be a separate control structure, but simply a prefix for the LOOP statement as is done in Ada, for example:

```
LOOP i IN 1..size
    ...
END LOOP
```

This then allows the combined effect of iteration in conjunction with exits for controlling execution of the loop:

```
LOOP i IN 1..size
    ...
  WHEN ... EXIT
    ...
END LOOP
```

The combining of iteration and loop exits in the same construct also solves another significant problem in the use of loops. This problem arises when a variable of a discrete type is used to control a WHILE loop. Consider the following example:

```
CONST max = 10
TYPE index = 1..max
VAR  list : ARRAY[index] OF INTEGER
```

```
VAR  i : 0..max                      VAR i : 1..max+1
...                                  ...
found ← FALSE                        found ← FALSE
i ← 0                                i ← 1
WHILE i < max AND NOT found DO       WHILE i <= max AND NOT found DO
    i +← 1                               found ← key = list[i]
    found ← key = list[i]                i +← 1
END WHILE                            END WHILE
```

Note that in either case the range of values for the loop control variable must be larger than what is actually needed within the loop body. Thus, it is necessary to have not only the **index range** (1..10), but also an **extended range** (0..10 or 1..11), to precisely specify the proper set of values. By using an iterative loop with a loop-exit it is possible to specify precisely the desired range of values and, thus, avoid the extended range, as in:

```
LOOP i IN index
  WHEN key = list[i] EXIT
END LOOP
IF i <= max THEN
    ...
```

While I believe it is important to explain to students that the iterative prefix is doing something special in regard to the loop index, having the index range is far preferable to having to specify the extended range. Thus, for situations such as the linear search it is necessary to combine the iterative prefix with a loop-exit to allow the declaration of an index range for the type of the index variable.

Notice, that the last form of the linear search requires that the loop index not be local to the body of the loop or that it be copied so that it can be tested after the loop. As well, it is necessary to define precisely what the value of the loop index is after the loop has terminated so that it can be tested. These problems can be eliminated by allowing a pre-exit action for the iterative prefix. This is done by associating exit-code with the iterative prefix; it is executed if the loop terminates because the loop index takes on all values of the prefix range (i.e. 1..size), for example:

```
LOOP i IN 1..size THEN
        ...        ←iterative prefix exit-code is executed if i
    EXIT            takes on all values of the range 1..size
    ...
END LOOP
```

Notice that the iterative prefix exit specification (i.e. the prefix range) and the iterative prefix exit-code are as close together as possible. Thus, the final version of the linear search would be:

```
LOOP i IN index THEN
        ...        ←code to be executed if the item is not found
    EXIT
  WHEN key = list[i] THEN
        ...        ←code to be executed if the item is found
    EXIT
END LOOP
```

I feel it is better to have one looping construct in which all the meaningful combinations can be expressed, than several packaged combinations which cannot be mixed. Thus, the student can build up the loop construct from a set of primitives to solve a problem, rather than having to contrive the solution to fit inadequate packaged loop constructs.

The following example contrasts the looping construct I have suggested with the IF-WHILE control structures. Here an element is looked up in a list of items, if it is unique it is added to the end of the list, if it exists in the list its associated list counter is incremented.

```
found ← FALSE                          LOOP i IN 1..size THEN
i ← 0                                          size +← 1
WHILE i < size AND NOT found DO                list[size].count ← 1
    i +← 1                                     list[size].data ← key
    found ← key = list[i].data              EXIT
END WHILE                                 WHEN key = list[i].data THEN
IF found THEN                                  list[i].count +← 1
    list[i].count +← 1                      EXIT
ELSE                                   END LOOP
    size +← 1
    list[size].count ← 1
    list[size].data ← key
END IF
```

6 Conclusion

I hope this discussion will remind programming language designers that simple issues such as these are extremely important, as I have seen new languages that do not provide them. As well, I hope it will influence educators when they are evaluating a programming language for use in teaching beginning programmers. However, I must admit that there are few student compilers for programming languages that support multi-exit loops or multi-level exits in any form, so one is usually stuck between a rock and a hard place here. But even if a current teaching language does not provide multi-exit capabilities, that should not prevent the ideas of multi-exit loop and multi-level exits from being taught. They can certainly be presented in pseudo-code definitions of algorithms. At the very least these constructs can be simulated, and while this may be distasteful to many, it is a viable alternative to not presenting the ideas of multi-exits. I hope this discussion may encourage programmers who have never been exposed to this type of a control structure to re-think the way they are doing things, since there is a better way.

By using multi-exit loops programs are smaller, use fewer variables, and are potentially more efficient because switches do not have to be set, reset and tested. Such programs are also easier to understand and modify. As well, learning to program with such a control structure can be done without an immediate need for logical variables and logical operators. Hence, they can be taught independently from and at a later time than control structures. I strongly believe this aids the beginning student in both the speed of learning and comprehension of the fundamental concept of specifying control flow in a program.

7 Acknowledgements

I would like to thank Glen Ditchfield, Rob Holte, Peter King, and Bob Zarnke for taking the time to read and comment on the ideas presented here.

References

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, Turing Machines and Languages with only two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[IBM81a] International Business Machines. *OS and DOS PL/I Reference Manual*, first edition, September 1981. Manual GC26-3977-0.

[IBM81b] International Business Machines. *Pascal/VS Language Reference Manual*, first edition, 1981. Manual SH20-6168-1.

[JW75] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer–Verlag, first edition, 1975.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, first edition, 1978.

[PKT73] W. W. Peterson, T. Kasami, and N. Tokura. On the Capabilities of While, Repeat, and Exit Statements. *Communications of the ACM*, 16(8):503–512, August 1973.

[SBE83] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26(11):853–860, November 1983.

[Uni83] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.

[Wir77] Niklaus Wirth. Modula: a Language for Modular Multiprogramming. *Software—Practice and Experience*, 7(1):3–35, January–February 1977.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, third, corrected edition, 1985.

[Zah74] C. T. Zahn. Control Statement for Natural Top-down Structured Programming. In *Symposium on Programming Languages*, Paris, France, 1974.