

# Loop Exits and Structured Programming: Reopening the Debate

Eric S. Roberts  
Department of Computer Science  
Stanford University

## ABSTRACT

Internal exits from loops represent a critically important control structure that should be taught in the introductory CS1 curriculum. Without access to those facilities, students are often incapable of solving simple programming problems that occur frequently in applications. This paper reviews the existing evidence in support of such facilities and argues that it is important to reconsider our traditional pedagogical approach as we adopt new languages of instruction.

## 1. INTRODUCTION

Twenty-five years ago, a fierce debate raged within the computer science community over the issue of *structured programming*. At its essence, structured programming represents a disciplined approach to software development based on abstraction, stepwise refinement, and a formal approach to program correctness [Dahl72]. Unfortunately, as the debate progressed, attention was focused less on the general issues raised by structured programming than on a relatively small detail: whether or not it is ever appropriate to use the **goto** statement in a well-structured program. Launched by a letter from Edsger Dijkstra in 1968 [Dijkstra68], the **goto** controversy raged back and forth over the next several years, often resembling a religious war in emotional intensity. In the late 1970s, the debate died down, and a period of relative peace ensued.

The underlying controversy, however, has not gone away, and the question of which control structures are acceptable remains a source of passionate concern. The intensity of that concern has been demonstrated by some of the reactions I have received to the new CS1/CS2 curriculum at Stanford and to my textbook, *The Art and Science of C: A Library-Based Approach* [Roberts95], which reflects the curricular revisions we have undertaken at Stanford. When we converted the curriculum from Pascal to C [Roberts93], I decided—on the basis of many years of classroom experience—that we should move beyond the limited control structures available in Pascal and allow students to exit from the interior of a loop in the following two cases:

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGCSE '95 3/95 Nashville, TN USA

© 1995 ACM 0-89791-693-x/95/0003....\$3.50

1. When the structure of a loop requires some preparation (such as reading an input value) before making the test for termination, I encourage students to use the **break** statement to force explicit exit from the interior of a loop. This strategy, which is discussed in Section 3, allows students to solve what Dijkstra calls the *loop-and-a-half problem* in a way that appeals to the student's intuition and avoids duplication of code.
2. In the context of a function, I encourage students to use the **return** statement as soon as the value of the function becomes known, even if that **return** statement would force an early exit from a loop within the function. Allowing **return** to be used in this way makes it much easier for students to solve a variety of problems, as illustrated in Section 4.

Although the response to my text has been quite favorable, the decision to allow internal loop exits in these restricted cases has elicited some negative response. One reviewer, for example, wrote that using **break** "is akin to using the proverbial **goto**." Another charged that my approach violates the basic tenets of structured programming. A third declared that my use of **break** to exit from a **while** loop is simply "unacceptable," ruling out any further consideration.

The negative reactions expressed in such reviews underscore the continued existence of controversy over what control structures are acceptable for academic use. This paper summarizes the evidence supporting the use of internal loop exits and embedded **return** statements, and examines the dangers associated with restricting students to a more tightly constrained control model.

For many years, it was easy to discount the evidence in favor of internal loop exits. As long as Pascal was overwhelmingly accepted as the best language for teaching introductory programming, the issue had little practical relevance. In Pascal, internal loop exits are simply not available, and those who adhere to the standard version of the language are therefore forced to accept the control structures that Pascal provides.

In the last few years, however, many schools have abandoned Pascal for more modern languages. For the most part, the new languages—even those that follow directly in the Pascal tradition such as Modula-2 and Ada—include both a structured mechanism for exiting from the interior of a loop and an active **return** statement. The availability of these facilities gives new relevance to the underlying pedagogical question of how to teach control

structures at the introductory level. It is time to reopen the debate.

## 2. SUFFICIENCY VERSUS PRACTICAL UTILITY

Before exploring the accumulated evidence that supports the use of internal loop exits, it is important to acknowledge that the control structures provided by Pascal have a sound theoretical basis. In their 1966 paper, Böhm and Jacopini proved that it is possible to code any flowchart program using four control structures: atomic actions, sequential composition, **if-then-else**, and **while-do**. These structures became known as *D structures*, after Edsger Dijkstra, and were soon extended to form a larger set of control primitives called *D' structures* [Ledgard75] that also includes the **if-then**, **repeat-until**, and **case** statements, as they exist in Standard Pascal. Thus, the statements provided by Pascal indeed constitute a sufficient set of control primitives, in the theoretical sense.

The key point, however, is that theoretical sufficiency is not in itself the principal criterion for program language design. After all, the arithmetic **if** and **goto** statements provided by Fortran II also constitute a sufficient set by this definition, but no one would seriously advocate using those statements as the basis for control structures in this day and age. The discipline encouraged by the use of *D structures* leads empirically to more reliable and more easily maintained programs than Fortran's primitives support. Thus, although the evolution of *D structures* represented an important practical advance for programming language design, theoretical sufficiency was not the primary reason for its success. If it were, language designers would have seen no need to augment the set of *D structures* with the extremely useful *D'* forms.

A similar illustration can be drawn from the standard set of Boolean operators. Most languages define **and**, **or**, and **not** as the primitive operations on Boolean data. If one's primary concern is linguistic economy, this set could easily be considered wasteful, because the single operator **nand** (or, equivalently, the operator **nor**) would be sufficient in theoretical terms. This theoretical result is critical for hardware designers, who are able to exploit the sufficiency of a single operator to achieve greater regularity in the design of integrated circuits. For programmers, on the other hand, the fact that **nand** is by itself sufficient has little practical relevance. The average applications programmer has no intuition regarding the behavior of **nand** and therefore would not know how to use it effectively. Programming problems tend to be phrased in terms of the traditional logical connectives, and it is for this reason—the admittedly subjective criterion of “naturalness” in the sense of being consistent with intuition—that **and**, **or**, and **not** form the basis of Boolean calculation.

The question of what set of control structures is appropriate is therefore larger than the question of whether a particular set of primitives is sufficient. It is also important to consider whether the control structures are suitable for the problems that tend to arise in practical programming, or, perhaps more importantly, whether the structures suit the ways in which programmers conceive of those problems. If they do not, and if empirical evidence

indicates that transforming a conceptual image of a problem into an approved control framework tends to introduce error or unnecessary complexity, it is important to examine that framework to determine whether extensions are required.

## 3. THE LOOP-AND-A-HALF PROBLEM

The fact that some loop structures seem to have a natural exit in the middle has long been recognized. In his 1974 comments on the **goto** controversy [Knuth74], Don Knuth outlined his own perspective, as follows:

The iteration statements most often proposed as alternatives to **goto** statements have been “**while B do S**” and “**repeat S until B**”. However, in practice the iterations I encounter very often have the form

```
A: S;
   if B then goto Z fi;
   T;
   goto A
Z:
```

where **S** and **T** both represent reasonably long sequences of code. If **S** is empty, we have a **while** loop, and if **T** is empty we have a **repeat** loop, but in the general case it is a nuisance to avoid the **goto** statements.

A typical example of such an iteration occurs when **S** is the code to acquire or generate a new piece of data, **B** is the test for end of data, and **T** is the processing of that data.

Loops of this sort, in which some processing must precede the test, come up quite often in programming and constitute what Dijkstra has called the *loop-and-a-half problem*. The canonical example of the loop-and-a-half problem is precisely the one that Knuth describes: reading a set of input values until some sentinel value appears. If the basic strategy is expressed in pseudocode, the problem has the following solution:

```
loop
  read in a value;
  if value = Sentinel then exit;
  process the value
endloop;
```

In this example, the **loop/exit/endloop** statement is an extension to the Pascal repertoire and represents a loop that is terminated only by the execution of the **exit** statement in the interior of the loop body. The **loop/exit/endloop** extension makes it possible to express the algorithm so that the input value is read in from the user, tested against the sentinel, and then processed as necessary.

The **loop/exit/endloop** approach, however, offends the sensibilities of many computer science instructors, who have learned through their experience with Pascal to “unwind” the loop so that the test appears at the top of the **while** loop, as follows:

```
read in a value;
while value != Sentinel do begin
  process the value;
  read in a value
end;
```

Unfortunately, this approach has two serious drawbacks. First, it requires two copies of the statements required to read the input value. Duplication of code presents a serious maintenance problem, because subsequent edits to one set of statements may not always be made to the other. The second drawback is that the order of operations in the loop is not the one that most people expect. In any English explanation of the solution strategy, the first step is to read in a number, and the second is to process it by adding it to the total. The traditional Pascal approach reverses the order of the statements within the loop and turns a *read/process* strategy into a *process/read* strategy. As the rest of this section indicates, there is strong evidence to suggest that the *read/process* strategy is more intuitive, in the sense that it corresponds more closely to the cognitive structures programmers use to solve such problems.

The problem of code duplication is in some sense objective. The statements that read in an input value do in fact appear twice in the program, and it is easy to imagine that an edit in one copy might not be made symmetrically in the other. The second problem, however, is inherently more subjective because it hinges on the question of which loop order is more “intuitive.” What is intuitive for some may not be intuitive for others, and it may be that a preference for one strategy over another represents nothing more than an individual bias. For that reason, it is important to rely on empirical measurement of the success that students have in using the different strategies.

The best known study of this form was conducted by Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich at Yale University in the early 1980s. In their experiments, student programmers with varying levels of experience were divided into experimental and control groups. The students in each group were then asked to write a program to average a list of integers terminated by a sentinel value. Those in the control group were instructed to solve the problem using Standard Pascal. Those in the experimental group were asked to use a variant of Pascal (Pascal L) in which the standard looping statements **while** and **repeat** were replaced by a new **loop** construct having the following form:

```

loop
  statements
  if condition leave;
  statements
again

```

The **loop** statement allows for a structured exit from the interior of the loop and therefore permits the use of a *read/process* strategy. The results of the experiment appear in the abstract of the paper published in *Communications of the ACM* [Soloway83]:

Subjects overwhelmingly preferred a *read/process* strategy over a *process/read* strategy. When writing a simple looping program, those using the **loop** . . . **leave** . . . **again** construct were more often correct than were those using the standard Pascal loop constructs.

The correctness argument is particularly compelling. At all levels of experience, students using Pascal L were more

likely to write correct programs than were their counterparts in the control group. The outcome is all the more significant because the students in the experimental group had little chance to become familiar with the new loop construct, as indicated by the following passage from the paper:

Given that students were exposed to the **loop** . . . **leave** . . . **again** construct of Pascal for only a few minutes, and given that they had much more familiarity and experience with Pascal’s standard loop constructs, we were quite impressed with the high performance of the Pascal L users. Thus, these data support the claim that people will write correct programs more often if they use the language that facilitates their preferred strategy.

I have looked for other studies that might contradict the findings of the Soloway report. Although I have found authors who assert that the use of internal loop exits is wrong, I have encountered none that support their claims with objective evidence.

It is also possible to use standard Pascal texts as evidence that the *read/process* strategy in fact corresponds to student intuition. An examination of at the approach used to solve the read-until-sentinel problem in the five best-selling Pascal textbooks [Cooper92, Dale92, Koffman91, Leestma87, Savitch91] reveals that:

- In each of the textbooks, the problem is solved by unwinding the loop so that the read phase of the operation occurs twice: once prior to the loop and once at the end of the loop body.
- In every text except Leestma and Nyhoff, the authors derive their solution by first presenting the *read/process* strategy and subsequently modifying it to fit the *process/read* form required by Pascal’s control structures.

To me, the approach these authors take suggests that they themselves accept that students find the *read/process* strategy more intuitive. To encourage students to write programs that fit Pascal’s restrictions, these texts teach them—often at great length—to abandon their initial strategy in favor of one that reverses the order of the operations within the loop. My own classroom experience certainly supports this conclusion. By introducing loop exits and using the *read/process* strategy, I find that I can save almost an entire day’s lecture.

The concern, of course, about introducing an internal loop exit form is that students will end up abusing it, since it is certainly possible to do so. Our experience at Stanford, however, suggests that this problem does not in fact arise. Students tend to write code based on the models they are given, using internal loop exits only in the disciplined way outlined in the class examples. In the last three years, over 2500 students have completed CS1 at Stanford, and the teaching assistants have reported no problems with overuse of the **break** construct. This finding is consistent with the results of a study by Sheppard, which demonstrated that internal loop exits do not in fact compromise program readability [Sheppard79].

#### 4. RETURNING FROM INSIDE A LOOP

In addition to encouraging the use of an internal loop exit to solve the loop-and-a-half problem, I also teach students that it is appropriate to use a **return** statement at any point in a function where the result value becomes known, even if the **return** statement is nested in such a way that it acts as a nonstandard exit from an enclosing control structure. The paradigmatic example that underscores the value of this technique is the *sequential search problem*.

In essence, the sequential search problem consists of writing an implementation for the following function header line:

```
function Search(var list: IntList;
               n, key: integer) : integer;
```

The parameter **list** is an array of integers that conforms to the Pascal type definition

```
type
  IntList = array [1..MaxList] of integer;
```

where **MaxList** is some constant upper bound on the list size. The actual number of elements actively stored in the array is typically smaller and is given by the parameter value **n**. The problem is to implement **Search** in such a way that it returns the least index **i** for which **list[i]** is equal to **key**; if the value **key** does not appear in **list**, the function **Search** should return 0. Functions that have this basic structure come up frequently in programming, even at the introductory level.

If the language includes an active **return** statement, such as the one provided by C, Modula-2, or Ada, the **Search** function has the following concise and natural implementation:

```
function Search(var list: IntList;
               n, key: integer) : integer;
var
  i: integer;
begin
  for i := 1 to n do
    if list[i] = key then return i;
  return 0
end;
```

If students are restricted to the facilities available in Standard Pascal, the **Search** function, as simple and as fundamental as it is, turns out to be extremely hard to code—so difficult in fact that many students are incapable of developing a correct solution. If you haven't encountered this problem, it is worth spending a few minutes to write the code on your own.

The difficulty of coding the sequential search algorithm was described in 1980 in an insightful paper by Henry Shapiro that never received the attention it deserved. The statistics he reports are far more conclusive than those in the Soloway study of internal loop exits. Of the students in his study who attempted to use the **for** loop strategy, all of them managed to solve the problem correctly, even though they were forced to use a **goto** statement to achieve the effect of the **return** statement in the preceding example.

In fact, Shapiro notes that

I have yet to find a single person who attempted a program using [this style] who produced an incorrect solution.

Students who attempted to solve the problem without using an explicit return from the **for** loop fared much less well: only seven of the 42 students attempting this strategy managed to generate correct solutions. That figure represents a success rate of less than 20% [Shapiro80].

That students would have difficulty in developing a general solution to this problem is not really surprising. When they are limited to Pascal's control structures, the answer is quite complicated. To see just how complicated the solutions can get, it is again useful to consider how the leading Pascal texts handle the problem.

The following code, adapted from the solution given on page 300 of Leestma and Nyhoff [Leestma87], illustrates that a correct solution is indeed possible:

```
function Search(var list: IntList;
               n, key: integer) : integer;
var
  i: integer;
  found: boolean;
begin
  i := 1;
  found := FALSE;
  while (i <= n) and not found do
    if list[i] = key then
      found := TRUE
    else
      i := i + 1;
  if found then
    Search := i
  else
    Search := 0
end;
```

This solution introduces a Boolean variable and uses 11 lines of code to do the work accomplished by three in the solution based on the **return** statement. It does, however, return the correct result. The Dale and Weems text [Dale92] uses a slight variation of this strategy that also returns the correct result.

The other leading Pascal texts adopt a different approach. The code suggested independently by Cooper, Koffman, and Savitch [Cooper92, Koffman91, Savitch91] has the following general form:

```
function Search(var list: IntList;
               n, key: integer) : integer;
var
  i: integer;
begin
  i := 1;
  while (i < n) and (list[i] <> key) do
    i := i + 1;
  if list[i] = key then
    Search := i
  else
    Search := 0
end;
```

These authors have taken great pains to avoid some of Pascal's pitfalls. The bounds test for the **while** loop specifies continuation only if  $i < n$ , even though the test  $i \leq n$  might seem more natural in this context. The problem with the test  $i \leq n$  is that Pascal's handling of the **and** operator can generate a subscript violation if  $n$  is equal to **MaxList**. On the last cycle, Pascal evaluates both halves of the condition and therefore evaluates **list[i]** even if it has already determined that  $i > n$ . Fixing this problem also requires that the test in the **if** statement check the element value and not the value of the index. Unfortunately, this change in the final test introduces a new problem: the function can fail if  $n$  is 0. As written, the implementation tests the first element in the array against the key even if the array has no active elements.

In fairness to the authors, it is important to point out that the original examples are not in fact buggy in the strictest sense. Koffman specifically rules out the possibility that  $n$  is 0 in the preconditions of the function, although this relationship is not tested within the implementation. In both Cooper and Savitch, the upper bound is a constant rather than a variable. Since the constant is not declared to be zero, the problem does not actually arise. Even so, the important point is that the coding from these texts cannot be considered as an appropriate model for a general-purpose search function. A student trying to solve the general case might well do exactly what I did: go through the functions and replace the constant bound with the corresponding variable. At that point, the implementation fails in the  $n = 0$  case.

The conclusion that the sequential search problem is difficult for students who use the Pascal strategy is strongly supported by our experience at Stanford. When we taught the introductory course in Pascal, we worked hard to present the sequential search algorithm in a careful and rigorous way that would ensure correct operation in special-case situations. Despite that care, we discovered that even our best students had trouble regenerating the solution after taking the course. The situation is entirely different now that we use a **return** statement to implement the sequential search algorithm. Students who manage to understand arrays at all understand the sequential search algorithm immediately; it no longer represents a source of difficulty.

## 5. CONCLUSIONS

When students are limited to the control structures available in Pascal, they tend to have significant difficulty solving certain programming problems that come up frequently in practical applications. Allowing students to use control forms that provide a structured mechanism for loop exit increases their comprehension along with their ability to write correctly functioning code. Although these facilities are absent from Pascal, they are widely available in modern programming languages and should be introduced when those languages are used in an instructional setting.

To a significant extent, the discipline that Pascal encouraged has had a positive effect on the computer science curriculum. As we move to new programming languages, however, it is crucial to analyze our experience to determine what aspects of traditional practice are

predicated more on the particular nature of Pascal than on some more general principle. If our programming methodology can be defended on its own merits, we should continue to support it. On the other hand, if our students are unable to write correct code using the existing paradigms, we must reevaluate them. As Dijkstra observed, "the tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities" [Dijkstra82]. As we adopt new tools, we must be ready to think in new ways.

## REFERENCES

- [Böhm66] C. Böhm and G. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," *Communications of the ACM*, May 1966.
- [Cooper92] Doug Cooper, *Oh! Pascal! Turbo Pascal 6.0* (third edition), New York: Norton, 1992.
- [Dahl72] Ole Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, London: Academic Press, 1972.
- [Dale92] Nell Dale and Chip Weems, *Turbo Pascal* (third edition), Lexington, MA: D.C. Heath, 1992.
- [Dijkstra68] Edsger W. Dijkstra, "Goto statement considered harmful," letter to the editor, *Communications of the ACM*, March 1968.
- [Dijkstra82] Edsger W. Dijkstra, "How do we tell truths that might hurt," *SIGPLAN Notices*, May 1982.
- [Knuth74] Donald Knuth, "Structured programming with goto statements," *Computing Surveys*, December 1974.
- [Koffman91] Elliot Koffman with Bruce Maxim, *Turbo Pascal* (third edition), Reading, MA: Addison-Wesley, 1991.
- [Ledgard75] Henry Ledgard and Michael Marcotty, "A genealogy of control structures," *Communications of the ACM*, November 1975.
- [Leestma87] Sanford Leestma and Larry Nyhoff, *Pascal: Programming and Problem Solving* (second edition), New York: Macmillan, 1987.
- [Roberts93] Eric S. Roberts, "Using C in CS1: Evaluating the Stanford experience," *SIGCSE Bulletin*, March 1993.
- [Roberts95] Eric S. Roberts, *The Art and Science of C: A Library-Based Approach*, Reading, MA: Addison-Wesley, 1995.
- [Savitch91] Walter Savitch, *Pascal: An Introduction to the Art and Science of Programming* (third edition), Redwood City, CA: Benjamin/Cummings, 1991.
- [Shapiro80] Henry Shapiro, "The results of an informal study to evaluate the effectiveness of teaching structured programming," *SIGCSE Bulletin*, December 1980.
- [Sheppard79] S. Sheppard, B. Curtis, P. Millman, and J. Clement, "Modern coding practices and programmer performance," *Computer*, December 1979.
- [Soloway83] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich, "Cognitive strategies and looping constructs: an empirical study," *Communications of the ACM*, Vol. 26, No. 11, November 1983.
- [Yuen94] C. K. Yuen, "Programming the premature loop exit: from functional to navigational," *SIGPLAN Notices*, March 1994.