The Use of Termination Indicators in Computer Programming

M. C. ER
Department of Computer Science, University of Western Australia, Nedlands, WA 6009, Australia

A simple programming technique called the* termination-indicator technique *is introduced. The basic idea is to use variables as termination indicators in multi-exit loops in order to simplify termination conditions of loops. Numerous examples are given illustrating elegant, efficient and structured solutions that benefit from this technique. Comparative results show that the termination-indicator technique is superior and more versatile than other programming techniques using cand/cor, sentinel and state-variables.

Received August 1984

1. INTRODUCTION

In a series of letters to the editor,[1-6] various authors quibbled about a simple programming problem which was first reported by Knuth[7] and subsequently cited by Arblaster, Sime and Green.[8] The programming problem is to do with searching and updating of arrays: given an array $a[1\,..\,N]$ with distinct values stored in $a[1\,..\,m]$, where $m \leqslant N$, the purpose is to record the number of times in an array $b[1\,..\,N]$ those distinct values have been looked up in array $a[1\,..\,m]$. If x is already present in $a[i]$, $1 \leqslant i \leqslant m$, the value $b[i]$ will be incremented by 1; if x is not found in $a[1\,..\,m]$, x will be stored in $a[m+1]$ provided $m < N$, and $b[m+1]$ is assigned a 1 with the search range $1\,..\,m$ being increased by 1. Atkinson[9] summarises the situations and presents his own solution to this searching-and-updating problem using state-variables.

What comes as a surprise is not the ignorance of some authors[2,3] who did not read Knuth's paper[7] and thus repeated what he had said, but rather the fact that their programs[2,3] did not work for some extreme cases, such as empty or full array, and the clumsiness of some solutions.[1,9] Atkinson's summary and discussion seem to have exhausted all possible programming solutions to this simple problem of searching and updating. But surprisingly, there is still a simple and efficient programming solution to this problem, which has not been discovered by others at all. In this paper we discuss such a solution, and we call the programming technique we shall use the *termination-indicator technique*, as it has something to do with the use of termination indicators.

2. THE TERMINATION-INDICATOR TECHNIQUE

The discussions in a series of letters[1-6] leading to Ref. 9 show that, in order to avoid the use of goto-statement, one is forced to choose one of the structured solutions using cand/cor, sentinel, or state-variable, each of which has its own limitations. We now introduce the termination-indicator technique which has none of the limitations, and the resulting algorithms are considerably cleaner and more transparent.

The central idea of the termination-indicator technique is to use auxiliary variables to serve as termination indicators. These termination indicators are normally initialised to values that will yield the worst-case performance of algorithms. During the course of execution of a program, termination indicators may be set to places where one of the termination conditions of loops is satisfied. In this way, the condition that terminates a multi-exit loop can be retained in the associated termination indicators. This argument applies to both single and nested loops.

An application of the termination-indicator technique to the searching-and-updating problem is shown in Fig. 1.

```
i: = 1;
t: = m+1; {t is a termination indicator}
while i ≠ t do
  if a[i] ≠ x then i: = i+1 else t: = i;
if t ⩽ m then b[t]: = b[t]+1
  else if t ⩽ N then begin
         m: = t; a[t]: = x; b[t]: = 1
         end
  else error
```

Figure 1. An efficient algorithm for the searching-and-updating problem using a termination indicator *t*.

In this algorithm, the termination indicator t is initialised to $m+1$ to provide the worst-case condition of searching the array when x is absent from $a[1\,..\,m]$. During the search, t is positioned at the element containing x if it exists; otherwise t is left intact. Thus from the position where t points to, we can tell whether x is present in $a[1\,..\,m]$, or x is not found in $a[1\,..\,m]$ but this segment can be extended, or the array is full. Unlike other solutions[9] that specifically test for array full, our solution detects array full as a natural by-product. Note further that our algorithm handles both empty and full arrays correctly. Regarding the efficiency of our algorithm, it makes two tests per cycle of loop, and runs at a comparable speed as the algorithm using cand/cor, but does not depend on the special facility of cand/cor. Also it does not depend on the use of sentinel and hence the full capacity of array $a[1\,..\,N]$ can be utilised. The code is compact and transparent and is definitely shorter than Atkinson's solution.

3. LINEAR SEARCH OF A LINKED LIST

Suppose the array $a[1\,..\,N]$ of the searching-and-updating problem is replaced by a linked list. With the following declarations:

```
ptrperson = ↑person;
person = record
           ss:integer;
           next:ptrperson
           end;
var p, first:ptrperson;
```

a linear search algorithm for searching a linked list may be given as shown in Fig. 2.

```
p:=first;
while (p<>nil) and (p↑.ss<>x) do p:=p↑.next
```

Figure 2. A linear search algorithm for a linked list.

Realising that the *and* operator may not be implemented as the *cand* operator in some Pascal compilers, Jensen and Wirth[10] suggest two alternative solutions as shown in Fig. 3.

```
p:=first; b:=true;
while (p<>nil) and b do
  if p↑.ss=x then b:=false else p:=p↑.next;
if b then {x absent} else {x found}
```

(*a*)

```
p:=first;
while p<>nil do
  begin
    if p↑.ss=x then goto 13;
    p:=p↑.next
  end
```

(*b*)

Figure 3. Two alternative solutions that avoid the use of cand in a linear search algorithm for a linked list.

These two alternative solutions are not as elegant as one would wish. It turns out that the termination-indicator technique can help in this linked-list problem as shown in Fig. 4.

```
p:=first;
t:=nil; {termination indicator}
while p≠t do
  if p↑.ss≠x then p:=p↑.next else t:=p;
if t=nil then {x absent} else {x found}
```

Figure 4. A linear search algorithm for a linked list using a termination indicator *t*.

The resulting algorithm using a termination indicator t is more pleasing than Jensen and Wirth's solutions and also more efficient than the algorithm shown in Fig. 3(*a*).

4. SEARCHING AND UPDATING A HASH TABLE

If the searching-and-updating problem is modified so that the array $a[1\,..\,m]$ is organised as a hash table, Knuth[7] presents a hashing-and-linear-reprobing algorithm using a goto-statement as shown in Fig. 5.

```
i:=h(x); {hashing}
while a[i]≠0 do
  begin
    if a[i]=x then goto found fi;
    i:=i-1;
    if i=0 then i:=m fi
  end
notfound: a[i]:=x;
          b[i]:=0;
found:    b[i]:=b[i]+1;
```

Figure 5. Knuth's hashing-and-linear-reprobing algorithm.

To rewrite this algorithm as a structured program, the sentinel technique does not work any more since there is no place to store a sentinel element. Anyhow, Knuth[7] comes out with two improved solutions, both requiring the use of goto-statements. One of his solutions is shown in Fig. 6.

```
i:=h(x);
while a[i]≠x do
  begin
    if a[i]=0 then
      a[i]:=x;
      b[i]:=0;
      goto found
    fi;
    i:=i-1;
    if i=0 then i:=m fi
  end;
found:b[i]:=b[i]+1
```

Figure 6. Knuth's improved version of the hashing-and-linear-reprobing algorithm.

His solutions do not take hash-table full into consideration, and will be trapped in non-terminating loops when this odd event occurs. To take this factor into account, each of his solutions perhaps has to maintain a counter, keeping track of the number of entries of the hash table that are occupied.

Applying the termination-indicator technique to derive a hashing-and-linear-reprobing algorithm, we come out with a simple solution as shown in Fig. 7. Our algorithm handles the case of hash-table full correctly and need not specifically test for the occurrence of this odd event.

```
i:=h(x);
t:=i mod m+1; {termination indicator}
while i≠t do
  if a[i]=x or a[i]=0
    then t:=i
    else begin
      i:=i-1;
      if i=0 then i:=m
    end;
if a[t]=x
  then b[t]:=b[t]+1
  else if a[t]=0
    then begin
      a[t]:=x;
      b[t]:=1
    end else {hash-table full}
```

Figure 7. A hashing-and-linear-reprobing algorithm using the termination-indicator technique.

5. INSERTION SORT

A sorting algorithm that utilises the linear search as a subprocess is the insertion sort. The termination-indicator technique can also be applied to sorting by insertion as shown in Fig. 8. Here we assume that an array $a[1..N]$ is to be sorted into an ascending order.

```
i: = 2;
while i ⩽ N do begin
  j: = i; x: = a[i];
  t: = 1; {termination indicator}
  while j ≠ t do
      if a[j−1] ⩽ x
        then t: = j
        else begin
          a[j]: = a[j−1];
          j: = j−1
        end;
  a[t]: = x;
  i: = i+1
  end
```

Figure 8. Sorting by insertion using a termination indicator.

Comparing with Dromey's insertion sort algorithm,[11] we see that his algorithm requires one additional pass during a pre-sort phase to find the smallest element in the array and assign it to $a[1]$ in order to prevent the subscript from being out of bound during the sort phase. In contrast, our algorithm requires no such pre-sort phase at all.

6. BUBBLE SORT

As a way of illustrating the flexibility of the termination-indicator technique, we further apply it to the bubble sort. It is well known that if there is no exchange of elements taking place during a pass of the bubble sort, subsequent passes will not move any element of an array of all, and indeed the bubble sort can be terminated at this stage. The use of a state-variable *sorted* to detect any exchanges taking place during a pass as seen in Ref. 11 is a well-known trick. But the problem is that this state-variable must be tested, together with other conditions, every time round the loop in order to exit from the loop when the termination condition is met. With the use of termination variable, such redundant test can be avoided as shown in Fig. 9. We assume that the array $a[0..N-1]$ is to be sorted into an ascending order.

We may interpret the manipulation of termination indicator t as follows: it is initialised to a 1 as the worst-case condition to begin with; during the $(N-i+1)$th pass, we hope that this would be the last pass of the bubble sort by assigning the value of i to t; but as soon as we find a pair of adjacent elements that are out of sequence, we reset t to the worst-case condition. As seen in Fig. 9, our solution performs less tests than other solutions.[11]

```
i: = N;
t: = 1; {termination indicator}
while i > t do begin
  i: = i−1; j: = 0;
  t: = i;
  while j ≠ i do begin
    if a[j] > a[j+1] then begin
      swap (a[j], a[j+1]);
      t: = 1
      end;
    j: = j+1
    end
  end
```

Figure 9. An application of the termination-indicator technique to the bubble sort.

7. CONCLUSIONS

In this paper we have described at length the termination-indicator technique and its applications. The termination-indicator technique stresses the use of termination indicators and is most applicable to loops or nested loops that have multi-exit conditions. As the history of computer programming shows, the problem of terminations of multi-exit loops has proved to be a mental stumbling block; many experienced programmers[1-3, 7, 9-12] simply could not come out with simple, efficient and structured solutions. Some authors[7] even used it as a lever to argue against gotoless programming! Fortunately, numerous programming examples in this paper show that the problem can be trivially solved by using the termination-indicator technique. Most surprisingly, termination conditions of all solutions are extremely pleasing – normally consisting of a comparison of a variable with a termination indicator. Perhaps more importantly, this programming technique is language-independent, as well as compiler-independent, and works in many different contexts.

When a problem is tricky, people often look for new programming constructs to accommodate specific circumstances,[7] but overlook the simplest programming solution. A study of programming techniques is not less important than an invention of programming constructs. Surely, a structured solution afforded by a simple programming technique is better than that cast in a complex programming construct.

Acknowledgement

The author wishes to thank the referee for his helpful comments, which improved the contents and presentation of this paper.

REFERENCES

1. I. D. Hill, Jumping to some purpose (letter). *The Computer Journal* **23**, 94 (1980).
2. G. L. Robinson, Jumping to some purpose (letter). *The Computer Journal* **23**, 288 (1980).
3. M. Missala and P. Rudnicki, Jumping to some purpose (letter). *The Computer Journal* **25**, 286 (1982).
4. J. Inglis, Jumping to some purpose (letter). *The Computer Journal* **25**, 495 (1982).
5. G. L. Robinson, Jumping to some purpose (letter). *The Computer Journal* **26**, 95 (1983).
6. I. Mearns, Jumping to some purpose (letter). *The Computer Journal* **26**, 190 (1983).

7. D. E. Knuth, Structured programming with goto statements. *Computing Surveys* **6**, 261–301 (1974).
8. A. T. Arblaster, M. E. Sime and T. R. G. Green, Jumping to some purpose. *The Computer Journal* **22**, 105–109 (1979).
9. L. V. Atkinson, Jumping about and getting into a state. *The Computer Journal* **27**, 42–46 (1984).
10. K. Jensen and N. Wirth, *Pascal – User Manual and Report*. Springer-Verlag, New York (1978).
11. R. G. Dromey, *How to Solve it by Computer*. Prentice-Hall, London (1982).
12. C. K. Yuen, Further comments on the premature loop exit problem. *SIGPLAN Notices* **19**, 93–94 (1984).