acm forum

PCs and CPs

This letter is in response to the February 1987 President's Letter in *Communications* ("Personal Computers and Computing Professionals," pp. 101–102). Right on and write on, Paul Abrahams. Last summer (1986), ACM-SIGGRAPH awarded me an educational resource grant to assist with the creation of computer art workshops for high school and middle school children. Since attending SIGGRAPH '86, I have gone into the classrooms of our children. These young people know a great deal about personal computers, video technology, computer music, . . . the electronic world. Their heroes, in some cases, are the hackers and computer wizards to whom Abrahams refers in his letter. I have been able to reach young people and have received the support of Parent Teacher Associations (PTAs) for my use of personal computers in the creation of computer art. SIGGRAPH Video Reviews are exciting for children to watch, but the opportunity to see and do creative work on an Apple II, Amiga or Macintosh goes a long way in educating children. As a result, it seems like a great idea for ACM to find a place for the hardware tinkers and software wizards who have made such a wonderful contribution to the development of young people.

Theresa-Marie Rhyne
Computer Artist/Art Educator
P.O. Box 3446
Stanford, California 94305

View from Watergate Bridge

The Forum strives for balanced presentation. One way to achieve this is by soliciting responses to received letters. Another is to publish all or a representative sampling of subsequent reader responses to letters. The former expedient was followed for the letter from Herb Grosch, to which the following response refers. The latter expedient is adopted here, the "balance" being perhaps skewed by the fact that this was the only response received. The editor accepts full responsibility for delaying its publication somewhat until it seemed reasonably certain that no more responses were forthcoming.
—R. L. Ashenhurst.

While reading Herb Grosch's letter in a recent ACM forum ("An ACM Watergate," *Communications*, Oct. 1986, p. 928–930), I was reminded of an old Dutch expression that my late father used for this sort of situations: "Vechten tegen de bierkaai," he used to say. It meant that no matter how hard one fought and argued and obtained agreements, the thing would crop up again and again. It was a fight without an end. And that is what the ACM has become.

For those of us who have been convinced of the necessity of Chapters and have been fighting for twenty years now for Chapter Rights and to make life more bearable for the common programmer, Herb is the only visible and audible voice left, it seems. Most of us gave up after the Council elections of 1982 and stopped paying dues. I still pay my dues every year and will for as long as Herb is on the Council. Unless they kick me out once this piece is published.

The publications boys in New York have tricks up the kazoo in order to protect their jobs. It has happened to me and to others that a piece is put "on hold" for publication until the establishment has thought of enough smart answers for publishing the piece with their comments. But the original author does not see their comments until he reads them in *Communications.* And if he then tries to get a rebuttal published, it is refused "because there is no sense in dragging it out," as I was once told after inquiring. We now read that the same thing again has befallen Herb Grosch. It's the secrecy that gets ye! They only do what they are legally obligated to and not what is morally right. I know: that is hard to prove, and they probably will scream of slander and libel and threaten legal action because their usual response is to hide behind the law and the rules of the Association. It's the way that the staff interprets figures and doctors up reports, holding the interesting stuff close to their chests and publishing good-to-them items only.

Slowly it's becoming impossible to say anything or ask questions anymore. Over the years the staff and the Council have become sacred and we, the rank-and-file members, we are the sacred jackasses who have let them become that holy in the first place.

It may be true that the total number of members is at an all time high, as Adele Goldberg states. And as long as the sign-up rate of new members is higher that the drop-out rate of old members, that number will continue to rise. But the number is deceiving. More than half the membership is Associate and Student members who have no vote in the ACM. We advertise some 300 Chapters but that number is also deceiving. Some 200 are Student Chapters, and you know how it is at school: if the professor says that it will help your grade if you pay nine dollars for ACM student membership, especially "if you are a borderline case" ("and you are all borderline," he adds!), then the whole class joins the ACM. However, not many become full-fledged ACM members after they have received their diplomas. As far as Regular Chapters go, perhaps some 60 of them show some degree of activity. The rest have died since 1982 because the leaders were burnt out by a lack of administrative and financial support from the National organization. Long-time members drop out because of disappointment in the ACM. Some months the number of members who do not renew their memberships is huge. That is what Herb refers to when he speaks of membership falling off. And that was also the reason why they were talking merger with the IEEE there for a while.

To maintain an oversized office in a high rent area costs hands full of money. That is the main reason why Chapter services have been cut to practically nothing. In order to get funds for Chapters and the common programmer, I suggest getting that office out of Manhattan and moving it west. This will accomplish two purposes: lower rent, and half of the staff will quit. Then we will have money for Chapters and local activities.

Jan Matser
ACM Arrowhead Chapter Chair (1967)
ACM San Francisco Peninsula Chair (1977)

"'GOTO Considered Harmful' Considered Harmful" Considered Harmful?

I enjoyed Frank Rubin's letter ("'**GOTO** Considered Harmful' Considered Harmful," March 1987, pp. 195–196), and welcome it as an opportunity to get a discussion started. As a software engineer, I have found it interesting over the last 10 years to write programs both with and without **GOTO** statements at key points. There are cases where adding a **GOTO** as a quick exit from a deeply nested structure is convenient, and there are cases where revising to eliminate the **GOTO** actually simplifies the program.

Rubin's letter attempts to "prove" that a **GOTO** can simplify the program, but instead proves to me that his implementation language is deficient. In the first solution example the **GOTO** programmers got the answer very effectively with no wasted effort:

```
for i := 1 to n
do begin
   for j := 1 to n do
      if x[i, j] <> 0 then
                  goto reject;
         writeln ('the
      first
      all zero row is ', i);
      break;
reject: end;
```

In the consolidated second example, the GOTO-less version seems somewhat more complex, even after the subscript beyond the end of the array is exchanged for a binary flag to determine the result:

```
i := 1;
repeat
   j := 1;
   while (j <= n) and
   (x[i, j] = 0) do
      j := j + 1;
   i := i + 1;
until (i > n) or (j > n);
if j > n then
   writeln('The first all
         zero row is ', i);
```

Both programs, however, serve to point out a missing feature of the language. In the first, the automatic incrementation of a counter is used, but the end condition cannot be tested with the loop construct. In the second, the loop construct tests for end condition, but cannot then increment the counter.

The ideal would be to take both good ideas and use them in combination:

```
found := false;
for i := 1 to n while (Λ
                       found)
   do for j := 1 to n
        while (x[i, j] = 0)
      do if j = n then
              found := true;
if found then
   writeln('The first all
         zero row is ', i);
```

This is not a legal program in Pascal, but the ability to use both a counter and a condition in the loop construct makes the entire job much simpler. The loop counting is done (correctly) by the looping construct, as is the exit testing. I have included a flag to avoid depending on the value of a loop index after exhausting the count, which could be undefined. If a language specifies the counter to be left one past the end of range, this flag would not be needed.

I generally prefer **GOTO**-less code, but will disagree with anyone who thinks there are no valid uses for the **GOTO** in practical engineering. The **GOTO** statement can be easily misused and should therefore be avoided. The hand-coded counters in the second example are also easily misused and should be avoided whenever possible.

The **IF** and **GOTO** are a minimum subset of control flow features, to which the programmer can return when the "correct" feature is not available. **GOTO**, hand coded counters, and extra flags should all be avoided when possible because their use is error prone. I would like to challenge language designers to make the **GOTO** useless by allowing its use and then providing "better alternatives" for each situation where a **GOTO** is needed to work around a language limitation.

Donald Moore
Prime Computer, Inc
192 Old Connecticut Path
Framingham, MA 01701

It was with a mixture of dismay and exasperation that I read Frank Rubin's letter to the Forum. I was dismayed to see this dead horse beaten once again, and exasperated by Rubin's sweeping claims about the virtues of the **GOTO** statement.

This is primarily a religious issue, and those of us who oppose the **GOTO** statement have little hope of converting those who insist on using it. To be sure, the statement has its place in programming, but, recalling Rubin's reference to butcher knives, it should be used only with great care. The fundamental problem is that a programmer, when encountering a **GOTO** in some fragment of code, is forced to begin a sequential search of the entire program to determine where the flow of control has gone. Even in Rubin's simplistic example I had to read the code twice to find the label he was jumping to.

Obviously, an occasional need arises for some type of **GOTO** statement. The solution is for the programming language to provide a **GOTO** statement which has restricted semantics, making it possible to easily determine the target of the desired branch. For example, here is Rubin's example program (determining the first all-zero row of an $N \times N$ matrix of integers), written in C:

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    if (x[i, j] != 0)
      break;
  if (j < n) {
    printf(''The first
          all-zero row is
                %d\n'', i);
    break;
    }
}
```

This fragment has two **GOTO** statements, both named **break**. [*Note: Rubin's program had the second* ***break*** *but not the first—Ed.*] **break** has the effect of jumping to the statement following the innermost loop enclosing the **break** statement. In both uses, the effect of a **GOTO** has been achieved, but the restricted semantics of **break** allow the programmer to easily determine the destination of the branch.

I contend that my version of this program is far more understandable than either of Rubin's programs, with or without **GOTO**. In fact, Mr. Rubin's examples of **GOTO**-less programming do more to highlight a problem in Pascal (which has no **BREAK** statement) than they do to convince me that a **GOTO** statement is required. He starts with an absolutely egregious program, and "improves" it by removing a flag. Here is my attempt at a **GOTO**-less version of the same program, in Pascal:

```
i := 1;
done := false;
while i <= n and not done
                          do
  begin
  j := 1;
  while j <= n and x[i, j]
                    = 0 do
    j := j + 1;
  if j <= n then
    begin
    writeln(''The first
      all-zero row is i);
    done := true
    end;
  i := i + 1
  end;
```

For lack of a **BREAK** statement, I had to use a flag to terminate the outer **while** loop. Unlike Rubin, I did not mix **while** and **repeat** loops, which is confusing, nor did I force the variable *i* to serve dual roles, indexing the array and pointing to the row following the first all-zero row. While I prefer my C version of this program, I would still stand my Pascal against any of Rubin's attempts.

The conclusion to be drawn from this exercise is that good **GOTO**-less code can almost always be written to be better than any equivalent code containing **GOTO**s. Contrary to Mr. Rubin's claims, I (and many others) have had many experiences trying to debug and maintain someone else's code containing **GOTO**s, and have yet to come away from such an experience feeling good about the individual who wrote the original code.

Chuck Musciano
Lead Software Engineer
Harris Corporation
PO Box 37, MS 3A/1912
Melbourne, FL 32902

My congratulations to Frank Rubin for coming out of the closet on "**GOTO**-less" programming. As a professional programmer for many years, I have read and listened to all the arguments in favor of **GOTO**-less programming, hoping that one of them would convince me to give up **GOTOs**. None has so far succeeded. Such an argument would have to show that **GOTOs** always violate the structure of a program even when they are used in accordance with good programming practices. Obviously **GOTOs** are misused, but it is usually not much easier to untangle heavily nested code than it is to decipher spaghetti code.

Both the overuse and the total elimination of **GOTOs** constitute misunderstandings of the relationship among syntactic elements in a programming language. **GOTOs** transfer control just like other, related transfer commands (e.g., **IF**. . .**THEN**). Hence, they should be used when other forms would be inappropriate—by leading to needlessly complex code, for instance. A linguistic analogy can be found in active and passive sentences. Active sentences are easier to produce and understand in relation to their passive counterparts. A "passive-less" English would certainly lead to simpler (better?) structures. However, most linguists would agree that English would loose a portion of its expressive power.

Finally, I will continue to do what I have always been doing: listening to **GOTO**-less arguments and writing well-organized and commented software that makes appropriate use of *all* available features of a programming language.

Michael J. Liebhaber
Child Language Program
University of Kansas
1043 Indiana
Lawrence, KS 66044

Frank Rubin's letter stated that ". . . **GOTO**-less programs are harder and costlier to create, test, and modify." He describes Dijkstra's original letter on the subject (*Communications*, March 1968, pp. 147–148) as ". . . academic and unconvincing . . ." without any support or justification. Finally, he concludes with some example programs which purport to illustrate the logical simplicity of programs which freely use **GOTO** plus BREAK contructs.

Example programs are claimed to fit the sample specification "Let X be an $N \times N$ matrix of integers. Write a program that will print the first all-zero row of X, if any." I had to make several assumptions in order to write the sample program:

1) the language does not support partial evaluation of logical expressions,
2) performance of the final product is not an issue, and
3) performance in the absence of any all-zero row is not specified—in particular, termination is not required.

Apparently, there are also several additional unstated assumptions:

1) the algorithm should test as few elements of matrix X as necessary,
2) the algorithm need not be easily changed to meet a different specification,
3) the language does not support recursion or multiple procedures,
4) the language does support both **GOTO** and **BREAK**, and
5) the program should terminate if a non-all-zero row is found.

Rubin's first example, of a program ". . . where **GOTOs** significantly reduce program complexity," will not run on my UCSD 1.1 **Pascal** system. My Pascal has no **BREAK** statement. This, however, can be circumvented by use of an *additional* **GOTO** and label as follows:

```
    .
    :
writeln
('the first all zero
              row is ', i);
  goto break
reject: end;
break: (*etc.*)
```

By violating all of the *unstated* assumptions, I was able to produce some relatively pleasant solutions to this problem, none of which caused me "to use extra flags, nest statements excessively, or use gratuitous subroutines."

The first solution tests additional elements of the matrix X as necessary, is easily changed to meet a different specification, uses multiple procedures, and does not use either GOTO or BREAK:

```
function allZero:boolean;
var
  az:boolean;
begin az := true;
  for j := 1 to n do
    az := az AND (x[i, j] =
                          0);
  allZero := az
end;

procedure firstZero;
begin i := 1;
  while not allZero do i :=
                        i + 1;
  WRITELN('First all zero
                row is ', i)
end;
```

The second solution uses recursion. With a minor change, the recursive solution tests minimal values of X. Many reject recursion as a viable candidate, but recent evidence [2] confirms that recursion is indeed faster for many classes of problems.

```
function allZero(i, j:
                 integer):
boolean;
begin
  if j > n then
    allZero := true
  else
    allZero := (x[i, j] =
    0) and allZero(i,
                      j + 1)
end;
```

```
procedure firstZero(i:
                    integer);
begin
  if i ≤ n then
    if allZero(i, 1) then
      writeln(''First all
        zero row is ', i)
    else
      firstZero(i + 1)
  else
    writeln('No all zero
                       row')
end;
```

It seems that Rubin takes issue with the complexity of deeply nested control structures. Recent work [3] sheds some light on ways to cope with such problems. In general, poor program layout results from a failure to understand an algorithm, not from the language or from the specific techniques used for implementation.

I submit that there are two issues here:

1) Poor and good programming are language independent. That Rubin is able to reduce the complexity of poor programs is not an indictment of the programming style, but rather an indictment of the programmer(s), and a tribute to Rubin's obvious skill.
2) Modifying programs in which there is a "... conceptual gap between the static program and the dynamic process ..." (to quote Dijkstra's original letter) is generally quite difficult. While some advocate scrapping programs instead of patching them ([1] is a recent example), it seems that writing a program as generally as possible can only make it less expensive to modify.

In order to see the real limitations of **GOTO** programming, try to modify the example programs in Rubin's letter. Modifications should include:

1) locating all rows which are all zero,
2) locating and computing an arithmetic mean for all rows which contain nonzero values, and
3) locating all rows in which the sum of the elements is odd.

Steven F. Lott
Computer Task Group
6700 Old Collamer Road
Syracuse, NY 13057

REFERENCES
1. Hekmatpour, S. Experience with Evolutionary Prototyping in a Large Software Project. *Software Engineering Notes 12:1*, 38–41. January 1987.
2. Louden, K. Recursion Versus Non-Recursion in Pascal: Recursion Can Be Faster. *SIGPLAN Notices 22:2*, 62–67 February 1987.
3. Perkins, C. R., R. W. Norman, S. Dancic. Coping with Deeply Nested Control Structures. *SIGPLAN Notices 22:2*, 68–77 February 1987.

I would like to comment on Frank Rubin's article on **GOTO**s. Although I agree with him in spirit, unfortunately he did not give a fair shake to the non-**GOTO** camp for a correct solution. The problem is to find the first row of all zeroes in an $n \times n$ matrix if such a row exists. A simple correct solution can be derived from the English description of the problem/solution. First, a practical definition of an algorithm can be given as:

1) if the current matrix element is equal to zero then look at the next element in the row;
2) if the current matrix element is not equal to zero then look at the first element in the next row;

But WHOOPS, ...

3) if the column number is equal to $n + 1$, then we have found a row with all zeroes, so write out that row number;
4) if the row number is equal to $n + 1$, then we have run out of rows and there are no rows in matrix X that is full of zeroes.

An English-definition of a procedure that accomplishes the above is

FIND(X, n, r, c) =

Returns the row number of the first row of an n by n matrix X that has all zeroes if such a row exists, or the value of $n + 1$ if the row does not exist. It also

Assumes that all rows whose index is less than r have at least one non-zero element, and that row r has zeroes as all of its elements from 1 to $c - 1$.

[Assumes $(\forall r')$ if $r' < r$ then $X[r'][1..n] \neq \overline{0}$) and $X[r][1..c-1] = \overline{0}$, and gives the first r'' where $r'' \geq r$, $X[r''][1..n] = \overline{0}$, else it gives the value of $n + 1$].

Thus, the LISP-like, tail-recursive definition of "Given an $n \times n$ matrix X, print out the row number of the first row with all zeroes if there exists such a row", is:

FIND(X, n, r, c) = [[[
$c = n + 1 \rightarrow r.$ {from clause 3}
$r = n + 1 \rightarrow r.$ {from clause 4}
$X[r, c] = 0 \rightarrow$ FIND(X, n, r, $c + 1$). {from clause 1}
$X[r, c] \neq 0 \rightarrow$ FIND(X, n, $r + 1$, 1). {from clause 2}
]]]

This definition FIND would be run as "FIND(X, n, 1, 1)" with n already instantiated as some integer. From the definition of FIND, it is easy to write the following program:

```
    :
r := 1;
c := 1;
while (c <> n + 1) and
               (r <> n + 1) do
  if X[r, c] = 0 then
    c := c + 1
  else
    begin
    r := r + 1;
    c := 1
    end;
if r <> n + then
  writelin('Found the
       first row with all
    zeroes, it is :', r);
    :
```

This program was written by putting the recursive clauses in order in a "**if** . . . **then** . . . **else if** . . . etc . . . ," and by putting the escape clauses into the **while** clause predicate location. Since there were two escape clauses, we have to differentiate as to which one terminated the **while** loop. We do this by using an **if** statement after the loop.

The loop invariant for the **while** is:

There exists no row previous to r that is all zeroes, and of row r, its elements from 1 to $c - 1$ are all zeroes,

$$(\neg(\exists r')(r' < r, X[r'][1..n] = \overline{0}))$$

$$\text{and} \quad X[r][1..c-1] = \overline{0}.$$

The condition that will be true at termination of the **while**, after 0 or more iterations is:

We ran out of rows and there was no row of all zeroes, or, the current row r is all zeroes and all the previous rows had at least one nonzero element each.

$$(r = n + 1 \quad \text{and} \quad (\neg(\exists r')(r' \le n, X[r'][1..n] = \overline{0})))$$

$$\text{or} \quad (X[r][1..n] = \overline{0} \quad \text{and} \quad (\neg(\exists r')(r' < r, X[r'][1..n] = \overline{0}))).$$

—which is nothing more than a conjunction of the loop invariant with the negation of the **while** loop guard. (This paragraph may be clouding the point).

Now I would like to criticize Rubin's example programs. In the third program in his letter, in which he eliminated the flag, one can tell that the program was written and then hodged-podged into being hopefully correct. This is shown by the "$i := i + 1$;" statement. If a row was all zeroes, then why increment i? Because it is necessary to make the program work.

Thus, all the statements are not fully (correctly) utilized, and an unnecessary loop construct seems to be an unwarranted complication.

In the first program (the "preferred" **GOTO** program) the "**for** $j := 1$ **to** n **do**" behavior is not consistent with the commonly understood definition of the **FOR** loop. A **FOR** loop specifies a definite number of iterations. Depending on the data of row i, the **FOR** j loop may do its body for n iterations, or it may do it for less. The construct used in that program is a quasi-**FOR** definition where it is somewhat like a **FOR** definition except. . . . So you have a **GOTO** which can prematurely break you out of the "**FOR** $j := 1$ **to** n **do**" loop, and a BREAK that can break you prematurely out of the "**for** $i := 1$ **to** n" loop. These two quasi-loops make the program error prone and make proving program correctness harder.

In conclusion, although the derivation of my program may appear contrived, I did derive a similar program in less than five minutes intuitively, except that the guards for the **while** loop were not as good as those in the presented version. Then I thought of how to systematically derive a correct solution from the problem, and thus, the letter.

Incidentally, there are intuitive ways to write non-**GOTO** programs that will run as efficiently as Rubin's **GOTO** program (or better). One involves a different data-structure, which would be an $n + 1$ by $n + 1$ matrix containing sentinels in the extra row and column.

Lee Starr
10 Overlook Terrace
Walden, NY 12586

TO OUR MEMBERS:

More than 15,000 members took advantage of the special multiple-year renewal offer in November and December 1986. As a result of this enthusiastic response, for which we were not fully prepared, processing of normal membership renewals was delayed, and some members who renewed through the special offer received incorrect second notices. If you received such a notice, we wish to assure you that your payments have been applied properly and your publications will arrive on schedule.

In addition, membership cards were not sent with the multiple-year renewal offer because of the nature of that offer. For those of you who responded to the offer, new membership cards are being prepared and will be sent as soon as possible.

We apologize for any inconvenience that these processing problems may have caused you, and urge you to contact the ACM Member Services Department at ACM Headquarters if you have any remaining unresolved problems with your membership.