A NOTE ON PROGRAM DEVELOPMENT[†]

David Gries

TR 74-202
March 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

---

† This research was supported by the National Science Foundation under grant GJ-28176.

A NOTE ON PROGRAM DEVELOPMENT

David Gries

In [3], Peter Nauer argues that top-down programming or step-wise refinement, as advocated by Edsger Dijkstra [2] and Niklaus Wirth [5], is not the only or the best way to program, and stresses that programming must allow for personality factors. Nauer makes his point by describing in detail his analysis and development of the 8-queens problem previously developed by Wirth [5]. (The problem is to print all chess boards -- excluding symmetric variations -- with 8 queens, where no capture is possible.)

Nauer's analysis of the problem and development of its solution is very interesting and should be studied by anyone interested in how people solve problems. Nauer knows how to attack and analyze a problem and is clearly more inventive and creative than most programmers. He also knows his programming language well -- he is the only person I know of to exploit ALGOL's call-by-name in a nontrivial, useful manner.

And yet, Nauer's solution leaves much to be desired. It has no structure and consequently is difficult to understand. Basically, it is just a bunch of good ideas held together with a thin thread of goto's. Certainly, it took me far too long to understand it to the point of feeling it might be correct.

Nauer does say in [3] that "the resulting program is to be taken as only an incidental result of the work ... this program is by no means ideal" [page 365]. But his program

should never have been keypunched and tested -- it is only half-developed. At some point during development (probably point 21), Nauer should have stopped and said: "I have my ideas now; let's see about putting them together into a well-structured program and verify its correctness." A top-down description should have then been constructed.

Producing a well-structured, reasonable program should not be left 'til after a program is "debugged", but must be made an integral part of the design phase.

The main thesis of this note is that program development should be described as a two-step process:

1. Develop the ideas for the program -- create data structures, work on ideas for subalgorithms (clusters), etc.

2. Write the program, using the ideas developed in step 1. In general, we should aim at a top-down description of the algorithm.

G. Polya [4], in discussing general problem solving, calls these two steps designing a plan and implementing the plan. Both are equally important.

Step 1 is often difficult, as Nauer points out, although most programmers do seem to be able to write a program (which works part of the time). Some people may be able to develop ideas top-down; others obviously cannot, or at least they don't feel inclined that way. Step 1 is even more difficult to teach, although there are more and more good books on problem solving (e.g. [4,6]).

Step 2 is difficult only because programmers don't want to do it. They feel it's a waste of time and would rather spend their time debugging. Step 2 requires patience, determination, attention to detail, and, of course, the ability to abstract.

Producing a top-down description of the algorithm during step 2 -- building Dijkstra's "necklace of pearls" -- is an important, necessary part of program development. Whether one creates the necklace top-down, bottom-up or middle-out doesn't really matter. But I also have the feeling that the act of performing step 2 in a careful, conscientious manner will influence the way step 1 is performed. The more one does step 2 in top-down style, the more one will tend to unconsciously perform step 1 that way also.

A well-structured solution

Nauer has performed step 1 for the 8-queens process, better than most of us could do. I have performed step 2 using Nauer's ideas. The reader is invited to compare this solution with Nauer's, with respect to ease of comprehension.

A few changes have been made. For example, M now counts the number of queens on the board instead of that number minus 2. The array COUNT now counts how many times each queen is placed on the board. Nauer's idea was to count, for each queen, how many times a column was rejected (see his point 23). Since there are 8 columns, one would expect this to be a multiple of 8, but Nauer inconsistently does not count when a column is rejected because a queen is already in that column. I feel this

inconsistency arose because of the unnecessary complexity of the algorithm.

Thirdly, queens 1 and 2 are put on the board in reverse order; that is, the row order is now 5, 4, 3, 2, 1, 6, 7, 8. This makes things a bit more systematic.

The solution is written in PL/C [1] (Cornell's PL/I subset) with the added feature of macros, which is currently being added to PL/C. This feature allows us to write whole English statements in a program, with their refinements appearing later as macro bodies. It also allows us to assign mnemonic names to constants (e.g. TRVE for '1'B).

A few comments on program style are in order. First of all, the indentation, as usual, is important. We use indentation not only to indicate the substatements of a PL/I statement, but also to indicate the refinement of an English statement. Thus, if we read

```
/*statement 0*1*/
     statement 1;
       .
       .
       .
     statement n;
X = Y;
Y = Z;
```

We see that the program consists of the three statements "statement 0", "X = Y", and "Z = W". The indented sequence "statement 1" through "statement n" forms the refinement of statement 0. This extra use of indentation allows the programmer to describe all levels of abstraction within the program.

Most comments, then, are just high-level statements whose refinements appear indented underneath.

If the indentation rules are followed carefully, then END's for compound statements and loops are redundant. We therefore do not place an END directly under the corresponding DO, but just after the last substatement that the DO - END pair encloses.

Finally, goto's are used only to terminate subalgorithms. Goto's are useful if used properly, simply because current languages don't have all the control structures we need. Nauer's solution contains goto's used in a way which contributes greatly to program misunderstanding; unfortunately most programmers use it in this manner also. Let me try to explain my points using part of Nauer's algorithm:

```
    comment  Select next possible column;
        if  col ≠ 0  then
Q1:     FREE[col] := true;
try for column:
        if  col = 8  then go to reject row;
        col := col +1
         :
         :
```

We see that this subalgorithm is supposed to select the next possible column. As we read, however, we see the label Q1 and surmise that part of the program will jump to this labelled statement. Immediately our train of thought is disturbed, because we begin wondering why a jump to Q1 is necessary. In fact we may forget about trying to understand subalgorithm

Select next possible column, and look for the statement which jumps to Q1. In this manner, our mind keeps fluttering from point to point, never alighting anywhere long enough to understand it.

A second, more crucial point. The subalgorithm select next possible column should be an independent subalgorithm in its own right, which could be lifted out and placed in another algorithm which needed the subalgorithm and which, of course, used the same data structures. Such independence of subalgorithms is necessary if we are to be able to understand a large program. And yet we find that this subalgorithm is not independent. It suddenly branches to another part of the program, reject row. If we are to understand select next possible column, then we must also understand subalgorithm reject row.

Now consider the solution given in this paper. Goto's are used, but only to terminate a subalgorithm. To indicate this clearly, labels label null statements. Consider the macro print_board_if_not_symmetric, which consists of the three statements

```
ROW(ROWM) = COLM;
/*CHECK FOR SYMMETRY AND THEN PRINT;*/
ROW(ROWM) = 0;
```

Within the refinement for the statement check for symmetry and then print, in several places, when nonsymmetry is detected, the statement is terminated by a goto END_CHECK_FOR_SYMMETRY_AND_PRINT.

The variables used within the program are as follows:

General information about solutions

a) N solutions have been printed.

b) COUNT(i) is the number of times queen i has been added to the board, $1 \leq i \leq 8$.

Information about the current board. This is initialized each time queen 1 is placed on the board, and holds thereafter.

a) M queens are on the board, $1 \leq M \leq 8$.

b) Queens 1, 2 are in rows 5, 4, in columns C5 and C4, respectively.

c) Queens 1,...,M-1 are described by arrays ROW and COL as follows:

$$\text{ROW}(i) = \begin{cases} 0 & \text{if none of queens } 1,\ldots,M\text{-}1 \text{ are in row } i \\ j & \text{if a queen is in row } i, \text{ column } j \end{cases}$$

FREE(i) = "none of queens 1,...,M-1 are in column i

d) Variables ROWM and COLM give the row and column of queen M.

e) Queens are put in rows 5, 4, 3, 2, 1, 6, 7, 8, in that order.

Rows LOWROW,...,HIGHROW are covered by queens 1,...,M. Queen M, when first placed on the board, is placed in column 0; it is subsequently moved to column 1.

```
QUEENS:  PROCEDURE OPTIONS(MAIN);
/* PRINT ALL CHESS BOARDS WITH 8 NON-CAPTURING QUEENS EXCLUDING */
/* SYMMETRIC VARIATIONS.*/
/* SEE ARTICLE FOR DESCRIPTION OF THE VARIABLES */
    DECLARE (I, M, ROWM, COLM, LOWROW, HIGHROW, C4, C5) FIXED BINARY;
    DECLARE (ROW(1:8), COUNT(1:8)) FIXED BINARY;
    DECLARE (FREE (1:8)) BIT(1);


 N = 0; COUNT = 0;

 /* PLACE QUEENS 1 AND 2 IN POSITIONS (5,C5), (4,C4) AND PRINT ALL */
 /* SOLUTIONS WITH THAT CONFIGURATION, FOR ALL POSSIBLE COLUMNS C4 AND*/
 /* C5. SEE WALK'S POINTS 12 AND 13 FOR POSSIBLE INITIAL BOARDS.*/
    DO C4 = 1, 2, 3;
      DO C5 = C4+2 BY 1 TO 9-C4;
         /*INITIALIZE BOARD WITH QUEEN 1 */
            FREE  = TRUE; ROW = 0; M = 1; COUNT(M) = COUNT(M)+1;
            ROW(1) = 5; COLM = C5; LOWROW = 6; HIGHROW = 5;
         /*ADD QUEEN 2 IN ROW 4 COLUMN C4.*/
            ADD_A_QUEEN_IN_COLUMN_0; COLM = C4;
         PUT SKIP LIST('NEW INITIAL BOARD:', C4, C5);
         PRINT_LEGAL_BOARDS_WITH_QUEENS_1_AND_2_IN_CURRENT_POSITIONS;
         END; END;
    PUT SKIP LIST('QUEEN PLACEMENT COUNT:', COUNT);



    END QUEENS;


*MACRO
****
   TRUE = '1'B %;  FALSE = '0'B %;

   ADD_A_QUEEN_IN_COLUMN_0 =
      DO; M = M+1; COUNT(M) = COUNT(M) +1;
          FREE (COLM) = FALSE; ROW(ROWM) = COLM;
          IF M > 5 THEN
               DO; LOWROW = 1;   HIGHROW = M-1; ROWM = M;   END;
          ELSE DO; LOWROW = 7-M; HIGHROW = 5;   ROWM = 6-M; END;
          COLM = 0;  END  %;


   DELETE_QUEEN_M =
      DO; M = M-1;
          IF M > 5 THEN
               DO; HIGHROW = M-1; ROWM = M;   END;
          ELSE DO; LOWROW  = 7-M; ROWM = 6-M; END;
          COLM = ROW(ROWM); ROW(ROWM) = 0; FREE (COLM) = TRUE; END %;


   FIND_NEXT_BOARD___IF_NONE_GOTO_END_PRINT_LEGAL_BOARDS =
      DO; DO WHILE (COLM = 8);
             DELETE_QUEEN_M;
             IF M = 2 THEN GOTO END_PRINT_LEGAL_BOARDS; END;
          COLM = COLM + 1;  END;  %;
```

```
PRINT_LEGAL_BOARDS_WITH_QUEENS_1_AND_2_IN_CURRENT_POSITIONS =
   DO; ALL_A_QUEEN_IN_COLUMN_0; /* PLACES QUEEN 3*/
       DO WHILE(TRUE);
          /* MOVE QUEENS 1 - 8 TO NEXT LEGAL BOARD, SINCE ALL BOARDS */
          /* WITH CURRENT SETUP HAVE BEEN PRINTED. TERMINATE MACRO IF*/
          /* NO MORE BOARDS (BY JUMPING TO END_PRINT_LEGAL_BOARDS.)*/
             FIND_NEXT_BOARD__IF_DONE_GOTO_END_PRINT_LEGAL_BOARDS;
             DO WHILE (BOARD_IS_NOT_LEGAL);
                FIND_NEXT_BOARD__IF_NONE_GOTO_END_PRINT_LEGAL_BOARDS;
                END;
          IF I < 8 THEN ADD_A_QUEEN_IN_COLUMN_0;
                   ELSE PRINT_BOARD_IF_NOT_SYMMETRIC;
          END;
       END_PRINT_LEGAL_BOARDS:;  END; /*;


PRINT_BOARD_IF_NOT_SYMMETRIC =
BEGIN; DECLARE (S, R, I, COLUMN(1:8)) FIXED DECIMAL(9);

    /* SEE MAUR'S POINT 34 AND HIS CHECK FOR SYMMETRY, PROCEDURE */
    /* SOLVE F/S BEEN CHANGED BECAUSE PL/I HAS NO CALL BY NAME.*/
     SOLVES: PROCEDURE( ARR, S, R);
        DECLARE (ARR(*), S, R) FIXED DECIMAL(9);
        DECLARE A FIXED DECIMAL(9);
        S = 0; R = 0;
        DO A = 4, 5, 3, 2, 1, 6, 7, 8;
           S = 10*S+ARR(A); R = 10*R + ARR(9-A); END; END SOLVES;

     ROW(ROW#) = COLM; /* FILL IN LAST ROW FOR SYMETRY CHECK AND*/
                       /* PRINTING. MUST BE DELETED AT END OF SUBALG.*/
     /* CHECK FOR SYMMETRY AND THEN PRINT.*/
        CALL SOLVES(ROW, S, R);
        IF R < S | 99999999-R < S THEN GOTO END_CHECK_FOR_SYMMETRY__;
        DO T = 1 TO 8; COLUMN(ROW(T)) = T; END;
        CALL SOLVES(COLUMN, R, T);
        IF R < S | 99999999-R < S THEN GOTO END_CHECK_FOR_SYMMETRY__;
        IF T < S | 99999999-T < S THEN GOTO END_CHECK_FOR_SYMMETRY__;
        /* PRINT SOLUTION.*/
           K = K+1; PUT SKIP LIST('SOLUTION', K);
           DO S = 1 TO 8;
              PUT SKIP EDIT('    ') (A);
              DO I = 1 TO 8;
                 IF I = ROW(S) THEN PUT EDIT('Q') (A(2));
                               ELSE PUT EDIT('.') (A(2)); END; END;
           PUT SKIP;
     END_CHECK_FOR_SYMMETRY__:;

     ROW(ROW#) = 0; END;  /*;

    DECLARE BOARD_IS_NOT_LEGAL ENTRY RETURNS(BIT(1));
  BOARD_IS_NOT_LEGAL: PROC RETURNS(BIT(1));
    DECLARE K FIXED BINARY;
    IF ¬ FREE (COLM) THEN RETURN(TRUE);
    /* CHECK FOR DIAGONAL CAPTURE. SEE MAUR'S PT 15 AND REVISION PT 30*/
        DO K = LOWROW TO HIGHROW;
           IF ABS(ROW#-K) = ABS(COLM-ROW(K))
              THEN RETURN(TRUE);
           END;
    /* CHECK FOR PARTIAL SYMMETRY. SEE MAUR'S POINT 34*/
        IF ABS(COLM-4.5) < 3 THEN
           IF ABS(ROW#-4.5) > 5-COLM
              THEN RETURN(TRUE);
    RETURN(FALSE);    /* BOARD IS LEGAL*/
    END BOARD_IS_NOT_LEGAL;
****
****
****
```

References

[1] Conway, R.W., and T.R. Wilcox. Design and implementation of a diagnostic compiler for PL/I. Comm. ACM 16(March 1973), 169-179.

[2] Dijkstra, E. Notes on structured programming, Technical University Eindhoven, 1970.

[3] Nauer, P. An experiment in program development. BIT 12(1972), 347-365.

[4] Polya, G. How to Solve It. Doubleday Anchor, New York, 1957.

[5] Wirth, N. Program development by stepwise refinement. Comm. ACM 14(April 1971), 221-227.

[6] Wickelgren, W. How to Solve Problems. Freeman and Co., San Francisco, 1974.