A PROGRAM TO SOLVE "THE CONVICT PROBLEM"

This is a ProTeX variant of the CWEB puzzle program of Lee Wittenberg.

1. The Problem Our task is to help a convict escape from prison. The prison is 4 cells square and the convict (represented by ‘C’ in the diagram, below) is in the northwesternmost cell. He can move horizontally or vertically (not diagonally) from cell to cell. The only exit is in the southeasternmost cell.

C	P	P	P
P	P	P	P
P	P	P	P
P	P	P	

Complicating the problem are the 14 policemen (represented by ‘P’s in the diagram) blocking the convict’s way. The convict must kill all the policemen before he can leave, but cannot return to any cell he has already been in (that would be too easy).

2. The Solution The program that solves the problem will be laid out like most C programs:

```
                    ▷▷ program ◁◁
◁◁ Header files used by the program ▷▷
◁◁ define ▷▷
◁◁ Global variables ▷▷
◁◁ Functions ▷▷
◁◁ The main program ▷▷
                         ◇◇◇
```

3. We will need to represent the prison—a 2-dimensional array seems the logical choice. As the convict "visits" each cell, we will put a number denoting the order in which the cell was visited into the appropriate array element (we will use zero to represent a cell that has not yet been visited). Thus we can use the zero-ness of a cell to determine whether or not it has been visited.

We can also use this property to avoid making special cases out of the outer cells (which have fewer than 4 exits to other cells). We create an array 2 elements wider than the prison and initialize all the array elements that do not correspond to actual cells to -1 (marking them as "already visited"—a convenient fiction). All cells in the prison now have 4 neighbors and can be treated exactly alike.

```
                    ▷▷ define ◁◁
#define has_been_visited(m,n) (prison[m][n]!=0)
                         ◇◇◇
```

```
                 ▷▷ Global variables ◁◁
int prison[6][6] = {
  {-1,-1,-1,-1,-1,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1, 0, 0, 0, 0,-1},
  {-1,-1,-1,-1,-1,-1}
  };
                         ◇◇◇
```

4. The main program is simple. We let the —solve— function do all the work. If —solve— succeeds for the initial cell $(1, 1)$, we print the configuration of the prison (showing the order in which the cells were visited); if not, we print an error message. The latter case should never happen—it will occur only if there is a bug in the program or if we have misunderstood the problem.

```
                 ▷▷ The main program ◁◁
void
main(void)
{
  if (solve(1,1))
          print_prison();
        else
          fprintf(stderr, "Impossible\n");
}
                         ◇◇◇
```

5. In order to produce the necessary output we need to include the "Standard I/O" library:

```
                    ▷▷ Header files used by the program ◁◁
#include <stdio.h>
```

◇ ◇ ◇

6. Printing the prison configuration is trivial, so we might as well get it out of the way.

```
                              ▷▷ Functions ◁◁
void print_prison(void)
{
  register int i,j;
  for (i=1; i<=4;i++)
    for (j=1; j<=4;j++)
        printf("%2d%c", prison[i][j], j==4?'\n':'\t');
  putc('\n',stdout);
}
```

◇ ◇ ◇

7. The —solve— function is fairly straightforward, but it has a few special cases to deal with. The parameters represent the row and column numbers of the next cell to visit. If the cell has already been visited, it can't be visited again, so the current attempt at a solution fails. The solution can also fail prematurely if the convict attempts to visit the exit cell $(4, 4)$ without having visited all the other cells (and killing the policemen therein).

If the cell has not been visited before, we mark it as having been visited and recursively check each of the 4 neighboring cells to see if a solution exists starting from that cell (remember, all cells previously visited have been marked and cannot be visited again). If a solution exists, we report success. Otherwise, we report failure and pretend that the convict hasn't actually visited this cell yet.

```
                               ▷▷ define ◁◁+
#define  succeed return 1
#define  fail return 0
```

◇ ◇ ◇

```
                              ▷▷ Functions ◁◁+
int
solve(register int m, register int n)
{
    if(has_been_visited(m,n))
                       fail;
    ◁◁ If this is the exit cell, |succeed| if we have visited all the other cells, |fail|
      otherwise ▷▷;
    prison[m][n]=++number_of_cells_visited;   /* mark cell as ``visited'' */
  if(solve(m+1,n)||solve(m,n+1)||solve(m-1,n)||solve(m,n-1))
    succeed;
  ◁◁ Pretend that the convict hasn't actually visited this cell yet ▷▷;
  fail;
}
```

◇ ◇ ◇

8.

```
▷▷ If this is the exit cell, |succeed| if we have visited all the other cells, |fail| otherwise ◁◁
      if (m==4&&n==4) {
```

```
  if (number_of_cells_visited==15)   /* all other cells have been visited */
    succeed;
  else fail;
}
```

◇◇◇

9.

```
     ▷▷ Pretend that the convict hasn't actually visited this cell yet ◁◁
--number_of_cells_visited;prison[m][n]=0;
```

◇◇◇

10. We start out with no cells having been visited.

```
                          ▷▷ Global variables ◁◁+
int number_of_cells_visited=0;
```

◇◇◇

11. Epilogue When we ran the program, it produced the 'Impossible' message. After checking out the program extensively, we found no bugs—we did not completely understand the problem. It turns out that the convict, while not allowed to return to a cell in which he has killed a policeman, *is* allowed to return to his original cell. The following is a valid solution:

$$\begin{aligned}(1,1) &\to (1,2) \to (1,1) \to (2,1) \to (3,1) \to (4,1) \to (4,2) \to (3,2) \to (2,2)\\ &\to (2,3) \to (1,3) \to (1,4) \to (2,4) \to (3,4) \to (3,3) \to (4,3) \to (4,4)\end{aligned}$$