

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/337312991>

An Easy Instruction to Code 8 Queen Problem

Article · November 2019

CITATIONS

0

READS

264

1 author:



[Vishv Mohan Malhotra](#)

The LNM Institute of Information Technology

78 PUBLICATIONS 346 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Balancing robustness and redundancy in the design of environmental sensor networks [PhD thesis] [View project](#)



Student Guides for PintOS (an Educational Operating System Package by Ben Pfaff): Threads and User Programs [View project](#)

CSE 104L: Programming Lab

Training Set – Recursive Functions

Year: 2019 – August-December

LNMIIT, Jaipur

A queen is perhaps the most powerful piece on a chessboard. Chessboard is an 8*8 board of squares (positions or locations). We will refer to these locations by their row and column numbers – in fact to avoid any confusion by their (x, y) coordinates. The origin is assumed to be at position (0, 0).

A queen (also called in India, Vazir or Mantri) attacks all positions that are on the same column, or on the same row, or on the same forward (45°) or backward (135°) diagonal. Mathematics to determine the attacked positions is simple. Queen at (x, y) attacks position (p, q) if any of these is true:

1. $x == p$
2. $y == q$
3. $(x - y) == (p - q)$
4. $(x + y) == (p + q)$

Further, if x, y, p and q are on a chessboard then all these conditions are true:

1. $0 \leq x \leq 7$,
2. $0 \leq y \leq 7$,
3. $0 \leq p \leq 7$, and
4. $0 \leq q \leq 7$.

Question: Can we insert 8 queens on a chessboard without their positions being in a conflict (attacking situation) configuration? **Answer:** Yes, we can. And, the program we write today does so. The puzzle is popularly called Eight Queens Problem.

Students must remember to follow all requirements of a good coding practices including appropriate comments, meaningful identifier names, indentation and use of blank-lines.

New requirements being added now include: to divide the program activities into functions that are non-trivial in size and focused on a single meaningful unit of work. This makes it possible to test the code systematically and encourage reuse of quality code where opportunities exist.

Task 01: We can represent the queens on a chessboard by an `int` array of length 8:

```
int queens[8] = {-1, -1, -1, -1, -1, -1, -1, -1};
```

Value of array element at index y, `queens[y]` is the x-coordinate of the queen in row y. In a solution of the problem, each row has exactly one queen. We use -1 to indicate that the row does not have a queen yet.

Task 02: Write a C function `void printBoard(int queens[]);` with the following specifications: print a chessboard as an 8 rows of 8 symbols. Use symbol '.' for each unoccupied position and a Q for the position holding a queen. Add a blank line after printing a complete chessboard as a separator.

The function must call `exit(1)` if the array `queens` has an element other than -1 to 7. Such configuration is obviously invalid.

The function can be tested by making calls from function `main()` to the function with different configurations of array `queens`. In your test suite please include this `queens` configuration: {0, 4, 7, 5, 2, 6, 1, 3}

Task 03: Write function `int isSafe(queens)` such that it returns 1 if none of the queens placed on the chessboard are in a conflict configuration. The function will return 0 if a conflict is detected. Some rows may not have any queen inserted in them.

If an element has value outside the permitted range (-1 to 7) then the function should call `exit(1)` to terminate the program (Note: prototype of the function `main()` of the program will be `int main(void) ...`). Also, if element `queens[i]` is -1 for some `i`, make sure that all `queens[j]`, `j>i` are also -1. If not, exit your program with `exit(1)`. These later checks are not essential but improve program's resistance against inadvertent mistakes. These are sometimes called *integrity* or *sanity* checks.

To help you make the safety check easy, you may wish to write a function `int conflicts(int x, int y, int p, int q)` with following contract:

Obligations of the caller:

1. Both positions (x, y) and (p, q) are on chessboard. That is each of these four values are within interval 0 to 7.
2. Positions (x, y) and (p, q) are not the same.

The function guarantees the return value to be:

1. If queen at position (x, y) attacks the queen at position (p, q) it returns 1.
2. Otherwise, it returns 0.
3. If either queen is outside the board, the program exits.

Test your function `conflicts()` first using a test suite. For this you can once again use function `main()` as the driver – the function that drives the program by making calls to other functions.

After you are satisfied with your implementation of `conflicts()`, you must test function `isSafe()`. You can easily derive your test suite by altering the test cases used in the previous task(s).

Consider if you can divide function `isSafe()` in smaller more specific functions. For example a function to test *sanity* and another to test freedom from conflicts among queens on the board.

Task 04: We will now write a recursive function `int placeQs(int queen[], int y)`. The specifications are as follows:

1. If `y >= 8` return true (1). A solution has been found for the problem. Hurray.
2. Run a for-loop: `for (x = 0; x < 8; x++) { ... }` attempting to insert a queen in row `y` at location `(x, y)`. This is done by assignment `queens[y] = x`. Two checks are made before the position is accepted as feasible.
 - a. One, the new arrangement must be determined to be safe by function `isSafe()`. And,
 - b. Recursive call `placeQs(queens, y+1)` must return true (1).
3. If both conditions 2a and 2b above are met, the function returns true (1) to indicate that it has managed to place all 8 queens. (Function `main()` will print the configuration for the user to see by calling `printBoard()`).
4. If one or more of these conditions fail, the search for the other configurations of the queens must continue. But before you let the for-loop continue, make sure you do the assignment

```
queens[y] = -1;
```

to undo your previous placement of the queen that did not deliver a solution.
5. If the for-loop has run to the last `x` value in the row, report the failure to place the queen in the current row. This is simply done by returning 0 (false) back to the calling function. No printing of a message is needed here.

Use the specified test case in task 02 to derive some useful test cases for this function. You can do so by setting values in array `queens[]` to -1 at the last few indices. Start by first using `queens[] = {0, 4, 7, 5, 2, 6, 1, -1};`

Other suggested configurations are:

```
queens[] = {0, 4, 7, 5, 2, 6, -11, -1};
```

```
queens[] = {0, 4, 7, 5, -1, -1, -1, -1};
```

Smart students may also include call to function `printBoard()` to help them see the results visually.

Task 05: In function `main()` you begin the ball rolling by simply calling `placeQs(queens, 0)`. And, if all goes well print the result by calling function `printBoard()`.

Task 06: There are multiple solutions for this popular programming puzzle. It is easy to get these variations from your program!

Make this simple change in your code. When you reach `y == 8`, print the chessboard configuration as you did previously; but this time you make the call from within function `placeQs()`. And, instead of reporting a success, report a failure (`return 0;`) to motivate the calling function to continue searching!

Task 07: Show your mastery on the code by reporting the count of number of solutions found by your program. There are two easy ways and I suggest that you try both ways to count the number of solutions.

First way is to declare a global variable `count`. Each time a solution chessboard is printed, increment `count` by 1. The value of this variable can be printed in function `main()` near the end of your program to report the number of solutions possible.

The alternate way is by using declaration `static int count;` in function `printBoard()`. It can track the number of solutions printed and returns the count as a return value. That is, the new prototype for function will be: `int printBoard(int queens[]);`

**** That is all for today ****