Introduction to Programming

—

Midterm examination on machine

—

G1: clement.riu(at)enpc.fr
G2: marie.haghebaert(at)inrae.fr
G3: thomas.belos(at)enpc.fr
G4: abderahmane.bedouhene(at)enpc.fr
G5: youval.vanlaer(at)enpc.fr
G6: pascal.monasse(at)enpc.fr
G7: thomas.daumain(at)enpc.fr

—

29/10/21

—

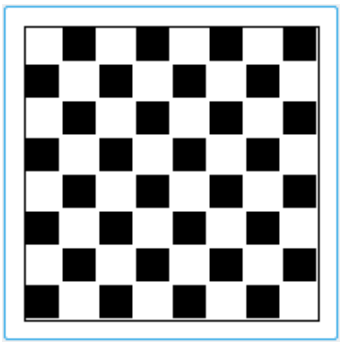
1 Instructions

1.1 The Queens Game

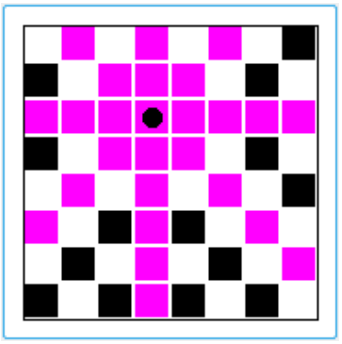
The goal of the game is to place as many queens as possible on a chessboard so that they do not threaten each other. Each queen threatens the cases on same row, column, or diagonal. For most board dimensions, it is possible to have a solution with one queen per row.

It is more important to deliver a clean code (commented and correctly indented) that *compiles* than answering all questions. For that, check after each step that the build works. At the end, create an archive with source code and file `CMakeLists.txt` to upload on educnet.

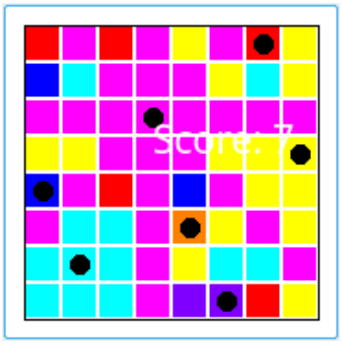
A variant of the queens game.



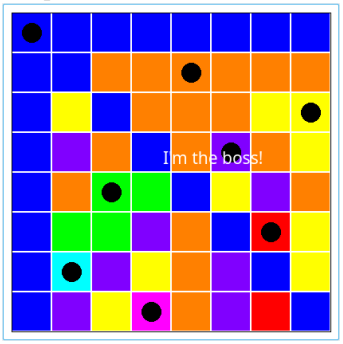
Initial board.



One queen and threatened cases.



Human player, score=7.



Machine, perfect score 8 queens.

1.2 Basic display

1. In a file *separate* from the one containing your `main` function, define the constants of the project: the dimension $n = 8$ (8×8 chessboard), a border (in pixels) so that the chessboard is not displayed right at window's edge, and a zoom factor $z = 100$ for display of each case of the board.
2. Define a structure `Board` comprising a 2D array of cases, each element encoding the color of the case.
3. Define a function `init` that puts the board in an alternance of black and white cases.
4. Define functions `draw` and `drawCase`, respectively for the whole board and for an individual case. Leave a one pixel wide space between adjacent cases.
5. In the main function, create, initialize and display the chessboard.

1.3 Human player

6. Define a function `getCase` converting pixel coordinates (x, y) into case coordinates (row and column). This function will be used to know the case of the board where a mouse click happens.
7. Define the function `centerCase` that returns the pixel coordinates of the *center* of the case at given row and column.
8. Write a function `isFree` that indicates whether a case at given row and column is threatened by a queen of not (white and black indicate free cases, any other color means it is not free).
9. Write a function `nFree` counting the number of free cases of a board.
10. Each queen is associated to a color. Write an array of `n`$= 8$ fixed colors (excluding black and white).
11. Write a function `forbid` that takes a board and a case to paint the free cases threatened by a queen placed there in the fixed color given by the column of the queen. Instead of writing 8 loops, use an auxiliary function taking a position and a direction vector $(dx, dy) \in \{-1, 0, 1\} \times \{-1, 0, 1\} \setminus \{(0, 0)\}$.
12. Using the previous functions, in a function `play` inside the file containing the main function, present an empty board to the player, let her click cases until no more free cases are left. After each click on a free case, the board is repainted to show the threatened case and the queen represented by a black disk.
13. Make the program write the score of the player (number of queens). It may be in a console instead of the graphical window if you do not know how to do it.

1.4 Machine player

A permutation P of $\{0, \ldots, n-1\}$ is these n integers written in any order. The idea is to place queens at all positions $(i, P(i))$ and check that they do not threaten each other. To generate all permutations, we can use the following algorithm:

(a) Take two arrays $t = \emptyset$ and $u = (0, \ldots, n-1)$

(b) If $u = \emptyset$, t is a permutation

(c) Otherwise, take all arrays t'_i deduced from t by appending a single element u_i of u and iterate (b) with arrays t'_i and $u' = u$ minus u_i.

14. Write a structure `tab` storing an array of `n` integers. We will use some arrays with fewer than `n` elements, but we will use a variable to store the actual number of elements.
15. The function `permute` takes two such structures t and u and the actual number of elements in t (if t has k elements, u has $n - k$) to generate all permutations.
16. For each permutation, a function `check` is called, which builds the corresponding board and verifies if it is a winning configuration.
17. In that case, the filled board is shown and a message bragging about the result is displayed. A mouse click is expected to continue searching for more configurations.