le code est parfois noyé par les vérifications — revoir la technique pour plus de lisibilité —


```c
#include "game.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "game_aux.h"

// Definition of struct game: it is made of an array of squares; and two arrays
// of uints
struct game_s {
    square* squareArray;
    uint* nbTentsRowArray;
    uint* nbTentsColArray;
};

//---------- DEBUG FUNCTIONS ----------

// Print an error message and stop the execution of the program
const char treat_error(char* error_message) {
    fprintf(stderr, "ERROR : %s \n", error_message);
    exit(EXIT_FAILURE);
}

// Check if the game passed as argument is valid and initialized
void check_game(cgame g) {
    if (g == NULL || (*g).squareArray == NULL || (*g).nbTentsRowArray == NULL ||
        (*g).nbTentsColArray == NULL) {
        treat_error("Unvalid game pointer!");
    }
    return;
}

//---------- DEFINITION OF GAME.H FUNCTIONS ----------

game game_new(square* squares, uint* nb_tents_row, uint* nb_tents_col) {
    // Checking prerequisites
    if (squares == NULL || nb_tents_row == NULL || nb_tents_col == NULL) {
        treat_error("Unvalid parameter");
    }

    // Allocating memory for an empty game
    game g = game_new_empty();

    // Checking if the game is valid
    check_game(g);

    // Initializing square array
    for (int i = 0; i < DEFAULT_SIZE * DEFAULT_SIZE; i++) {
        (*g).squareArray[i] = squares[i];
    }
    // Initializing expected tents arrays
    for (int i = 0; i < DEFAULT_SIZE; i++) {
        (*g).nbTentsRowArray[i] = nb_tents_row[i];
        (*g).nbTentsColArray[i] = nb_tents_col[i];
    }

    // We return the initialized game pointer
    return g;
}

game game_new_empty(void) {
    struct game_s* myGame;
    myGame = malloc(sizeof(struct game_s));
    myGame->squareArray =
        (uint*)calloc((DEFAULT_SIZE * DEFAULT_SIZE), sizeof(uint));
    myGame->nbTentsColArray = (uint*)calloc(DEFAULT_SIZE, sizeof(uint));
    myGame->nbTentsRowArray = (uint*)calloc(DEFAULT_SIZE, sizeof(uint));

    check_game(myGame);
```

faire le test en plus

quelle est la différence ?

inutile —


```c
    return myGame;
}

game game_copy(cgame g) {
    check_game(g);

    // initialization and testing of copyGame
    game copyGame = game_new_empty();

    // copy structure elements in our copyGame
    memcpy(copyGame->squareArray, g->squareArray,
           sizeof(uint) * (DEFAULT_SIZE * DEFAULT_SIZE));
    memcpy(copyGame->nbTentsRowArray, g->nbTentsRowArray,
           sizeof(uint) * DEFAULT_SIZE);
    memcpy(copyGame->nbTentsColArray, g->nbTentsColArray,
           sizeof(uint) * DEFAULT_SIZE);

    return copyGame;
}

bool game_equal(cgame g1, cgame g2) {
    check_game(g1);
    check_game(g2);

    for (uint i = 0; i < DEFAULT_SIZE; i++) {
        // Checking if each row and each col are equal
        if (game_get_expected_nb_tents_row(g1, i) !=
                game_get_expected_nb_tents_row(g2, i) ||
            game_get_expected_nb_tents_col(g1, i) !=
                game_get_expected_nb_tents_col(g2, i)) {
            return false;
        }

        // Checking if each square is equal
        for (uint j = 0; j < DEFAULT_SIZE; j++) {
            if (game_get_square(g1, i, j) != game_get_square(g2, i, j)) {
                return false;
            }
        }
    }
    return true;
}

void game_delete(game g) {
    // Checking prerequisite
    check_game(g);

    // Freeing all of the pointers inside of g
    free((*g).squareArray);
    free((*g).nbTentsRowArray);
    free((*g).nbTentsColArray);

    // Freeing the pointer g itself
    free(g);

    return;
}

void game_set_square(game g, uint i, uint j, square s) {
    // Checking prerequisites
    check_game(g);

    if (i > DEFAULT_SIZE || j > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameters: an index is greater than DEFAULT_SIZE");
    if (s > 3)
        treat_error("Unvalid parameters: This square type doesn't exist");

    // Setting square in array
```

pourquoi (*g). et g -> ? ou

si le jeu est NULL on est content, Non?

un appel de fonction

pourquoi calloc et pas malloc ?

```
    (*g).squareArray[(DEFAULT_SIZE * i) + j] = s;

    return;
}

square game_get_square(cgame g, uint i, uint j) {
    check_game(g);
    square squareValue = g->squareArray[(DEFAULT_SIZE * i + j)];
    return squareValue;
}

void game_set_expected_nb_tents_row(game g, uint i, uint nb_tents) {
    // Checking prerequisites
    check_game(g);
    if (i > DEFAULT_SIZE)
        treat_error(
            "Invalid parameter: row array is greater than DEFAULT_SIZE");

    // Changing the expected number of tents in row i
    (*g).nbTentsRowArray[i] = nb_tents;
    return;
}

void game_set_expected_nb_tents_col(game g, uint j, uint nb_tents) {
    // Checking prerequisites
    check_game(g);
    if (j > DEFAULT_SIZE)
        treat_error(
            "Invalid parameter: col array is greater than DEFAULT_SIZE");

    // Changing the expected number of tents in row i
    (*g).nbTentsColArray[j] = nb_tents;
    return;
}

uint game_get_expected_nb_tents_row(cgame g, uint i) {
    check_game(g);
    if (i > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameter: row index is greater than DEFAULT_SIZE");
    return g->nbTentsRowArray[i];
}

uint game_get_expected_nb_tents_col(cgame g, uint j) {
    check_game(g);
    if (j > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameter: col index is greater than DEFAULT_SIZE");
    return g->nbTentsColArray[j];
}

uint game_get_expected_nb_tents_all(cgame g) {
    check_game(g);
    uint allExpectedTent = 0;
    for (uint counter = 0; counter < DEFAULT_SIZE; counter++) {
        allExpectedTent += game_get_expected_nb_tents_row(g, counter);
    }
    return allExpectedTent;
}

uint game_get_current_nb_tents_row(cgame g, uint i) {
    check_game(g);
    if (i > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameter: row index is greater than DEFAULT_SIZE");

    uint nbtents = 0;
    for (uint j = 0; j < DEFAULT_SIZE; j++) {
        if (game_get_square(g, i, j) == TENT) {
```



```
            nbtents++;
        }
    }
    return nbtents;
}

uint game_get_current_nb_tents_col(cgame g, uint j) {
    check_game(g);
    if (j > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameter: col index is greater than DEFAULT_SIZE");

    uint nbtents = 0;
    for (uint i = 0; i < DEFAULT_SIZE; i++) {
        if (game_get_square(g, i, j) == TENT) {
            nbtents++;
        }
    }
    return nbtents;
}

uint game_get_current_nb_tents_all(cgame g) {
    check_game(g);
    uint nbtents = 0;
    for (uint i = 0; i < DEFAULT_SIZE; i++) {
        nbtents += game_get_current_nb_tents_row(g, i);
    }
    return nbtents;
}

void game_play_move(game g, uint i, uint j, square s) {
    // Checking prerequisites
    check_game(g);

    if (i > DEFAULT_SIZE || j > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameters: an index is greater than DEFAULT_SIZE");
    if (s > 3)
        treat_error("Unvalid parameters: This square type doesn't exist");

    if (game_get_square(g, i, j) == TREE) return;

    // Playing the move
    game_set_square(g, i, j, s);
    return;
}

int game_check_move(cgame g, uint i, uint j, square s) {
    check_game(g);

    if (game_get_square(g, i, j) == TREE || s == TREE) return ILLEGAL;

    // Checking if putting a grass square makes it impossible to win or not
    if (s == GRASS) {
        int emptySquaresLeft = 0;

        // Checking row i
        for (int k = 0; k < DEFAULT_SIZE; k++) {
            if (k != i && game_get_square(g, i, k) == EMPTY) emptySquaresLeft++;
        }
        if (game_get_expected_nb_tents_row(g, i) > emptySquaresLeft)
            return LOSING;

        // Checking col j
        emptySquaresLeft = 0;
        for (int k = 0; k < DEFAULT_SIZE; k++) {
            if (k != j && game_get_square(g, k, j) == EMPTY) emptySquaresLeft++;
        }
        if (game_get_expected_nb_tents_col(g, j) > emptySquaresLeft)
```

```
            return LOSING;
    }

    if (s == TENT) {
        // Rule 1 + 4

        bool hasTree = false;
        // Checking the row before and after row i for diagonal + orthogonal
        // check
        for (int i_test = -1; i_test <= 1; i_test++) {
            // Checking if the row doesn't exist
            if (i + i_test >= 0 && i + i_test < DEFAULT_SIZE) {
                // Checking the col before and after row i for diagonal +
                // orthogonal check
                for (int j_test = -1; j_test <= 1; j_test++) {
                    // Checking if the col doesn't exist
                    if (j + j_test >= 0 && j + j_test < DEFAULT_SIZE) {
                        // Do not do the check the square of the move
                        if (i_test != 0 || j_test != 0) {
                            // Rule 1
                            if (game_get_square(g, i + i_test, j + j_test) ==
                                TENT)
                                return LOSING;
                            // Rule 4
                            if ((i_test == 0 || j_test == 0) &&
                                game_get_square(g, i + i_test, j + j_test) ==
                                    TREE)
                                hasTree = true;
                        }
                    }
                }
            }
        }
        // No tree orthogonally
        if (hasTree == false) return LOSING;

        // The 2 following checks assume the player isn't playing a tent on a
        // tent, however if it is the case, then the checks would count 1 tent
        // when it shouldn't (which may result in false positives). The int
        // tentNotPlaced counteracts this issue
        uint tentNotPlaced = 1;
        if (game_get_square(g, i, j) == TENT) {
            tentNotPlaced = 0;
        }

        // Rule 2
        if ((game_get_current_nb_tents_row(g, i) + tentNotPlaced >
             game_get_expected_nb_tents_row(g, i)) ||
            (game_get_current_nb_tents_col(g, j) + tentNotPlaced >
             game_get_expected_nb_tents_col(g, j)))
            return LOSING;

        // Rule 3
        if (game_get_current_nb_tents_all(g) + tentNotPlaced >
            game_get_expected_nb_tents_all(g))
            return LOSING;
    }

    return REGULAR;
}

bool game_is_over(cgame g) {
    bool res = true;
    bool treeIsHere;
    for (uint i = 0; i < DEFAULT_SIZE * DEFAULT_SIZE; i++) {
        if (g->squareArray[i] == TENT) {
            // 1rd condition : no tent around
            // Left
            if (i % DEFAULT_SIZE != 0) {
```



```
                if (g->squareArray[i - 1] == TENT) {
                    res = false;
                }
            }
            // Right
            if (i % DEFAULT_SIZE != DEFAULT_SIZE - 1) {
                if (g->squareArray[i + 1] == TENT) {
                    res = false;
                }
            }
            // Up
            if (i >= DEFAULT_SIZE) {
                if (g->squareArray[i - DEFAULT_SIZE] == TENT) {
                    res = false;
                }
            }
            // Down
            if (i < DEFAULT_SIZE * (DEFAULT_SIZE - 1)) {
                if (g->squareArray[i + DEFAULT_SIZE] == TENT) {
                    res = false;
                }
            }
            // Up-left
            if (i % DEFAULT_SIZE != 0 && i >= DEFAULT_SIZE) {
                if (g->squareArray[i - 1 - DEFAULT_SIZE] == TENT) {
                    res = false;
                }
            }
            // Up-right
            if (i % DEFAULT_SIZE != DEFAULT_SIZE - 1 && i >= DEFAULT_SIZE) {
                if (g->squareArray[i + 1 - DEFAULT_SIZE] == TENT) {
                    res = false;
                }
            }
            // Down-left
            if (i % DEFAULT_SIZE != 0 &&
                i < DEFAULT_SIZE * (DEFAULT_SIZE - 1)) {
                if (g->squareArray[i - 1 + DEFAULT_SIZE] == TENT) {
                    res = false;
                }
            }
            // Down-Right
            if (i % DEFAULT_SIZE != DEFAULT_SIZE - 1 &&
                i < DEFAULT_SIZE * (DEFAULT_SIZE - 1)) {
                if (g->squareArray[i + 1 + DEFAULT_SIZE] == TENT) {
                    res = false;
                }
            }
            /* Older code
            if (g->squareArray[i-1] == TENT
                || g->squareArray[i-1-DEFAULT_SIZE] == TENT
                || g->squareArray[i-DEFAULT_SIZE] == TENT
                || g->squareArray[i+1-DEFAULT_SIZE] == TENT
                || g->squareArray[i+1] == TENT
                || g->squareArray[i+1+DEFAULT_SIZE] == TENT
                || g->squareArray[i+DEFAULT_SIZE] == TENT
                || g->squareArray[i-1+DEFAULT_SIZE] == TENT){
                    res = false;
            }
            */

            // 4th condition : at least a tree horizontally or vertically
            treeIsHere = false;
            // Horizontal
            for (uint h = 0; h < DEFAULT_SIZE; h++) {
                if (g->squareArray[i / DEFAULT_SIZE + h] == TREE) {
                    treeIsHere = true;
                }
            }
```

```
            // Vertical
            for (uint v = 0; v < DEFAULT_SIZE; v++) {
                if (g->squareArray[i % DEFAULT_SIZE + v * DEFAULT_SIZE] ==
                    TREE) {
                    treeIsHere = true;
                }
            }
            // Checking
            if (!treeIsHere) {
                res = false;
            }
        }
    }

    // 2nd && 3rd condition : tents number per row and col then total number of
    // tents
    for (uint c = 0; c < DEFAULT_SIZE; c++) {
        if (game_get_expected_nb_tents_row(g, c) !=
                game_get_current_nb_tents_row(g, c) ||
            game_get_expected_nb_tents_col(g, c) !=
                game_get_current_nb_tents_col(g, c) ||
            game_get_expected_nb_tents_all(g) !=
                game_get_current_nb_tents_all(g)) {
            res = false;
        }
    }
    return res;
}

void game_fill_grass_row(game g, uint i) {
    check_game(g);
    if (i > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameters: row index is greater than DEFAULT_SIZE");

    for (uint j = 0; j < DEFAULT_SIZE; j++) {
        if (game_get_square(g, i, j) == EMPTY) {
            game_set_square(g, i, j, GRASS);
        }
    }
    return;
}

void game_fill_grass_col(game g, uint j) {
    check_game(g);
    if (j > DEFAULT_SIZE)
        treat_error(
            "Unvalid parameters: col index is greater than DEFAULT_SIZE");

    for (uint i = 0; i < DEFAULT_SIZE; i++) {
        if (game_get_square(g, i, j) == EMPTY) {
            game_set_square(g, i, j, GRASS);
        }
    }
    return;
}

void game_restart(game g) {
    check_game(g);

    for (int i = 0; i < DEFAULT_SIZE; i++) {
        for (int j = 0; j < DEFAULT_SIZE; j++) {
            if (game_get_square(g, i, j) != TREE) {
                game_set_square(g, i, j, EMPTY);
            }
        }
    }
}
```

Admettons que (i, j) soient des coordonnées illégales (en dehors du jeu) alors on peut considérer qu'il n'y a ni tente, ni arbre en (i, j) -

il me semble qu'on peut exploiter cette propriété pour simplifier grandement votre code -