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

struct game_s {
  square *square;
  uint *nb_tent_col;
  uint *nb_tent_row;
};

// test
void memory_error(char *s) {
  fprintf(stderr, "Erreur %s mÃ©moire\n", s);
  exit(EXIT_FAILURE);
}

void game_is_null(cgame g) {
  if (g == NULL) {
    fprintf(stderr, "Error : Invalid parameter \n");
    exit(EXIT_FAILURE);
  }
}
void size_error(char *s) {
  fprintf(stderr, "Invalide %s !\n", s);
  exit(EXIT_FAILURE);
}

unsigned int error_other(unsigned int *element) {
  if (element == NULL) {
    fprintf(stderr, "error malloc \n");
    return true;
  }
  return false;
}
// DEBUT

game game_new(square *square, uint *nb_tents_row, uint *nb_tents_col) {
  // farah
  if (nb_tents_row == NULL || square == NULL || nb_tents_col == NULL) {
    fprintf(stderr, "error Memory \n");
    exit(EXIT_FAILURE);
  }
  game new = game_new_empty();
  if (new == NULL) memory_error("allocation 1");

  for (uint i = 0; i < (DEFAULT_SIZE * DEFAULT_SIZE); i++)
    new->square[i] = square[i];
  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    new->nb_tent_col[i] = nb_tents_col[i];
    new->nb_tent_row[i] = nb_tents_row[i];
  }
  return new;
}

game game_new_empty(void) {
  game new = malloc(sizeof(struct game_s));

  game_is_null(new);

  new->square =
      (square *)malloc((DEFAULT_SIZE * DEFAULT_SIZE) * sizeof(square));
  new->nb_tent_col = (uint *)malloc((DEFAULT_SIZE) * sizeof(uint));
  new->nb_tent_row = (uint *)malloc((DEFAULT_SIZE) * sizeof(uint));
  if (new->square == NULL || new->nb_tent_col == NULL ||
      new->nb_tent_row == NULL) {
    fprintf(stderr, "Error : Invalid parameter \n");
    exit(EXIT_FAILURE);
```



```c
  }

  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    for (uint j = 0; j < DEFAULT_SIZE; j++) {
      new->nb_tent_col[j] = 0;
      new->nb_tent_row[i] = 0;
      new->square[i * DEFAULT_SIZE + j] = EMPTY;
    }
  }

  return new;
}

game game_copy(cgame g) {
  // farah
  game_is_null(g);
  game game_copied = game_new_empty();

  for (uint i = 0; i < DEFAULT_SIZE * DEFAULT_SIZE; i++)
    game_copied->square[i] = g->square[i];

  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    game_copied->nb_tent_col[i] = g->nb_tent_col[i];
    game_copied->nb_tent_row[i] = g->nb_tent_row[i];
  }

  return game_copied;
}

/*bool game_equal(cgame g1, cgame g2) {
  game_is_null(g1);
  game_is_null(g2);

  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    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;
}*/

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

  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    for (uint j = 0; j < DEFAULT_SIZE; j++) {
      if ((game_get_square(g1, i, j) != game_get_square(g2, i, j)))
        return false;
      if ((game_get_expected_nb_tents_col(g1, j)) !=
          game_get_expected_nb_tents_col(g2, j))
        return false;
      if ((game_get_expected_nb_tents_row(g1, i)) !=
          game_get_expected_nb_tents_row(g2, i))
        return false;
    }
  }

  return true;
}

void game_delete(game g) {
  // farah
  if (g == NULL || g->square == NULL || g->nb_tent_col == NULL ||
      g->nb_tent_row == NULL) {
    fprintf(stderr, "error\n");
    exit(EXIT_FAILURE);
```

```
  }
  free(g->square);
  free(g->nb_tent_row);
  free(g->nb_tent_col);
  free(g);
}

//
//

uint game_get_expected_nb_tents_col(cgame g, uint j) {
  game_is_null(g);

  if (j >= DEFAULT_SIZE) {
    size_error("size");
  }

  return g->nb_tent_col[j];
}

uint game_get_expected_nb_tents_row(cgame g, uint i) {
  game_is_null(g);

  if (i >= DEFAULT_SIZE) {
    size_error("size");
  }

  return g->nb_tent_row[i];
}

uint game_get_expected_nb_tents_all(cgame g) {
  game_is_null(g);

  uint nb_tents_all = 0;
  uint i, j;
  for (i = 0; i < DEFAULT_SIZE; i++) {
    for (j = 0; j < DEFAULT_SIZE; j++) {
      if (g->square[i * DEFAULT_SIZE + j] == TENT) {
        nb_tents_all = nb_tents_all + 1;
      }
    }
  }
  return nb_tents_all;
}

void game_set_expected_nb_tents_row(game g, uint i, uint nb_tents) {
  game_is_null(g);

  if (i >= DEFAULT_SIZE) {
    size_error("size");
  }

  g->nb_tent_row[i] = nb_tents;
}

void game_set_expected_nb_tents_col(game g, uint j, uint nb_tents) {
  game_is_null(g);

  if (j >= DEFAULT_SIZE) {
    size_error("size");
  }

  g->nb_tent_col[j] = nb_tents;
}

//

void game_play_move(game g, uint i, uint j, square s) {
  game_is_null(g);
```



```
  if (i >= DEFAULT_SIZE || j >= DEFAULT_SIZE) {
    size_error("size");
  }

  if (game_check_move(g, i, j, s) == ILLEGAL || s == TREE) {
    printf("Illegal move!!! You can not do that!\n");
  }

  game_set_square(g, i, j, s);
}

//

void game_set_square(game g, uint i, uint j, square s) {
  game_is_null(g);

  if (i >= DEFAULT_SIZE || j >= DEFAULT_SIZE) size_error("size");

  g->square[i * DEFAULT_SIZE + j] = s;
}

square game_get_square(cgame g, uint i, uint j) {
  game_is_null(g);

  if (i >= DEFAULT_SIZE || j >= DEFAULT_SIZE) size_error("size");

  return g->square[i * DEFAULT_SIZE + j];
}

uint game_get_current_nb_tents_all(cgame g) {
  game_is_null(g);

  uint nb_tents_all = 0;

  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    for (uint j = 0; j < DEFAULT_SIZE; j++) {
      if (g->square[i * DEFAULT_SIZE + j] == TENT)
        nb_tents_all = nb_tents_all + 1;
    }
  }

  return nb_tents_all;
}

uint game_get_current_nb_tents_row(cgame g, uint i) {
  game_is_null(g);

  if (i >= DEFAULT_SIZE) size_error("size");
  uint nb_tents_row = 0;

  for (uint j = 0; j < DEFAULT_SIZE; j++) {
    if (g->square[i * DEFAULT_SIZE + j] == TENT)
      nb_tents_row = nb_tents_row + 1;
  }
  return nb_tents_row;
}

uint game_get_current_nb_tents_col(cgame g, uint j) {
  game_is_null(g);

  if (j >= DEFAULT_SIZE) size_error("size");

  uint nb_tents_col = 0;

  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    if (game_get_square(g, i, j) == TENT) nb_tents_col = nb_tents_col + 1;
  }
  return nb_tents_col;
```

```
}

//
//
// EMMANUEL

/*int game_check_move(cgame g, uint i, uint j, square s) {
  cgame G = game_default_solution();
  game_is_null(g);
  game_is_null(G);
  if (s == game_get_square(G, i, j)) return REGULAR;
  if (game_get_square(g, i, j) == TREE) return ILLEGAL;
  return LOSING;
}*/
int game_check_move(cgame g, uint i, uint j, square s) {
  uint tab_ajout_i[8] = {-1, -1, -1, 0, 0, +1, +1, +1};
  uint tab_ajout_j[8] = {-1, 0, +1, -1, +1, -1, 0, +1};

  if (i < DEFAULT_SIZE && g != NULL && j < DEFAULT_SIZE &&
      (s == EMPTY || s == GRASS || s == TENT || s == TREE)) {
    if (s == TREE || game_get_square(g, i, j) ==
                     TREE) {  /////////////// TEST ILLEGAL (mouvement TREE)
      return ILLEGAL;
    }

    /// TESTS TENTS QUI SE TOUCHENT
    // uint v = 0;
    if (i == 0) {
      if (j == 0) {
        if (s == TENT && (game_get_square(g, i + 1, j) == TENT ||
                          game_get_square(g, i, j + 1) == TENT)) {
          return LOSING;
        }
      } else if (j == DEFAULT_SIZE) {
        if (s == TENT && (game_get_square(g, i + 1, j) == TENT ||
                          game_get_square(g, i, j - 1) == TENT)) {
          return LOSING;
        }
      } else {
        if (s == TENT && (game_get_square(g, i + 1, j) == TENT ||
                          game_get_square(g, i, j - 1) == TENT ||
                          game_get_square(g, i, j + 1) == TENT)) {
          return LOSING;
        }
      }
    } else if (i == DEFAULT_SIZE) {
      if (j == 0) {
        if (s == TENT && (game_get_square(g, i - 1, j) == TENT ||
                          game_get_square(g, i, j + 1) == TENT)) {
          return LOSING;
        }
      } else if (j == DEFAULT_SIZE) {
        if (s == TENT && (game_get_square(g, i - 1, j) == TENT ||
                          game_get_square(g, i, j - 1) == TENT)) {
          return LOSING;
        }
      } else {
        if (s == TENT && (game_get_square(g, i - 1, j) == TENT ||
                          game_get_square(g, i, j - 1) == TENT ||
                          game_get_square(g, i, j + 1) == TENT)) {
          return LOSING;
        }
      }
    } else {
      if (j == 0) {
        if (s == TENT && (game_get_square(g, i - 1, j) == TENT ||
                          game_get_square(g, i + 1, j) == TENT ||
                          game_get_square(g, i, j + 1) == TENT)) {
          return LOSING;
```

trop de cas ... (voir fin)


```
        }
      } else if (j == DEFAULT_SIZE) {
        if (s == TENT && (game_get_square(g, i - 1, j) == TENT ||
                          game_get_square(g, i + 1, j) == TENT ||
                          game_get_square(g, i, j - 1) == TENT)) {
          return LOSING;
        }
      } else {
        if (s == TENT && (game_get_square(g, i - 1, j) == TENT ||
                          game_get_square(g, i + 1, j) == TENT ||
                          game_get_square(g, i, j - 1) == TENT ||
                          game_get_square(g, i, j + 1) == TENT)) {
          return LOSING;
        }
      }
    }

    /// TESTS NB TENTS TOTAL
    if (game_get_current_nb_tents_all(g) > game_get_expected_nb_tents_all(g) ||
        (game_get_current_nb_tents_all(g) ==
             game_get_expected_nb_tents_all(g) &&
         s == TENT)) {
      return LOSING;
    }

    /// TEST NB TENTS ROW/COL
    uint cx = 0;
    uint cy = 0;
    for (uint x = 0; x < DEFAULT_SIZE; x++) {
      if (game_get_square(g, x, j) == EMPTY) {
        cy++;
      }
    }
    for (uint y = 0; y < DEFAULT_SIZE; y++) {
      if (game_get_square(g, i, y) == EMPTY) {
        cx++;
      }
    }
    if (cy + game_get_current_nb_tents_col(g, j) <
            game_get_expected_nb_tents_col(g, j) ||
        cx + game_get_current_nb_tents_row(g, i) <
            game_get_expected_nb_tents_row(g, i)) {
      return LOSING;
    }
    // check si ya un tent adjacent

    for (uint p = 0; p < 8; p++) {
      if (i + tab_ajout_i[p] < DEFAULT_SIZE &&
          j + tab_ajout_j[p] < DEFAULT_SIZE)
        if (game_get_square(g, i + tab_ajout_i[p], tab_ajout_j[p]) == TENT)
          return LOSING;
    }
    /// TEST LOSING
    /*if (v != 0) {
      return LOSING;
    }*/
    /// SI NI LOSING NI ILLEGAL -> REGULAR
    return REGULAR;
  }
  exit(EXIT_FAILURE);
}

/*bool game_is_over(cgame g) {
  cgame G = game_default_solution();
  game_is_null(g);
  game_is_null(G);
  return (game_equal(g, G));
}*/
bool game_is_over(cgame g) {
```

```
  game_is_null(g);

  uint t_row[DEFAULT_SIZE] = {3, 0, 4, 0,
                              4, 0, 1, 0};  // initialisation de la ligne
  uint t_col[DEFAULT_SIZE] = {4, 0, 1, 2, 1, 1, 2, 1};

  uint nb_tree = 0;
  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    for (uint j = 0; j < DEFAULT_SIZE; j++) {
      if (game_get_square(g, i, j) == TREE) nb_tree += nb_tree;
      if (game_get_expected_nb_tents_col(g, j) != t_col[j]) return false;
      if (game_get_expected_nb_tents_row(g, i) != t_row[i]) return false;
      if (game_get_square(g, i, j) != TREE) {
        if (game_check_move(g, i, j, game_get_square(g, i, j)) != REGULAR)
          return false;
      }
    }
  }

  if (game_get_expected_nb_tents_all(g) != nb_tree) return false;

  return true;
}

void game_fill_grass_row(game g, uint i) {
  game_is_null(g);
  if (i < 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);
    }
  }
}

void game_fill_grass_col(game g, uint j) {
  game_is_null(g);
  if (j < 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);
    }
  }
}

void game_restart(game g) {
  game_is_null(g);
  //  g = game_default();
  for (uint i = 0; i < DEFAULT_SIZE; i++) {
    for (uint j = 0; j < DEFAULT_SIZE; j++) {
      if ((game_get_square(g, i, j) != TREE)) game_set_square(g, i, j, EMPTY);
    }
  }
}
//
```

Pour simplifier le code:

définir (par exemple) une fonction qui permet de répondre à la question "y a-t-il une tente - par exemple - en position (i, j) ?" si (i, j) n'est pas un couple de coordonnées valides alors il n'y a pas de tente en (i, j)

⟹ il faut chercher à simplifier le code éliminer les redondances