```c
/**
* @file colorsBack.c
* Command handling
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include "error.h"
#include "utils.h"
#include "defines.h"
#include "playGame.h"
#include "colorsBack.h"

bool movePiece ( game_t * game, int argc, char ** argv, char * msg );
bool save      ( game_t * game, int argc, char ** argv, char * msg );
bool undo      ( game_t * game, int argc, char ** argv, char * msg );
bool quit      ( game_t * game, int argc, char ** argv, char * msg );
bool help      ( game_t * game, int argc, char ** argv, char * msg );
bool roflcopter( game_t * game, int argc, char ** argv, char * msg );

typedef struct{
    char com[ MAX_COM_LEN ];
    bool (* func )( game_t *, int, char **, char * );
} command_t;

const command_t commands[] = {
    {"[", movePiece},
    {"save", save},
    {"undo", undo},
    {"quit", quit},
    {"help", help},
    {"ROFLcopter",roflcopter}
};


/**
* Parses the command and calls the corresponding function.
*
* @param game       contains all information about current game
* @param s          containins the command line about to be processed
* @param[out] msg   its an output containing the type of error
*
* @return false if there is an error, otherwise true
*
* @see editDistance()
* @see movePiece()
* @see save()
* @see undo()
* @see quit()
* @see help()
* @see ROFLcopter()
*/

bool
newCommand( game_t * game, const char * s, char * msg )
{
    int i;
    bool sol;
    int argc;
    char ** argv;

    if( errorCode() != NOERROR ){
        sprintf( msg, "%s", errorMessage( errorCode() ) );
        return false;
    }

    argv = newMatrix( MAX_ARGS, MAX_COM_LEN );

    if( errorCode() != NOERROR ){
        sprintf( msg, "%s", errorMessage( errorCode() ) );
        return false;
    }

    msg[0] = 0;
    // parse the command
    argc = sscanf( s, "%s %s %s %s %s %s %s %s %s %s",
                   argv[0], argv[1], argv[2], argv[3], argv[4],
                   argv[5], argv[6], argv[7], argv[8], argv[9] );

    if( argc < 1  ){
        freeMatrix( argv, MAX_ARGS );
        return true;
    }

    sol = false;
    sprintf( msg, "Unknown command" );
    double auxsim, maxsim = 0;
    int maxi = 0;

    //call the appropiate function
    for( i = 0 ; i < sizeof(commands)/ sizeof(command_t) ; i++ ){
        if( strncmp( argv[0], commands[i].com, strlen(commands[i].com) ) == 0
                         && !isalpha( argv[0][strlen(commands[i].com)] ) ){
            maxsim = 0;
            msg[0] = 0;
            sol = commands[i].func( game, argc, argv, msg );
            if( errorCode() != NOERROR ){
                sprintf( msg, "%s", errorMessage( errorCode() ) );
                sol = false;
            }
            break;
        }
        // check for command similarity
        if( ( auxsim = editDistance( argv[0], commands[i].com ) ) > maxsim ){
            maxsim = auxsim;
            maxi = i;
        }
    }
    // if not succesful and there was a command similar enough
    if( maxsim >= MIN_SIMILARITY )
        sprintf( msg+15, "\nDid you mean: \"%s\"", commands[maxi].com );

    freeMatrix( argv, MAX_ARGS );

    if( errorCode() != NOERROR ){
        sprintf( msg, "%s", errorMessage( errorCode() ) );
        sol = false;
    }

    return sol;
}


```

```c
/**
* Checks if there is a valid path between (@a x1,@a y1) and (@a x2,@a y2)
* in the board.
*
* @param game   contains all information about current game
* @param x1     initial x coordenate
* @param y1     initial y coordenate
* @param x2     final x coordenate
* @param y2     final y coordenate
*
* @return true if there is a valid path, otherwise false
*/

bool
areConnected( game_t * game, int x1, int y1, int x2, int y2 )
{
    // BFS to find minimum path
    struct coord{
        int x,y;
    } move[4] = { {-1,0}, {0,1}, {1,0}, {0,-1} };

    struct node{
        int x, y;
    } queue[ game->players[ game->state.next ].board.emptySpots + 1 ];

    int read = -1, write = 0, x, y, i;

    bool touched[ game->options.height ][ game->options.width ];

    memset( &touched[0][0], false, sizeof(touched) * sizeof(bool) );
    queue[write++] = (struct node){x1,y1};

    while( ++read < write ){
        if( queue[read].x == x2 && queue[read].y == y2 )
            break;
        for( i = 0 ; i < sizeof(move)/sizeof(struct coord) ; i++ ){
            x = queue[read].x + move[i].x;
            y = queue[read].y + move[i].y;
            if( entre( 0, x, game->options.width )
                && entre( 0, y, game->options.height )
                && game->players[ game->state.next ].board.matrix[y][x] == 0
                && !touched[y][x] ){
                        touched[y][x] = true;
                        queue[write++] = (struct node){x,y};
            }
        }
    }
    return read < write;
}


/**
* Moves token from a position to another checking for winning plays (lines).
*
* @param game        contains all information about current game
* @param argc        size of @a argv
* @param argv        parameters followinf the command
* @param[out] msg    output with message to write in the panel
*
* @return false if some message is to be written
*
* @see areConected()
* @see randFill()
* @see winningPlay()
*/

bool
movePiece( game_t * game, int argc, char ** argv, char * msg )
{
    int i;
    char s[ argc * MAX_ARGS ];

    s[0] = 0;
    for( i = 0 ; i < argc ; i++ )
        strcat( s, argv[i] );

    int x1, y1, x2, y2;
    i = sscanf( s, "[%d,%d][%d,%d]", &y1, &x1, &y2, &x2 );
    if( i < 4 ){
        sprintf( msg, "Format error:\n"
                    "Must be: [ row_1, column_1 ][ row_2, column_2 ]" );
        return false;
    }
    if( ! entre( 0, y1, game->options.height ) ){
        sprintf( msg, "Rank error:\nThe first row must belong to the "
                    "interval [0,%d]", game->options.height - 1 );
        return false;
    }
    if( ! entre( 0, x1, game->options.width ) ){
        sprintf( msg, "Rank error:\nThe first column must belong to the "
                    "interval [0,%d]", game->options.width - 1 );
        return false;
    }
    if( ! entre( 0, y2, game->options.height ) ){
        sprintf( msg, "Rank error:\nThe second row must belong to the "
                    "interval [0,%d]", game->options.height - 1 );
        return false;
    }
    if( ! entre( 0, x2, game->options.width ) ){
        sprintf( msg, "Rank error:\nThe second column must belong to the "
                    "interval [0,%d]", game->options.width - 1 );
        return false;
    }
    if( game->players[ game->state.next ].board.matrix[y1][x1] == 0 ){
        sprintf( msg, "The origin position must not be empty" );
        return false;
    }
    if( game->players[ game->state.next ].board.matrix[y2][x2] != 0 ){
        sprintf( msg, "The target position must not be occupied" );
        return false;
    }
    if( !areConnected( game, x1, y1, x2, y2 ) ){
        sprintf( msg, "There must be a path of unoccupied spaces from the "
                    "origin position to the target position" );
        return false;
    }
    // maintain undo
    game->players[ game->state.next ].canUndo = true;

    copyMatrix( game->players[ game->state.next ].lastBoard. matrix,
                game->players[ game->state.next ].board.matrix,
                game->options.height, game->options.width );

    game->players[ game->state.next ].lastBoard.points =
                                game->players[ game->state.next ].board.points;

    game->players[ game->state.next ].lastBoard.emptySpots =
                            game->players[ game->state.next ].board.emptySpots;
    // maintain board
    game->players[ game->state.next ].board.matrix[y2][x2] =
                    game->players[ game->state.next ].board.matrix[y1][x1];

    game->players[ game->state.next ].board.matrix[y1][x1] = 0;

    // if made a line, erase it and count points (winning play)
    if( ! winningPlay( game, game->state.next, x2, y2, true ) ){
        // else, randFill()
        randFill( game, game->state.next, game->options.tokensPerTurn, false );
        // change turn
        i = 0;
        do{
            game->state.next++;
            game->state.next %= game->numPlayers;
            i++;
        }while( game->players[ game->state.next ].board.emptySpots <= 0 &&
                    i <= game->numPlayers );
    }else{
        // print number of points just made
        sprintf( msg, "%d point/s move",
                        game->players[ game->state.next ].board.points -
                            game->players[ game->state.next ].lastBoard.points );
        // if board is empty
        if( game->players[ game->state.next ].board.emptySpots >=
                                    game->options.width * game->options.height ){

            randFill( game, game->state.next, game->options.tokensPerTurn, true );
            if( errorCode() != NOERROR ){
                sprintf( msg, "%s", errorMessage( errorCode() ) );
                return false;
            }
        }
    }

    return true;
}


/**
* Saves current game to file.
*
* @param game       contains all information about current game
* @param argc       size of @a argv
* @param argv       parameters followinf the command
* @param[out] msg   output with message to write in the panel
*
* @return false if some message is to be written
*
* @see writeGame()
*/

bool
save( game_t * game, int argc, char ** argv, char * msg )
{
    if( argc > 2 || strcmp( "save", argv[0] ) != 0 ){
        sprintf( msg, "Wrong usage\n"
        "Try 'save --help' for more information");
        return false;
    }
    if( argc == 1 ){
        sprintf( msg, "Missing file operand\n"
                      "Try 'save --help' for more information");
        return false;
    }
    if( strcmp( argv[1], "--help" ) == 0 ){
        sprintf( msg, "Usage: save filename\n"
                      "Saves the current game to file 'filename'");
        return false;
    }
    writeGame( game, argv[1] );

    if( errorCode() != NOERROR ){
        sprintf( msg, "%s", errorMessage( errorCode() ) );
        return false;
    }

    return true;
}


/**
* Undoes the last move made.
*
* @param game       contains all information about current game
* @param argc       size of @a argv
* @param argv       parameters followinf the command
* @param[out] msg   output with message to write in the panel
*
* @return false if some message is to be written
*/

bool
undo( game_t * game, int argc, char ** argv, char * msg )
{
    if( argc == 2 && strcmp( argv[1], "--help" ) == 0 ){
        sprintf( msg, "Usage: undo\n"
                      "Undoes the last move\n"
                      "It can only be used once some move has been done, "
                      "and it can't be used two consecutive times" );
        return false;
    }
    if( argc > 1  || strcmp( "undo", argv[0] ) != 0 ){
        sprintf( msg, "Wrong usage\n"
                      "Try 'undo --help' for more information");
        return false;
    }
    if( game->options.mode != SINGLEMODE ){
        sprintf( msg, "'undo' command is only available in one player"
                      ", no time mode" );
        return false;
    }
    if( game->players[ game->state.next ].canUndo == false ){
        sprintf( msg, "'undo' command cannot be used twice in a row or in "
                      "the first turn" );
        return false;
    }
    game->players[ game->state.next ].canUndo = false;
    // swap boards;
    board_t aux = game->players[ game->state.next ].board;

    game->players[ game->state.next ].board =
                                game->players[ game->state.next ].lastBoard;

```

```c
    game->players[ game->state.next ].lastBoard = aux;

    return true;
}


/**
* Quits the game.
*
* @param game       contains all information about current game
* @param argc       size of @a argv
* @param argv       parameters followinf the command
* @param[out] msg   output with message to write in the panel
*
* @return false if some message is to be written
*/

bool
quit( game_t * game, int argc, char ** argv, char * msg )
{
    if( argc == 2 && strcmp( argv[1], "--help" ) == 0 ){
        sprintf( msg, "Usage: quit\n"
                        "Quits the current game" );
        return false;
    }
    if( argc > 1 || strcmp( "quit", argv[0] ) != 0 ){
        sprintf( msg, "Wrong usage\n"
            "Try 'quit --help' for more information");
        return false;
    }
    game->state.quit = true;
    return true;
}


/**
* Prints help for the user. Secret commands are not shown.
*
* @param game       contains all information about current game
* @param argc       size of @a argv
* @param argv       parameters followinf the command
* @param[out] msg   output with message to write in the panel
*
* @return false if some message is to be written
*/

bool
help( game_t * game, int argc, char ** argv, char * msg )
{
    if( argc == 2 && strcmp( argv[1], "--help" ) == 0 ){
        sprintf( msg, "Usage: help\n"
                        "Shows available commands" );
        return false;
    }
    if( argc > 1 || strcmp( "help", argv[0] ) != 0 ){
        sprintf( msg, "Wrong usage\n"
        "Try 'help --help' for more information");
        return false;
    }
    sprintf( msg,
                "Type 'name --help' to find out more about the function 'name'"
                "\n \n"
                "  [row_1,column_1][row_2,column_2]\n"
                "  save filename\n"
                "  undo\n"
                "  quit\n"
                "  help\n" );
    return true;
}


/**
* Secret command.
* Prints a ROFLcopter.
*
* @param game       contains all information about current game
* @param argc       size of @a argv
* @param argv       parameters followinf the command
* @param[out] msg   output with message to write in the panel
*
* @return false if some message is to be written
*/

bool
roflcopter( game_t * game, int argc, char ** argv, char * msg )
{
    sprintf( msg,
#ifdef __unix__
            "\033[5mROFL:ROFL\033[25m:LOL:\033[5mROFL:ROFL\033[25m\n"
            "           |\n"
            " \033[5mL\033[25m   /---------\n"
            "\033[5mL\033[25mO\033[5mL\033[25m===       []\\\n"
            " \033[5mL\033[25m    \\         \\\n"
            "       \\_________\\\n"
            "         |     |\n"
            "      -------------/\n"
#else
            ".............<ROFL ROFL ROFL ROFL>.\n"
            ".......................| |.........\n"
            "................... __\\||/___......\n"
            ".\\\\...............|'-|-|.\\\\...\\....\n"
            "..\\ \\_...........|--|--|..\\\\...\\...\n"
            "../ L \\______,/---------\\___\\___\\..\n"
            ".|L O L|------------O------ ----,\\.\n"
            "..\\ L /________,---''----------, /.\n"
            "../ /...........\\____________ ,/...\n"
            ".//............\\___//__/\\__\\\\__/...\n"
#endif
            );

    return false;
}

```

```c
/**
* @file colorsBack.h
* Command handling
*/

#ifndef COLORSBACK_H
#define COLORSBACK_H

#include <stdbool.h>
#include "game.h"


bool
newCommand( game_t * game, const char * s, char * msg );


#endif // COLORSBACK_H

```

```c
/**
* @file colors.c
* Contains console color functions depending on the OS.
*/

#include "stdbool.h"
#include "defines.h"
#include "colors.h"

#if defined(__unix__)

    #include "unixColors.c"

#elif defined(__win32__)   || defined(__WIN32__) || \
      defined(win32)       || defined(WIN32)     || \
      defined(__win32)     || defined(__WIN32)   || \
      defined(__windows__) || defined(__WINDOWS__)

    #include "winColors.c"

#else

    #include "noColors.c"

#endif

```

```c
/**
* @file colorsFront.c
* Contains the main function with the game loop.
*/

#include <stdlib.h>
#include <stdbool.h>
#include "error.h"
#include "defines.h"
#include "game.h"
#include "colorsBack.h"
#include "menu.h"
#include "userInterface.h"
#include "playGame.h"


/**
* This function is actually the hole game loop. It takes care of calling every
* other function, including the ones on the frontend, which print everything,
* and the ones on the back end, which process everything.
*
* @return a code to be handled by the operating system depending of the state
*         of the program
*
* @see menu()
* @see drawTable()
* @see drawText()
* @see drawPanel()
* @see askCommand()
* @see newCommand()
*/

int
main()
{
    game_t * game;
    char command[ MAX_COM_LEN ];
    char message[ MAX_ERR_LEN ];

    while( menu( &game ) ){
        if( errorCode() != NOERROR ){
            drawPanel( errorMessage( errorCode() ) );
            continue;
        }
        message[0]=0;
        while( !game->state.quit && !gameOver( game, game->state.next ) ){
            clearError();
            clearScreen();
            drawTable( game );
            drawPanel( message );
            askCommand( command );
            newCommand( game, command, message );
        }
        if( gameOver( game, game->state.next ) ){
            drawWinner( game );
            clearScreen();
            drawTable( game );
            drawPanel("GAME OVER\nPress ENTER to return to main menu\n");
            askCommand( command );
        }
        freeGame( game );
    }
    return EXIT_SUCCESS;
}

```

```c
/**
* @file colors.h
* Contains console color functions depending on the OS.
*/

#ifndef COLORS_H
#define COLORS_H


typedef enum {
    BLACK         ,
    RED           ,
    GREEN         ,
    BROWN         ,
    BLUE          ,
    VIOLET        ,
    SKY_BLUE      ,
    LIGHT_GREY    ,
    GRAY          ,
    PINK          ,
    LIGHT_GREEN   ,
    YELLOW        ,
    LIGHT_BLUE    ,
    LIGHT_VIOLET  ,
    LIGHT_SKY_BLUE,
    WHITE

} color;

typedef enum {
    CLEAR        ,
    NONE         ,

    BOLD         ,
    UNDERLINE    ,
    BLINK        ,
    INVERTED     ,

    NO_BOLD      ,
    NO_UNDERLINE ,
    NO_BLINK     ,
    NO_INVERTED

} attr;

void
textColor( color c );

void
backColor( color c );

void
textAttr( attr a );


#endif // COLORS_H

```

```c
/**
* @file defines.h
* Contains several defines
*/

#ifndef DEFINES_H
#define DEFINES_H



/// maximum error length
#define MAX_ERR_LEN 1000
/// maximum command length
#define MAX_COM_LEN 100
/// maximum arguments to a command (counting the command)
#define MAX_ARGS 10


/// maximum main text lenght
#define MAX_TEXT 5000
/// maximum panel lines (a.k.a. commands history)
#define MAX_PANEL_LINES 25
/// actual number of panel lines @see drawPanel()
#define PANEL_LINES 10 // may be a cool variable


/// horizontal board lines enabled @see drawTable()
#define HOR_LINES true // may be a cool variable


/// colors enablead @see colors.c
#define USE_COLORS true // may be a cool variable


/// input options @see menu.c
#define MAX_MINUTES 10000 // Max playing time: 1 week: 7 * 24 * 60 = 10080
#define MIN_TAB_DIM 5
#define MAX_TAB_DIM 150 // I don't think you have a bigger monitor
#define MIN_COLORS 2
#define MAX_COLORS 9
#define MIN_TOK_PER_LINE 3


/// edit distance options @see editDistance()
#define MIN_SIMILARITY 0.65
#define MIN_EDIT_LEN 3
#define MAX_EDIT_LEN 35


/// maximum processing time for {@link randFill()}
#define MAX_WAITING_TIME 15



#endif // DEFINES_H

```

```
/**
* @file error.c
* Error handling
*/

#include "error.h"

static error nError;


/**
* Clears errors.
*
* @see raiseError()
*/

void
clearError()
{
    nError = NOERROR;
}


/**
* Returns the current error code.
*
* @return error code
*
* @see raiseError()
*/

error
errorCode()
{
    return nError;
}


/**
* Raise an error.
*
* @param num    error code
*/

void
raiseError( error num )
{
    nError = num;
}


/**
* Returns the error message corresponding to the given error code.
*
* @param  error error code
*
* @return error message
*
* @see raiseError();
*/

char *
errorMessage( error num )
```

```c
{
    switch(num){
        case NOERROR:           return "No error";
        case ARITHMETICERROR:   return "Arithmetic error";
        case MEMORYERROR:       return "Memory error";
        case FILEERROR:         return "Error with file";
        case TIMEERROR:         return "Error while geting time";
        case INPUTERROR:        return "Error while reading input";
        case CORRUPTFILE:       return "Input file was corrupted";
        case COMPUTATIONALERROR:return "Computer power is not enough";

        default:                return "Unknown error";
    }
}

```

```c
/**
* @file error.h
* Error handling
*/

#ifndef ERROR_H
#define ERROR_H

// if DEBUG is defined, raise_error_if prints in screen the error, file,
// function & line
// #define DEBUG


// Error types
typedef enum{
    NOERROR,
    ARITHMETICERROR,
    MEMORYERROR,
    FILEERROR,
    TIMEERROR,
    INPUTERROR,
    CORRUPTFILE,
    COMPUTATIONALERROR
} error;

void
clearError();

error
errorCode();

void
raiseError( error num );

char *
errorMessage( error num );

// if "comp" is TRUE raise error "num" and return "ret"
#define raiseErrorIf( comp, num, ret ) \
            do{ if(!(comp)){raiseError(num); return ret;} }while(0)


#ifdef DEBUG
    #include <stdio.h>
    #undef raiseErrorIf
    #define raiseErrorIf( comp, num, ret )                              \
        do{ if (!(comp)){                                               \
            raiseError(num);                                            \
            fprintf( stderr, "\nIn file %s\n%d :: %s => %s\nAsertion failed"\
              ": %s\n", __FILE__,__LINE__,__func__,errorMessage(num),#comp);\
            return ret;                                                 \
          } }while(0)
#endif

#endif // ERROR_H

```

```c
/**
* @file game.c
* Operations for the struct @c game
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "error.h"
#include "utils.h"
#include "defines.h"
#include "playGame.h"
#include "game.h"


#define SAFE_FWRITE_INT( x )                                            \
    do{ int_ = (int)(x);                                                \
        raiseErrorIf( fwrite( &int_, sizeof(int_), 1, out ), FILEERROR,);\
    }while(0)

#define SAFE_FWRITE_CHAR( x )                                           \
    do{ char_ = (char)(x);                                              \
        raiseErrorIf( fwrite( &char_, sizeof(char_), 1, out ), FILEERROR,);\
    }while(0)

#define SAFE_FREAD_INT( x )                                             \
    do{ raiseErrorIf( fread( &(x), sizeof(int), 1, in ),               \
                    feof(in) ? CORRUPTFILE : FILEERROR, NULL ); }while(0)

#define SAFE_FREAD_CHAR( x )                                            \
    do{ raiseErrorIf( fread( &(x), sizeof(char), 1, in ),              \
                    feof(in) ? CORRUPTFILE : FILEERROR, NULL ); }while(0)


/**
* Creates a new game, based on @a options.
*
* @throws MEMORYERROR          if there was a problem while allocating memory
* @throws COMPUTATIONALERROR   if after @b MAX_WAITING_TIME time no solution
*                              for {@link randFill()} has been obtained
*
* @param options    contains options about the game
*
* @return a pointer to the new game
*
* @see newMatrix()
* @see randfill()
*/

game_t *
newGame( options_t * options )
{
    int i,j;
    game_t * sol = malloc( sizeof(game_t) );
    raiseErrorIf( sol, MEMORYERROR, NULL );

    sol->options = *options;
    // filling the game mode
    switch( sol->options.mode ){
        case SINGLEMODE:
        case TIMEMODE:
            sol->numPlayers = 1;
            break;
        case MULTIPLMODE:
            sol->numPlayers = 2;
            break;
        default:
            sol->numPlayers = 0;
            break;
    };
    // filling players
    sol->players = malloc( sol->numPlayers * sizeof(player_t) );
    if( !sol->players ){
        free( sol );
        raiseError( MEMORYERROR );
        return NULL;
    }
    // filling the players boards
    for( i = 0 ; i < sol->numPlayers ; i++ ){
        sol->players[i].board.matrix=
            newMatrix( sol->options.height, sol->options.width );
        sol->players[i].lastBoard.matrix =
            newMatrix( sol->options.height, sol->options.width );

        if( !sol->players[i].board.matrix || !sol->players[i].lastBoard.matrix ){
            for( j = 0 ; j <= i ; j++ ){
                freeMatrix( sol->players[j].lastBoard.matrix, sol->options.height );
                freeMatrix( sol->players[j].board.matrix, sol->options.height );
            }
            free( sol->players );
            free( sol );
            raiseError( MEMORYERROR );
            return NULL;
        }

        sol->players[i].board.points = 0;
        sol->players[i].board.emptySpots = sol->options.height * sol->options.width;
        sol->players[i].canUndo = false;

        randFill( sol, i, sol->options.initialTokens, true );

        if( errorCode() != NOERROR ){
            for( j = 0 ; j <= i ; j++ ){
                freeMatrix( sol->players[j].lastBoard.matrix, sol->options.height );
                freeMatrix( sol->players[j].board.matrix, sol->options.height );
            }
            free( sol->players );
            free( sol );
            return NULL;
        }
    }
    // filling the starting conditions
    sol->state.next = 0;
    sol->state.lastTime = time(NULL);
    sol->state.timeLeft = sol->options.initialSeconds;
    sol->state.quit = false;

    return sol;
}


/**
* Frees the memory reserved by {@link newGame()}.
*
* @param  game  the information of the game about to be freed
*
```

```
* @see newGame()
* @see freeMatrix()
*/

void
freeGame( game_t * game )
{
    int i;
    for( i = 0 ; i < game->numPlayers ; i++ ){
        freeMatrix( game->players[i].lastBoard.matrix, game->options.height );
        freeMatrix( game->players[i].board.matrix, game->options.height );
    }
    free( game->players );
    free( game );
}


/**
* Saves @a game in a file.
*
* @throws FILEERROR if there was an problem while opening/writing the file
*
* @param  game  contains all the data about the game
* @param  file  contains the name of file about to be saved
*
* @see readGame()
*/

void
writeGame( game_t * game, const char * file )
{
    int i,x,y;
    int int_;
    char char_;
    //opening the file
    FILE * out = fopen(file, "wb");
    raiseErrorIf(out,FILEERROR,);

    // saving all the content in the file
    SAFE_FWRITE_INT( game->options.mode );
    if( game->options.mode == TIMEMODE ){
        time_t aux = time(NULL);
        game->state.timeLeft -= aux - game->state.lastTime;
        game->state.lastTime = aux;
        SAFE_FWRITE_INT( game->state.timeLeft );
    }else
    if( game->options.mode == MULTIPLMODE )
        SAFE_FWRITE_INT( game->state.next + 1 );
    SAFE_FWRITE_INT( game->options.height );
    SAFE_FWRITE_INT( game->options.width );
    SAFE_FWRITE_INT( game->options.numColors );
    SAFE_FWRITE_INT( game->options.tokensPerLine );
    SAFE_FWRITE_INT( game->options.tokensPerTurn );
    for( i = 0 ; i < game->numPlayers ; i++ ){
        SAFE_FWRITE_INT( game->players[i].board.points );
        for( y = 0 ; y < game->options.height ; y++ )
            for( x = 0 ; x < game->options.width ; x++ )
                SAFE_FWRITE_CHAR( game->players[i].board.matrix[y][x] + '0' );
    }
    //finishing and closing the file
    i = fclose(out);
    raiseErrorIf(i==0,FILEERROR,);
}
```

```


/**
* Validates @a game, in order to prevent corrupted input files.
*
* @param  game  contains all the data about the game
*
* @return true if the game is valid, otherwise, false
*
* @see readGame()
*/

static bool
validateGame( game_t * game )
{
    int i,x,y;

    if( !entre( 0, game->options.mode, 3 ) )
        return false;
    if( game->options.mode == TIMEMODE &&
                        !entre( 0, game->state.timeLeft, 60*MAX_MINUTES + 1 ) )
        return false;
    if( game->options.mode == MULTIPLMODE && !entre( 0, game->state.next, 2 ) )
        return false;
    else
    if( !entre( 0, game->state.next, 1 ) )
        return false;
    if( !entre( MIN_TAB_DIM, game->options.width, MAX_TAB_DIM + 1) )
        return false;
    if( !entre( MIN_TAB_DIM, game->options.height, MAX_TAB_DIM + 1 ) )
        return false;
    if( !entre( MIN_COLORS, game->options.numColors, MAX_COLORS + 1 ) )
        return false;
    if( !entre( MIN_TOK_PER_LINE, game->options.tokensPerLine,
                        min( game->options.width, game->options.height ) + 1 ) )
        return false;
    if( !entre( 1, game->options.tokensPerTurn,
                            game->options.width * game->options.height + 1 ) )
        return false;
    for( i = 0 ; i < game->numPlayers ; i++ ){
        if( game->players[i].board.points < 0 )
            return false;
        for( y = 0 ; y < game->options.height ; y++ )
            for( x = 0 ; x < game->options.width ; x++ )
                if( !entre( 0, game->players[i].board.matrix[y][x],
                                                    game->options.numColors +1))
                    return false;
    }
    return true;
}


/**
* Reads a game from a file.  Use {@link newGame()} to load the game.
*
* @throws MEMORYERROR   if there was an problem while allocating memory
* @throws FILEERROR     if there was an problem while opening/reading from file
*
* @param file   contains the name of the file about to be read
*
* @return a pointer to a game_t containing all the data about the game
*
* @see writeGame()
```

```c
* @see newGame()
* @see validateGame()
*/

game_t *
readGame( const char * file )
{
    int i,x,y;
    options_t options;
    state_t state;
    game_t * sol;

    FILE * in  = fopen(file,"rb");
    raiseErrorIf(in,FILEERROR,NULL);

    SAFE_FREAD_INT( options.mode );
    state.next = 0;
    state.quit = false;
    if( options.mode == TIMEMODE ){
        SAFE_FREAD_INT( state.timeLeft );
        state.lastTime = time(NULL);
    } else
    if( options.mode == MULTIPLMODE ){
        SAFE_FREAD_INT( state.next );
        state.next--;
    }
    SAFE_FREAD_INT( options.height );
    SAFE_FREAD_INT( options.width );
    SAFE_FREAD_INT( options.numColors );
    SAFE_FREAD_INT( options.tokensPerLine );
    SAFE_FREAD_INT( options.tokensPerTurn );

    options.initialTokens = 0;
    sol = newGame( &options );
    raiseErrorIf(errorCode()==NOERROR,errorCode(),NULL);
    sol->state = state;

    for( i = 0 ; i < sol->numPlayers ; i++ ){
        SAFE_FREAD_INT( sol->players[i].board.points );
        for( y = 0 ; y < sol->options.height ; y++ )
            for( x = 0 ; x < sol->options.width ; x++ ){
                SAFE_FREAD_CHAR( sol->players[i].board.matrix[y][x] );
                sol->players[i].board.matrix[y][x] -= '0';
                if( sol->players[i].board.matrix[y][x] )
                    sol->players[i].board.emptySpots--;
            }
    }
    raiseErrorIf(!fread( &i, sizeof(int), 1, in ),CORRUPTFILE,NULL);
    i = fclose(in);
    raiseErrorIf(i==0,FILEERROR,NULL);

    raiseErrorIf(validateGame(sol),CORRUPTFILE,NULL);

    return sol;
}

```

```c
/**
* @file game.h
* Operations for the struct @c game
*/

#ifndef GAME_H
#define GAME_H

#include <stdbool.h>
#include <time.h>


typedef enum{
    SINGLEMODE, TIMEMODE, MULTIPLMODE
} modus_t; // mode_t is already defined in some compilers libraries

typedef struct{
    char ** matrix;
    int points;
    int emptySpots;
} board_t;

typedef struct{
    board_t board;
    board_t lastBoard;
    bool canUndo;
} player_t;

typedef struct{
    time_t lastTime;
    time_t timeLeft; // seconds left (mode 1)
    int next; // next player (mode 2)
    bool quit;
} state_t;

typedef struct{
    modus_t mode; // 0 normal, 1 time, 2 two-players
    size_t height;
    size_t width;
    int numColors;
    int tokensPerLine; // minimum number of consecutive pieces to make a line
    int tokensPerTurn; // new random pieces at each round
    int initialTokens; // tokens located at random when starting
    time_t initialSeconds; // seconds before game is over
} options_t;

typedef struct{
    int numPlayers;
    options_t  options;
    state_t    state;
    player_t * players;
} game_t;


game_t *
newGame( options_t * options );

void
freeGame( game_t * game );

void
writeGame( game_t * game, const char * file );

```

```c
game_t *
readGame( const char * file );


#endif // GAME_H

```

```makefile
### FILES ###

# executable name
TARGET=colorLines
# sources names
SOURCES=error utils playGame game colorsBack colors menu userInterface colorsFront

### DIRECTORIES ###

# .o files directories
OPATH=lib/
# .h files directories
HPATH=back_end/include front_end/include
# .c files directories
CPATH=back_end/source front_end/source

### COMPILATIO FLAGS ###

# sources flags
CFLAGS=-Wall -pedantic -std=c99 -lm
# target flags
TFLAGS=-Wall -pedantic -std=c99 -lm
# extra debug flags
DEBUGFLAGS=-g -O0
# extra release flags
RELEASEFLAGS=-O3

### OTHERS ###

# compiler
CC=gcc

### possible targets: debug release clean
all: release

OBJS=$(addsuffix .o,$(SOURCES) )
OUTPUT_OPTION=-o $(OPATH)$@
LDFLAGS += -L $(OPATH)
CFLAGS += $(addprefix -I ,$(HPATH))
vpath %.c $(CPATH)

.PHONY: all clean debug release

$(TARGET): $(OBJS)
    $(CC) $(TFLAGS) -o $(TARGET) $(addprefix $(OPATH),$(OBJS))

debug: override CFLAGS += $(DEBUGFLAGS)
debug: override TFLAGS += $(DEBUGFLAGS)
debug: $(TARGET)

release: override CFLAGS += $(RELEASEFLAGS)
release: override TFLAGS += $(RELEASEFLAGS)
release: $(TARGET)

clean:
    rm -f $(OPATH)*.o
    rm -f $(TARGET)
    find ./ -name "*~" -delete
```

```makefile
### FILES ###

# executable name
TARGET=colorLines
# sources names
SOURCES=error utils playGame game colorsBack colors menu userInterface colorsFront

### DIRECTORIES ###

# .o files directories
OPATH=lib/
# .h files directories
HPATH=back_end/include front_end/include
# .c files directories
CPATH=back_end/source front_end/source

### COMPILATIO FLAGS ###

# sources flags
CFLAGS=-Wall -pedantic -std=c99 -lm
# target flags
TFLAGS=-Wall -pedantic -std=c99 -lm
# extra debug flags
DEBUGFLAGS=-g -O0
# extra release flags
RELEASEFLAGS=-O3

### OTHERS ###

# compiler
CC=gcc

### possible targets: debug release clean
all: release

OBJS=$(addsuffix .o,$(SOURCES) )
OUTPUT_OPTION=-o $(OPATH)$@
LDFLAGS += -L $(OPATH)
CFLAGS += $(addprefix -I ,$(HPATH))
vpath %.c $(CPATH)

.PHONY: all clean debug release

$(TARGET): $(OBJS)
	$(CC) $(TFLAGS) -o $(TARGET) $(addprefix $(OPATH),$(OBJS))

debug: override CFLAGS += $(DEBUGFLAGS)
debug: override TFLAGS += $(DEBUGFLAGS)
debug: $(TARGET)

release: override CFLAGS += $(RELEASEFLAGS)
release: override TFLAGS += $(RELEASEFLAGS)
release: $(TARGET)

clean:
	rm -f $(OPATH)*.o
	rm -f $(TARGET)
	find ./ -name "*~" -delete

```

```c
/**
* @file menu.c
* Main menu for the game.
*/

#include <stdio.h>
#include "error.h"
#include "utils.h"
#include "defines.h"
#include "userInterface.h"
#include "menu.h"

typedef enum{
    MODE0 = SINGLEMODE,
    MODE1 = TIMEMODE,
    MODE2 = MULTIPLMODE,
    READFROMFILE,
    QUIT
} modeOption_t;


/**
* Reads an integer and asserts it's in the interval [@a a,@a b].
*
* @throws INPUTERROR    if there was an error while reading from standard input
*
* @param a  lower bound
* @param b  upper bound
*
* @return integer between a and b
*
* @see askCommand()
* @see validateInt()
*/

static int
askInt( int a, int b )
{
    int sol;
    char c;
    static char error[MAX_ERR_LEN];
    static char buffer[MAX_COM_LEN];

    error[0] = 0;
    do{
        clearScreen();
        drawText( NULL );
        drawPanel( error );
        askCommand( buffer );
        raiseErrorIf( errorCode() == NOERROR, errorCode(), -1 );
        error[0] = 0;
        if ( ( c = sscanf( buffer, " %d %c ", &sol, &c ) ) != 1 && buffer[0] != '\n' )
            sprintf( error, "Format error:\nMust be an integer" );
    }while( c != 1 || !validateInt( a, sol, b+1, error ) );

    return sol;
}


/**
* Reads two integers and asserts they are in the intervals [@a a1,@a b1] and
* [@a a2,@a b2] respectively.
*
```

```c
* @throws INPUTERROR    if there was an error while reading from standard input
*
* @param a1         lower bound for @a n1
* @param[out] n1    pointer to first element
* @param b1         upper bound for @a n1
* @param a2         lower bound for @a n2
* @param[out] n2    pointer to the second element
* @param b2         upper bound for @a n2
*
* @see askCommand()
* @see validateInt()
*/

static void
ask2Int( int a1, int * n1, int b1, int a2, int * n2, int b2 )
{
    char c;
    static char error[MAX_ERR_LEN];
    static char buffer[MAX_COM_LEN];

    error[0] = 0;
    do{
        clearScreen();
        drawText( NULL );
        drawPanel( error );
        askCommand( buffer );
        raiseErrorIf( errorCode() == NOERROR, errorCode(), );
        error[0] = 0;
        if( ( c = sscanf( buffer, " %d %d %c", n1, n2, &c ) ) != 2 && buffer[0] != '\n' )
            sprintf( error, "Format error:\nMust be two integers, "
                        "space separated" );
    }while( c != 2 || !validateInt( a1, *n1, b1+1, error ) ||
                        !validateInt( a2, *n2, b2+1, error ) );
}


/**
* Reads a string and asserts it's not empty.
*
* @throws INPUTERROR    if there was an error while reading from standard input
*
* @param[out] str   destination string
*
* @return string with the filename (@a str)
*
* @see askCommand()
*/

static char *
askString( char * str )
{
    char c;
    static char error[MAX_ERR_LEN];
    static char buffer[MAX_COM_LEN];

    do{
        clearScreen();
        drawText( NULL );
        drawPanel( error );
        askCommand( buffer );
        raiseErrorIf( errorCode() == NOERROR, errorCode(), NULL );
        error[0] = 0;
        if( ( c = sscanf( buffer, " %s", str ) ) != 1 && buffer[0] != '\n' )
```

```c
            sprintf( error, "Format error:\nMust be a string" );
    }while( c != 1 );
    return str;
}

/**
* Displays mode menu and asks game mode.
*
* @throws INPUTERROR    if there was an error while reading from standard input
*
* @return game mode
*
* @see askInt()
*/

static modeOption_t
chooseMode()
{
    drawText("Enter the game mode [1-5]:\n"
            "  1. Single player normal mode\n"
            "  2. Single player time mode\n"
            "  3. Two players\n"
            "  4. Recover game from file\n"
            "  5. Quit\n" );

    switch( askInt(1,5) ){
        case 1: return MODE0;
        case 2: return MODE1;
        case 3: return MODE2;
        case 4: return READFROMFILE;
        case 5:
        default: return QUIT;
    }
}

/**
* Displays game options menu and asks for them.
*
* @throws INPUTERROR    if there was an error while reading from standard input
*
* @param mode   game mode
*
* @return structure containing the options chosen.
*
* @see askInt()
* @see ask2Int()
*/

static options_t
chooseOptions( modus_t mode )
{
    options_t options;
    int h, w;

    options.mode = mode;
    if( options.mode == TIMEMODE ){
        drawText("Enter the time limit (in minutes):\n");
        options.initialSeconds = 60 * askInt( 1, MAX_MINUTES );
        raiseErrorIf( errorCode() == NOERROR, errorCode(), options );
    }

    drawText("Enter the dimensions of the board "
            "(rows and columns space separated):\n");
    ask2Int( MIN_TAB_DIM, &h, MAX_TAB_DIM, MIN_TAB_DIM, &w, MAX_TAB_DIM );
    raiseErrorIf( errorCode() == NOERROR, errorCode(), options );

    options.height = h;
    options.width = w;

    drawText("Enter the number of colors with which you wish to play:\n");
    options.numColors = askInt( MIN_COLORS, MAX_COLORS );
    raiseErrorIf( errorCode() == NOERROR, errorCode(), options );

    drawText("Enter the number of pieces that are initially on the board:\n");
    options.initialTokens = askInt(1, options.width * options.height );
    raiseErrorIf( errorCode() == NOERROR, errorCode(), options );

    drawText("Enter the number of pieces that make a line:\n");
    options.tokensPerLine = askInt( MIN_TOK_PER_LINE,
                                    min( options.width, options.height ) );
    raiseErrorIf( errorCode() == NOERROR, errorCode(), options );

    drawText("Enter the number of pieces that are added on each turn:\n");
    options.tokensPerTurn = askInt(1, options.width * options.height );
    raiseErrorIf( errorCode() == NOERROR, errorCode(), options );

    return options;
}


/**
* Displays the game menu. Uses {@link chooseMode()},
* for @b MODE0, @b MODE1, @b MODE2 calls {@link chooseOptions()}
* for @b READFROMFILE calls {@link readGame()}.
*
* @throws INPUTERROR           if there was an error while reading from
*                              standard input
* @throws MEMORYERROR          if there was a problem while allocating memory
* @throws FILEERROR            if there was an problem while opening/reading
*                              from file
* @throws COMPUTATIONALERROR   if after @b MAX_WAITING_TIME time no solution
*                              for {@link randFill()} has been obtained
*
* @param[out] game pointer to a @c game_t structure
*
* @return false if user wants to quit game, true otherwise
*
* @see chooseMode()
* @see chooseOptions()
* @see askString()
* @see readGame()
*/

bool
menu( game_t ** game )
{
    options_t options;
    char str[MAX_COM_LEN];
    modeOption_t modeOption;

    clearError();

    modeOption = chooseMode();

    raiseErrorIf( errorCode() == NOERROR, errorCode(), true );

    switch( modeOption ){
        case MODE0:
        case MODE1:
        case MODE2:
            options = chooseOptions( (modus_t)modeOption );
            raiseErrorIf( errorCode() == NOERROR, errorCode(), true );
            *game = newGame( &options );
            return true;
        case READFROMFILE:
            drawText("Enter the name of the file:\n");
            *game = readGame( askString( str ) );
            return true;
        case QUIT:
        default:
            return false;
    }
}

```

```c
/**
* @file menu.h
* Main menu for the game.
*/

#ifndef MENU_H
#define MENU_H

#include "game.h"


bool
menu( game_t ** game );


#endif // MENU_H

```

```c
/**
* @file noColors.c
* Dummy library, used when not in Unix nor Windows.
*/


/**
* Dummy function, used when not in Unix nor Windows.
*
* @param c  color
*/

void
textColor( color c )
{
}


/**
* Dummy function, used when not in Unix nor Windows.
*
* @param c  color
*/

void
backColor( color c )
{
}


/**
* Dummy function, used when not in Unix nor Windows.
*
* @param a  attribute
*/

void
textAttr( attr a )
{
}

```

```c
/**
* @file playGame.c
* Functions for token handling
*/

#include <stdlib.h>
#include <time.h>
#include "error.h"
#include "defines.h"
#include "utils.h"
#include "colors.h"
#include "playGame.h"

typedef struct {
    int x,y;
}direction_t;


/**
* Looks for lines of the same color, intersecting (@a x,@a y). If lines are
* found, they are erased.
*
* @param game   game structure
* @param x      coordinate
* @param y      coordinate
* @param dir    line direction to check
*
* @return new empty spots, tokens extrancted from the board
*/

static int
lookForLine( game_t * game, int nPlayer, size_t x, size_t y, direction_t dir )
{
    int i, dx, dy, tokens = 1;

    color c = game->players[nPlayer].board.matrix[y][x];
        // count how many tokens of the same color are aligned
    dx = x + dir.x; dy = y + dir.y;
    while ( entre( 0, dx, game->options.width ) &&
            entre( 0, dy, game->options.height ) &&
            game->players[nPlayer].board.matrix[dy][dx] == c ){
                dx += dir.x;
                dy += dir.y;
                tokens++;
    }
    dx = x - dir.x; dy = y - dir.y;
    while ( entre( 0, dx, game->options.width ) &&
            entre( 0, dy, game->options.height ) &&
            game->players[nPlayer].board.matrix[dy][dx] == c ){
                dx -= dir.x;
                dy -= dir.y;
                tokens++;
    }
        // erase the line
    if( tokens >= game->options.tokensPerLine ){
        for( i = 0 ; i < tokens ; i++ ){
            dx += dir.x;
            dy += dir.y;
            game->players[nPlayer].board.matrix[dy][dx] = 0;
        }
        game->players[nPlayer].board.matrix[y][x] = c;
        return tokens;
    }
    return 0;
}


/**
* Checks for lines, erases them, actualizes emptySports and, if @a countPoints
* is true, then it also actualizes points.
*
* @param game          game structure
* @param x             coordinate
* @param y             coordinate
* @param countPoints   if false points won't be taken into account
*
* @return number of lines deleted
*
* @see lookForLine()
*/

int
winningPlay( game_t *game, int nPlayer, size_t x, size_t y, bool countPoints )
{
    int i, aux, emptySpots=0, lines = 0;
    direction_t directions[]={ {0,1}, {1,0}, {1,1}, {-1,1} };

    for( i = 0 ; i < 4 ; i++){
        aux = lookForLine( game, nPlayer, x, y, directions[i] );
        if(aux){
            lines++;
            emptySpots += aux - 1;
        }
    }
    if(emptySpots){
        emptySpots++;
        game->players[nPlayer].board.matrix[y][x] = 0;
        game->players[nPlayer].board.emptySpots += emptySpots;
        if( countPoints ){
            if( lines > 1 )
                game->players[nPlayer].board.points += 8;
            else
                switch( emptySpots - game->options.tokensPerLine ){
                    case 0:
                        game->players[nPlayer].board.points += 1;
                        break;
                    case 1:
                        game->players[nPlayer].board.points += 2;
                        break;
                    case 2:
                        game->players[nPlayer].board.points += 4;
                        break;
                    case 3:
                        game->players[nPlayer].board.points += 6;
                        break;
                    default:
                        game->players[nPlayer].board.points += 8;
                        break;
                }
        }
    }
    return emptySpots;
}


/**
```

```c
* Fills the board with a certain number of random tokens.
*
* @throws COMPUTATIONALERROR    if after @b MAX_WAITING_TIME time no solution
*                               has been obtained (just on force mode)
*
* @param game       contains all the information about the current game
* @param nPlayer    player number
* @param cant       indicates the number of tokens about to be placed
* @param force      indicates if when a line is made, and tokens are erased,
*                   tokens should still be filled until @a cant are reached
*
* @see winningPlay()
*/

void
randFill( game_t * game, int nPlayer, size_t cant, bool force )
{
    int i, j, pos;
    time_t initTime = time(NULL);

    struct point{
        int x,y;
    } vec[ game->players[nPlayer].board.emptySpots ], aux;

    do{
        // fill vec with the coordinates of all the empty spots
        pos = 0;
        for( i = 0 ; i < game->options.height ; i++ )
            for( j = 0 ; j < game->options.width ; j++ )
                if( !game->players[nPlayer].board.matrix[i][j] )
                    vec[pos++] = (struct point){j,i};
        // random shuffle vec
        for( i = 0 ; i < game->players[nPlayer].board.emptySpots ; i++ ){
            pos = rand() % game->players[nPlayer].board.emptySpots;
            aux = vec[i];
            vec[i] = vec[pos];
            vec[pos] = aux;
        }
        // fill with cant tokens
        pos = game->players[nPlayer].board.emptySpots;
        j = 0;
        for( i = 0 ; i < cant ; i++ ){
            if( game->players[nPlayer].board.emptySpots <= 0 || pos <= i ){
                break;
            }
            game->players[nPlayer].board.emptySpots--;
            game->players[nPlayer].board.matrix[ vec[i].y ][ vec[i].x ] =
                                                    rand() % game->options.numColors + 1;

            j += 1 - winningPlay( game, nPlayer, vec[i].x, vec[i].y, false );
        }
        cant -= j;

        raiseErrorIf( time(NULL)-initTime < MAX_WAITING_TIME, COMPUTATIONALERROR,);
    }while( force && cant > 0 && game->players[nPlayer].board.emptySpots > 0 );
}


/**
* Checks if game is over for player @a nPlayer
*
* @param nPlayer player number
*
```

```c
* @returns true if game is over for player @a nPlayer. false otherwise
*/

bool
gameOver( game_t * game, int nPlayer )
{
    return ( game->options.mode == TIMEMODE
            && game->state.timeLeft - time(NULL) + game->state.lastTime <= 0 )
            || game->players[nPlayer].board.emptySpots <= 0;
}

```

```c
/**
* @file playGame.h
* Functions for token handling
*/

#ifndef PLAYGAME_H
#define PLAYGAME_H

#include <stdbool.h>
#include "colors.h"
#include "game.h"

int
winningPlay( game_t *game, int nPlayer, size_t x, size_t y, bool countPoints );

void
randFill( game_t * game, int nPlayer, size_t cant, bool force );

bool
gameOver( game_t * game, int nPlayer );


#endif // PLAYGAME_H

```

```c
/**
* @file unixColors.c
* Console colors in Unix.
*/

#include <stdio.h>

static const char color2font_color[] =
{ 30,31,32,33,34,35,36,37,90,91,92,93,94,95,96,97 };
static const char color2font_bkcolor[] =
{ 40,41,42,43,44,45,46,47,100,101,102,103,104,105,106,107 };
static const char attr2font_attr[] =
{ 0,10,1,4,5,7,22,24,25,27 };


/**
* Sets the font color.
*
* @param c  color
*/

void
textColor( color c )
{
    if( !USE_COLORS ) return;
    printf( "\033[%dm", color2font_color[(int)c] );
}


/**
* Sets the background color.
*
* @param c  color
*/

void
backColor( color c )
{
    if( !USE_COLORS ) return;
    printf( "\033[%dm", color2font_bkcolor[(int)c] );
}


/**
* Sets text attributes.
*
* @param a attribute
*/

void
textAttr( attr a )
{
    if( !USE_COLORS ) return;
    printf( "\033[%dm", attr2font_attr[(int)a] );
}

```

```c
/**
* @file userInterface.c
* Funtions for human-computer interaction
*/

#include <stdio.h>
#include <string.h>
#include <time.h>
#include "error.h"
#include "utils.h"
#include "defines.h"
#include "playGame.h"
#include "userInterface.h"


static char commandsBuffer[ MAX_PANEL_LINES ][ MAX_COM_LEN + 1 ];

static int commandsBufferPos = -1;


/**
* Clears the screen printing '\n'
*
*/

void
clearScreen()
{
    int i;
    backColor(BLACK);
    for( i = 0 ; i < MAX_TAB_DIM ; i++ )
        printf("\n");
    textAttr(CLEAR);
}

/**
* Draws the boards of each player in the game, the score and if necessary
* the time left, provided that it's not game over, in which case it prints
* just the score and "GAME OVER"
*
* @param game   contains all information about current game
*/

void
drawTable( game_t * game )
{
    int i, j, player, col;
    static const int colors[] = {
        BLACK, RED, LIGHT_BLUE, GREEN, YELLOW,
        VIOLET, PINK, SKY_BLUE, BROWN, LIGHT_GREEN
    };
    backColor(BLACK);
    textColor(WHITE);

    printf("\n   ");
    for( player = 0 ; player < game->numPlayers ; player++ ){
        for( i = 0 ; i < game->options.width ; i++ )
            printf("  %-2d",i);
        printf("       ");
    }

    if( HOR_LINES ){
        printf("\n   ");
        for( player = 0 ; player < game->numPlayers ; player++ ){
            for( i = 0 ; i < game->options.width ; i++ )
                printf("+---");
            printf("+      ");
        }
    }
    printf("\n");

    for( i = 0 ; i < game->options.height ; i++ ){

        for( player = 0 ; player < game->numPlayers ; player++ ){

            printf("%2d",i);
            for( j = 0 ; j < game->options.width; j++ ){

                col = game->players[
                        ( game->state.next + player ) % game->numPlayers
                ].board.matrix[i][j];

                printf(" | ");
                textColor( colors[ (int)col ] );
                if(col)
                    printf("%d", (int)col );
                else
                    printf(" ");
                textColor(WHITE);
            }
            printf(" |   ");
        }
        if( HOR_LINES ){
            printf("\n   ");
            for( player = 0 ; player < game->numPlayers ; player++ ){
                for(j=0; j < game->options.width; j++)
                    printf("+---");
                printf("+      ");
            }
        }
        printf("\n");
    }
    // draw points, time/player/GAME OVER
    char s[15], t[15];
    sprintf( s, "%%%ds", game->options.width * 4 - 10 );

    for( player = 0 ; player < game->numPlayers ; player++ ){

        i = ( game->state.next + player ) % game->numPlayers;

        printf("    SCORE %-4d", game->players[i].board.points );

        if( gameOver( game, player ) ){
            printf( s, "GAME OVER" );
        }else
        if( game->options.mode == TIMEMODE ){
            j = game->state.timeLeft - time(NULL) + game->state.lastTime;
            if( j >= 60 )
                sprintf( t, "%d:%d", j/60, j%60 );
            else
                sprintf( t, "%d", j );
            printf( s, t );
        }else
        if( game->options.mode == MULTIPLMODE ){
            sprintf( t, "Player %d", i + 1 );
            printf( s, t );
        }
        printf("    ");
    }

    textAttr(CLEAR);
}


/**
* Draws the text main panel. If @a str is provided, then the text is set to be
* it, else if @a str is NULL, the last text provided is used.
*
* @param str    text to print (can be NULL)
*/

void
drawText( const char * message )
{
    static char buffer[ MAX_TEXT ] = "";
    if( message && message[0] )
        strcpy( buffer, message );
    backColor(BLACK);
    textColor(WHITE);
    printf("%s",buffer);
    textAttr(CLEAR);
}


/**
* Draws the text panel (secondary). If @a message is provided, then it is
* printed. All the previous commands and ' > ' are printed in either case.
*
* @param str    text to print (can be NULL or empty)
*/

void
drawPanel( const char * message )
{
    char msg[ MAX_ERR_LEN ];
    int i;

    if( commandsBufferPos == -1 ){
        for( i = 0 ; i < MAX_PANEL_LINES ; i++ )
            sprintf( commandsBuffer[i], "\n" );
        commandsBufferPos = 0;
    }

    backColor(BLACK);

    if( message && message[0] ){
        sprintf( msg, "%s", message );
        char * aux = strtok( msg, "\n" );
        while( aux ){
            sprintf( commandsBuffer[ commandsBufferPos++ ], "%s\n", aux );
            commandsBufferPos %= MAX_PANEL_LINES;
            aux = strtok( NULL, "\n" );
        }
    }

    printf("\n\n");
    for( i = PANEL_LINES ; i > 0 ; i-- ){
        if( commandsBuffer[ (commandsBufferPos-i+MAX_PANEL_LINES)
                                                    % MAX_PANEL_LINES ][1] == '>' )
            textColor(WHITE);
        else
            textColor(GRAY);
        printf( "%s", commandsBuffer[ (commandsBufferPos-i+MAX_PANEL_LINES)
                                                              % MAX_PANEL_LINES ] );
    }

    textColor(WHITE);
    printf( " > " );
    textAttr(CLEAR);
}


/**
* If in @b MULTIPLMODE, draws in the panel the winner of the game.
*
* @param game   contains all information about current game
*
* @see drawPanel()
*/

void
drawWinner( game_t * game )
{
    char str[15];
    int i=0,j;
    if( game->options.mode != MULTIPLMODE )
        return;

    for( j = 1 ; j < game->numPlayers ; j++ )
        if( game->players[j].board.points >= game->players[i].board.points )
            i = j;
    if( i && game->players[0].board.points == game->players[i].board.points )
        sprintf( str, "Tie game." );
    else
        sprintf( str, "Player %d wins!!!", i+1 );

    drawPanel( str );
}

/**
* Reads a command from the standard input.
*
* @throws INPUTERROR    if there was an error while reading from standard input
*
* @param[out] str   buffer ( has at least MAX_COM_LEN size allocated )
*
* @return string with the command (@a str)
*/

char *
askCommand( char * result )
{
    backColor(BLACK);
    textColor(WHITE);

    result[ MAX_COM_LEN-5 ] = 0;

    raiseErrorIf( fgets( result, MAX_COM_LEN-3, stdin ), INPUTERROR, NULL);

    if( result[ MAX_COM_LEN-5 ] ){      // we make sure it's \n terminated
        result[ MAX_COM_LEN-5 ] = '\n';
        clearBuffer();
    }
    sprintf( commandsBuffer[ commandsBufferPos++ ], " > %s", result );
    commandsBufferPos %= MAX_PANEL_LINES;

    textAttr(CLEAR);
    printf("\n");

    return result;
}

```

```c
/**
* @file userInterface.h
* Funtions for human-computer interaction
*/

#ifndef UI_H
#define UI_H

#include "game.h"
#include "colors.h"

void
clearScreen();

void
drawTable( game_t * game  );

void
drawText( const char * message );

void
drawPanel( const char * message );

void
drawWinner( game_t * game );

char *
askCommand( char * result );


#endif // UI_H

```

```c
/**
* @file utils.c
* Contains useful functions that are used several times across the code.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "error.h"
#include "defines.h"
#include "utils.h"


/**
* Creates a new matrix and fills it with 0.
*
* @throws MEMORYERROR if there was a problem while allocating memory
*
* @param height height of the new matrix
* @param width  width of the new matrix
*
* @return a matrix of characters. every element in the matrix is a 0
*
* @see freeMatrix()
*/

char **
newMatrix( size_t height, size_t width )
{
    int i;
    char ** sol = malloc( height * sizeof(char*) );
    raiseErrorIf( sol, MEMORYERROR, NULL );

    for( i = 0 ; i < height ; i++ ){
        sol[i] = calloc( width, sizeof(char) );
        // free allocated memory in case of error
        if( !sol ){
            freeMatrix( sol, i-1 );
            raiseError( MEMORYERROR );
            return NULL;
        }
    }
    return sol;
}


/**
* Frees the reserved memory for a matrix created by {@link newMatrix()}.
*
* @param mat    the matrix of characters about to be freed
* @param height the height of the matrix
*
* @see newMatrix()
*/

void
freeMatrix( char ** mat, size_t height )
{
    int i;
    for( i = 0 ; i < height ; i++ )
        free( mat[i] );
    free(mat);
```

```c
}


/**
* Copies a matrix to another one.
*
* @param[out] to    output matrix in wich from was copied
* @param from       the matrix to be copied
* @param height     the height of the matrix
* @param width      the width of the matrix
*
* @see newMatrix()
* @see freeMatrix()
*/

void
copyMatrix( char ** to, char ** from, size_t height, size_t width )
{
    int i;
    for( i = 0 ; i < height ; i++ )
        memcpy( to[i], from[i], width );
}


/**
* Checks if @a b is in the interval [@a a,@a c).
*
* @param a  lower limit
* @param b  number to be compared
* @param c  the upper limit
*
* @return   true if @a b is in [@a a,@a c), otherwise, false
*/

bool
entre( int a, int b, int c )
{
    return a<=b && b<c;
}


/**
* Checks if @a b is in the interval [@a a,@a c). If it's not, returns the
* corresponding message.
*
* @param a          lower limit
* @param b          number to be compared
* @param c          the upper limit
* @param[out] err   buffer to print the @b RANGEERROR message
*
* @return   true if @a b is in [@a a,@a c), otherwise, false
*
* @see entre()
*/

bool
validateInt( int a, int b, int c, char * err )
{
    if( !entre(a,b,c) ){
        sprintf( err, "Range error:\nIt must belong to the "
        "interval [%d,%d]", a, c-1 );
        return false;
    }
    return true;
}


/**
* Clears the standard input buffer.
*
*/

void
clearBuffer()
{
    while(getchar() != '\n');
}


/**
* Calculates the minimum between two numbers.
*
* @param a  first number
* @param b  second number
*
* @return the minimum beetween @a a and @a b
*/

int
min( int a, int b )
{
    return (a<=b)?a:b;
}


/**
* Calculates the maximum between two numbers.
*
* @param a  first number
* @param b  second number
*
* @return the maximum beetween @a a and @a b
*/

int
max( int a, int b )
{
    return (a>=b)?a:b;
}


/**
* Calculates the edit distance between @a str1 and @a str2
* (a.k.a Damerau–Levenshtein distance).
*
* @param str1   first string
* @param str2   second string
*
* @return a number between 0 and 1 that indicates similarity between
*        @a str1 and @a str2. greater is better
*
* @see F.J. Damerau. A technique for computer detection and correction of
*                   spelling errors. Communications of the ACM, 1964.
*
* @see V.I. Levenshtein. Binary codes capable of correcting deletions,
*                       insertions, and reversals. Soviet Physics Doklady, 1966.
```

```
*/

double
editDistance( const char * str1, const char * str2 )
{
    int i,j,cost=0;
    int s1len=strlen(str1);
    int s2len=strlen(str2);
    int mat[3][ s2len+1 ];
    int prev2, prev=0, this = 0;

    if( min( s1len, s2len ) < MIN_EDIT_LEN || max( s1len, s2len ) > MAX_EDIT_LEN )
        return 0;

    for( i = 0 ; i <= s2len ; i++ )
        mat[0][i] = i;

    for( i = 0 ; i <= s1len ; i++ ){

        prev2 = prev;
        prev = this;
        this = (i+1) % 3;

        mat[this][0] = i+1;

        for( j = 0 ; j <= s2len ; j++ ){
            cost = toupper(str1[i]) != toupper(str2[j]);
            // in this order: deletions, insertions, substitutions
            mat[this][j+1] =
            min( mat[prev][j+1]+1, min( mat[this][j]+1, mat[prev][j]+cost ) );
            // transposes
            if( i && j && toupper(str1[i]) == toupper(str2[j-1])
                && toupper(str1[i-1]) == toupper(str2[j]) )

                mat[this][j+1] = min( mat[this][j+1], mat[prev2][j-1] + cost );
        }
    }

    return 1 - mat[prev][s2len] / (double)max( s1len, s2len );
}

```

```c
/**
* @file utils.h
* Contains useful functions that are used several times across the code.
*/

#ifndef UTILS_H
#define UTILS_H

#include <stdlib.h>
#include <stdbool.h>


char **
newMatrix( size_t height, size_t width );

void
freeMatrix( char ** mat, size_t height );

void
copyMatrix( char ** to, char ** from, size_t height, size_t width );

bool
entre( int a, int b, int c );

bool
validateInt( int a, int b, int c, char * err );

void
clearBuffer();

int
min( int a, int b );

int
max( int a, int b );

double
editDistance( const char * str1, const char * str2 );


#endif // UTILS_H

```

```c
/**
* @file winColors.c
* Console colors in Windows.
*/

#include <windows.h>
#include <wincon.h>
#include <stdio.h>

static const char crazyWinColorsMap[] = {
    /*NEGRO*/BLACK,/*ROJO*/BLUE,/*VERDE*/GREEN,/*MARRON*/SKY_BLUE,
    /*AZUL*/RED,/*VIOLETA*/VIOLET,/*CELESTE*/BROWN,/*GRIS_CLARO*/LIGHT_BLUE,
    /*GRIS*/GRAY,/*ROSA*/LIGHT_BLUE,/*VERDE_CLARO*/LIGHT_GREEN,/*AMARILLO*/LIGHT_BLUE_SKY,
    /*AZUL_CLARO*/PINK,/*VIOLETA_CLARO*/LIGHT_VIOLET,/*CELESTE_CLARO*/YELLOW,
    /*BLANCO*/WHITE
};

static const char color2font_color[] =
{ 0, FOREGROUND_RED | FOREGROUND_INTENSITY, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_BLUE,
  FOREGROUND_BLUE | FOREGROUND_RED, FOREGROUND_GREEN | FOREGROUND_BLUE,
  FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE, FOREGROUND_RED | FOREGROUND_GREEN,
  FOREGROUND_RED | FOREGROUND_INTENSITY, FOREGROUND_GREEN | FOREGROUND_INTENSITY,
  FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_INTENSITY, FOREGROUND_BLUE | FOREGROUND_INTENSITY,
  FOREGROUND_BLUE | FOREGROUND_RED | FOREGROUND_INTENSITY,
  FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY,
  FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY
};

static const char color2font_bkcolor[] =
{ 0, BACKGROUND_RED | BACKGROUND_INTENSITY, BACKGROUND_GREEN, BACKGROUND_RED, BACKGROUND_BLUE,
  BACKGROUND_BLUE | BACKGROUND_RED, BACKGROUND_GREEN | BACKGROUND_BLUE,
  BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_BLUE, BACKGROUND_RED | BACKGROUND_GREEN,
  BACKGROUND_RED | BACKGROUND_INTENSITY, BACKGROUND_GREEN | BACKGROUND_INTENSITY,
  BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_INTENSITY, BACKGROUND_BLUE | BACKGROUND_INTENSITY,
  BACKGROUND_BLUE | BACKGROUND_RED | BACKGROUND_INTENSITY,
  BACKGROUND_GREEN | BACKGROUND_BLUE | BACKGROUND_INTENSITY,
  BACKGROUND_RED | BACKGROUND_GREEN | BACKGROUND_BLUE | BACKGROUND_INTENSITY
};


/**
* Sets the font color.
*
* @param c  color
*/

void
textColor( color c )
{
    WORD wColor;
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_SCREEN_BUFFER_INFO csbi;

    if( !USE_COLORS ) return;
    if(GetConsoleScreenBufferInfo(hStdOut, &csbi)){
        wColor = (csbi.wAttributes & 0xF0) + ( crazyWinColorsMap[c] & 0x0F);
        SetConsoleTextAttribute(hStdOut, wColor);
    }
}


/**
* Sets the background color.
```

```c
*
* @param c  color
*/

void
backColor( color c )
{
    WORD wColor;
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_SCREEN_BUFFER_INFO csbi;

    if( !USE_COLORS ) return;
    if(GetConsoleScreenBufferInfo(hStdOut, &csbi)){
        wColor = (csbi.wAttributes & 0x0F) + ( crazyWinColorsMap[c] & 0xF0);
        SetConsoleTextAttribute(hStdOut, wColor);
    }
}


/**
* Sets text attributes.
*
* @param a attribute
*/

void
textAttr( attr a )
{
    if( !USE_COLORS ) return;
    if( a == CLEAR ){
        backcolor(BLACK);
        frontcolor(WHITE);
    }
}

```