Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics

Basics of Programming 1

CTetris Developer Documentation

Version 1.0.0

Author
Diego Davidovich Gomes

November 24, 2024

Contents

1 Introduction

This document provides an overview of the CTetris game, its implementation details, and the underlying logic to help developers understand the code. The document explains the solutions, modules, used data structures, algorithms and functions implemented in CTetris. If you would like to know more about the practical part of the program, consult the **User Guide**.

2 Solutions

This section presents the high-level description of the program, focusing in the overall design and implementation approach.

2.1 General Design

The first challenge that appeared during the development of the game is how to implement it without the complexity of graphic libraries such as SDL 2.0. To address this issue, the game is designed on the usage of the external library **ncurses**, which provides a simpler, terminal-based interface for rendering and user interaction. It allows to bring features such as colors, keyboard and keypad input, multiple windows, subwindows and efficient text rendering.

With this features in hand, the game was structured in such way that each major functionality is a encapsulated into a subprogram and runs separately. Functions are limited to one functionality or are a build up for a bigger function (e.g. to move a piece, collision checking is executed first and if it is allowed, then the piece is drawn in the new position). Some functions manipulate the screen while others are responsible for manipulating the data and the state of the game.

The documentation of the **ncurses** library can be read in [1] or accessed online in [2], specially for the many library specific functions.

3 Modules

This section provides a detailed overview of the individual modules within the program. Each module is described in terms of its structure, functionality, and implementation details and what is the role.

3.1 Overview

The program is split into four distinct modules: menu, game, leaderboard and info. They works as ”subprograms” with their own specific set of functions. To achieve this, each module has their own header and source file. The main function is only responsible for initializing the ncurses screen and set ncurses settings (remove cursor, disable writing text to the screen and line buffering), receive the selected option of the menu and call the respective function.

3.2 Project Structure and Compilation

The file structure starts with the main directory, ctetris. Inside this directory, there are four folders and two files: one is the compiled game, and the other is the makefile, responsible for simplifying the compiling process and links files from different folders together.

If we look into the four folders, each serves a distinct purpose and organizes the program's components systematically:

- **docs**: stores the user and developer documentation
- **files**: saves the scores of the previous games and the sorted file generated by the program
- **include**: contains the **header** files, which defines the used libraries, the definition of constants, global variables, type definitions, data structures and the functions prototypes. There is one header file for each .c file, except for main.c
- **src**: short form for **source**, it contains the actual function definitions and calls, as well as the **main.c**

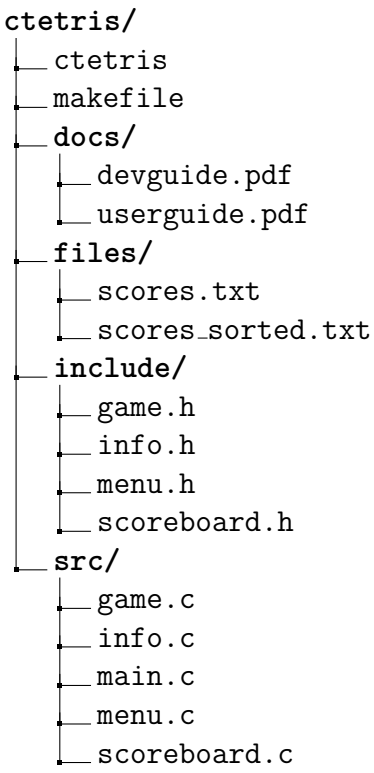

Figure 1: File Structure

The program is easily compiled using the command `make` in the terminal line, given that the current directory is set to **ctetris**. The `makefile` simplifies the process of compiling the program by automatically handling dependencies and linking the necessary files, ensuring that all source files in the project are correctly compiled and linked together. The program still can be compiled manually using:

```
gcc -o ctetris src/main.c src/game.c src/menu.c src/scoreboard.c src/info.c
-std=c99 -I include -Wall -lncurses
```

To run the program, type `./ctetris` in the terminal line.

3.3 Menu

The menu module can be called as the access point for the other modules, as it is the responsible for allowing the user to choose an option and then return it to the main and then call one of the other three modules. The menu implementation is rather simple, as it creates a subwindow in the **stdscr** (standard terminal screen) and displays the available options, highlighting the selected. The input function is placed in a loop which ends with the user presses **Enter**.

3.4 Game

This is the **core module** of the program. It is by far the most complex and the one that contains most functions. The game contains a total of 6 subwindows that display different elements in the screen: the controls, the statistics of the game, the next piece and most important: the playfield. The playfield itself is a subwindow of a slightly larger subwindow which is responsible for display the border. With the use of the ncurses functions, it is possible to easily manipulate the each subwindow separately. One interesting fact is that to achieve the square format in the terminal, the visual sizes of the playfield, pieces are doubles are actually **double spaces**. All the game related functions such as drawing a piece, creating a matrix, erasing lines, keyboard input, checking for gameover, gravity and other core functionalities are implemented here. At the end, a file which stores the points, number of cleared lines, the date and time of the game is created or new data is appended, which can be displayed and sorted at the **leaderboard** module.

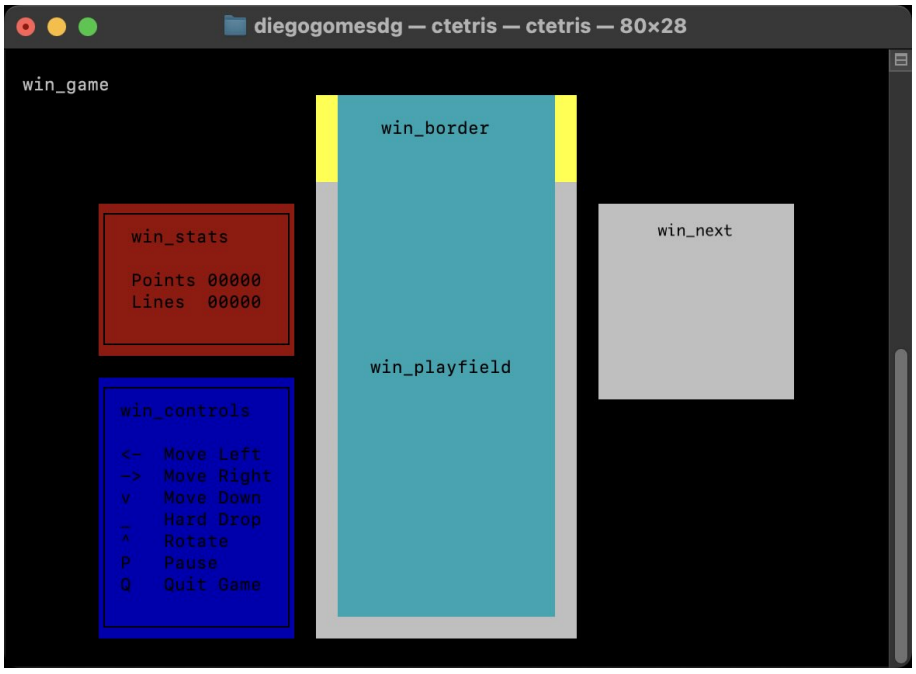

Figure 2: Subwindows in the Game

3.5 Leaderboard

The Leaderboard module serves as a window where users can view previous game scores in an organized and easily accessible manner. The data is stored in an external file that maintains a chronological order based on date and time of the games played. The

Figure 3: Leaderboard Table

data is sorted into four categories: the best scores, the worse scores, the newest games and the oldest games. Due to screen constrains and for simplicity, it is only possible to display at maximum 10 entries for any chosen category.

When the module is loaded, a new sorted file is generated. The module reads the scores produced by the game and then using quick sorting, it sorts it according to the number of points and attributes a rank for it. To display the data into the terminal, the data is read into a linked list according to the chosen category. And to simplify the reading, the data reads either the first 10 entries or the last 10 entries of one of the two files. The linked list then is feed into the functions responsible for printing the title, headers and rows.

3.6 Info

This module is a more of an informative module than functional. It consist in of a main display function which calls the `pages()` functions to display the title of the text and the text itself. The title is formatted and displayed using the `centered_title()` and the text uses the `text_wrapper()` function to display text without cutting the words. The module displays information about the development of the program, a brief user guide and recalls the game commands.

4 Data Structures

4.1 Structures

1. `struct position`
   Defined in **game.h**. The structure stores the values of the y (rows) and x (columns) position on the window or relatively to a point (e.g. matrix). It is essentially used to create and manipulate tetraminoes.

```
typedef struct position {
    unsigned int y;
    unsigned int x;
} POSITION;
```

2. `struct stats`
   Defined in **game.h**. The structure is defined and used to simply store the number of points and cleared lines in the game.

```
typedef struct position {
    unsigned int y;
    unsigned int x;
} POSITION;
```

3. `struct game_data`
   Defined in **leaderboard.h**. Is a structure made to read and write the entries of the files which save the past games. It contains the rank, the number of points and lines cleared, the date of the game and the time of the game.

```
typedef struct game_data {
    int rank;
    int points;
    int lines;
    struct tm t;
} GAME_DATA;
```

4. `struct node`
   Defined in **leaderboard.h**. This structure defines the nodes of a linked list, with the pointer to the next node and the `struct game_data`. It is used to read the entries of the files that stores the scores.

```
typedef struct node {
    struct game_data data;
    struct node *next;
} NODE;
```

4.2 Tetraminoes

The **Tetraminoes** are the pieces in the game. There are a total of 7 pieces, resulting of a combination of 4 pixels. One interesting fact is that **Tetris** comes from the word **Tetra**, meaning "four". To implement this into the game, it is used a **4×4** 2D array, where the top left element is the 0,0 position.

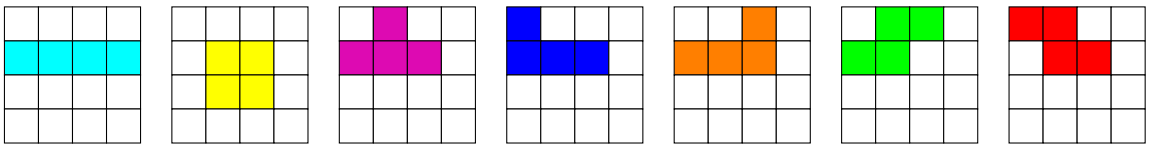

Figure 4: Tetraminoes in a 4 × 4 matrix (I, O, T, J, L, S, Z)

The **Tetraminoes** are defined in **game.c** as a three-dimensional matrix of POSITION structures, with 7 pieces, 4 possible rotations and 4 structure positions (one for each pixel). Thus, for each piece, there is four possible coordinates combinations which correspond to the rotation and for each rotation, there are four pair coordinates. For example, this is the definition of the first piece (I) and the respective rotations:

```
POSITION tetraminoes[7][4][4] = {
    {
    {{1,0},{1,1},{1,2},{1,3}}, // 1st rotation
    {{0,2},{1,2},{2,2},{3,2}}, // 2nd rotation
    {{2,0},{2,1},{2,2},{2,3}}, // 3rd rotation
    {{0,1},{1,1},{2,1},{3,1}}  // 4th rotation
}, // ...
```

4.3 Game Matrix

The game matrix a **two-dimensional integer array** that is used for tracking the state of the playfield. The matrix is dynamically allocated using malloc, with the size being defined by the height and width of the playfield, defined as PLAYFIELD_Y and PLAYFIELD_X in game.h. The standard size of the matrix is 24×10, although the visible area of the playfield is only the 20 bottom rows. The 4 additional rows constitute the **buffer area**, where the pieces are spawned. In this implementation, the buffer area serves for checking if the game is over and avoid overflow.

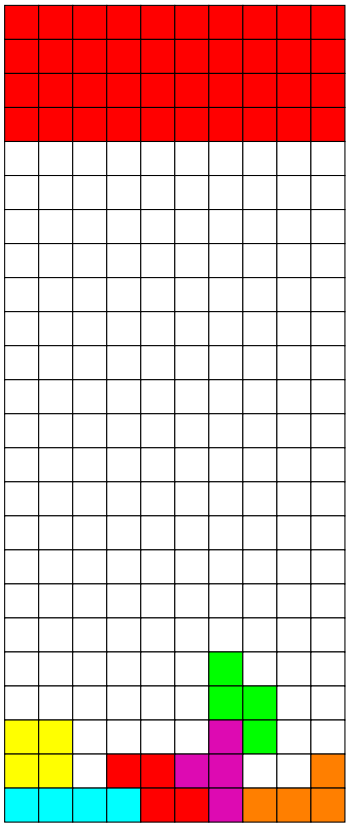

Figure 5: Game Matrix

Remark: The pieces are not stored into the game matrix until the piece lands in another piece or reaches the bottom of the playfield. This is due to the fact that if that is made, it would be necessary to have to matrices, one to store the falling piece and another to store the state, because the **collision detection** calculates if the next position is allowed. If there is only one matrix, the piece would "collide with itself) in the next state.

4.4 Linked List

The **Linked List** is used in **leaderboard.c** to dynamically read the entries of the scores and store in the linked list. It provides a simple solution for reading the first 10 entries or the last 10 entries of a file, and also avoids errors if the number of entries is less than 10. Each `struct node` contains the `struct game_data` and the pointer to the next `struct node`.

5 Algorithms

This section is dedicated to describe the main algorithms used in the modules, specially the game module. The goal is to demonstrate the logic behind the algorithms and explain the reasons behind the implementation.

5.1 Piece Drawing

To draw a piece into the screen, the function `game_draw_piece()` receives the position in the screen, the piece type, the orientation and if it should erase it or not. The position essentially tracks the position of the tetramino until it gets locked and another piece is drawn. Knowing the type and the orientation of the piece, it is possible to retrieve the relative values to Y and X of the tetramino definition, as described in 4.2. With the piece ID and the rotation, it is possible to loop through each position of the piece and add it up to the Y and X position, then move the cursor to the position and print double space with color according to the ID.

Example: In the position $Y = 5$ and $X = 5$, the position of each square in the piece T (ID: 2) with orientation 0 (`tetraminoes[2][0][y]`) would be calculated in the following way:

1. Position 1: $y = 5 + 0$ and $x = 5 + 1$
2. Position 2: $y = 5 + 1$ and $x = 5 + 0$
3. Position 3: $y = 5 + 1$ and $x = 5 + 1$
4. Position 4: $y = 5 + 1$ and $x = 5 + 2$

Remark: The example above is simplified, as actually every coordinate x is multiplied by 2.

5.2 Collision Detection

The idea for checking collision involves a similar approach from the previous algorithm in 5.1. Starting with the same approach of looping through each relative position and adding up to the Y and X location, each position is calculated and yields a Y and X value at the position i. Instead of printing to the screen, each coordinate is checked if it is

surpassing the boundaries of the playfield or if the position overlaps a registered piece in the game matrix (any value different than 0). If any position is found invalid, it returns FALSE.

5.3 Piece Movement, Dropping and Locking

The piece movement is the most complex function in the **game.c**, or at least with the most lines). Although the general approach is to make each function execute one functionality, in the case of this algorithm, it is easier to merge all piece manipulation functions together.

A piece P is initially placed in the **buffer zone** of the screen ($Y = 0$ and $X = \frac{PLAYFIELD\_X}{2} - 1$). While the piece is not locked, the game can receive controls from the user to shift left, right, down or hard drop (push all the way down), otherwise the piece will go down in a defined amount of time. For each input, the game first checks if the new position is valid or not using the `game_check_position()`. If the position is valid, the piece is first erased from the screen (`game_draw_piece() has a del parameter, either 0 or 1` and then the values of Y and X are incremented, where the piece is redraw again with the next Y and X.

Personal Note: Due to the way that gravity is handled, there is a rather unexpected bug which results from the user pressing the key down without releasing, resulting into the piece infinitely rotating or frozen in the wall. I tried to fix the bug with different methods, but it usually makes the game laggy.

Example:

```
/* Shift to the Left */
case KEY_LEFT:
    if (game_check_position(game_matrix, y , x - 1, type,
   orientation)) {
        game_draw_piece(y, x, type, orientation, 1);
        game_draw_piece(y, --x, type, orientation, 0);
    } break;
```

A similar approach is used for the right movement and down movement. For the rotation, the new orientation is also tested first, but there a implementation of **wall kicks**, so if the user rotates the piece in the border, the piece is shifted either to the left or right. In this case, if the new orientation is not allowed, the algorithm will try to shift the piece X coordinate by one or two, either to the left or to the right. If the new coordinate is suitable, the X is incremented by that amount and the piece is drawn in the new position.

To lock the piece, it is checked simply if the position below is valid or not. If it not valid, it means that there is another piece there or the piece reached the bottom of the playfield, therefore the piece is locked into the position and written into the **game matrix**.

5.4 Line Clearing

To clear the lines, the function `game_check_playfield()` scans each row of the visible playfield area with the function `game_check_row()`, which checks if in a row Y, all the

X are different than 0 in the game matrix. If the condition is satisfied, it returns TRUE. If is TRUE, there is a call to the function `game_remove_row()`, which sets the row to 0 in the game matrix and then shifts down every row above by 1, from bottom to top. Then, according to the new state of the game matrix, the playfield is redrawn using `game_playfield_update()`.

5.5 Game Over

This is a relatively simple function, as it loops through the first four rows and for each column (double for loop) and checks if any locked piece (values different than 0 in the game matrix) is present. If any non-zero values is found, it signals TRUE, i.e. the game is over, otherwise is FALSE.

6 Functions

This section provides a list of all the functions defined in the program. Each function is described with details about its purpose, input parameters, return type, and the results it produces.

6.1 Overview

Each module has its own **"main function"**, which serves as the entry point for the module and it is listed as the first function of each module, working as a subprogram to be called by the **main.c**.

The functions are designed to be modular, meaning that each function a single, well-defined action.

It is worth mentioning that although the majority of the functions `void` function, meaning that they do not return a value. This usually means that the function directly changes elements in the screen or perhaps the value of the data structure directly (e.g. in game, there are functions that directly change the matrix) and therefore does not need to return.

Some functions are auxiliary, while others are made to be a smaller subset of a larger function. For example, the function `print_table()` in **leaderboard.h** relies on helpers functions such as `print_sort_title()`, `print_headers()` and `print_row()`.

6.2 menu.h

1. `int menu()`
   Creates a menu in a subwindow, displays the options and allows the user to navigate through the options and select one. The menu handles resizing of the window for aesthetic purposes.

   - **Input**: none
   - **Return**: returns the selected option by the user to the main (1: Play; 2: Scoreboard; 3: Info; 4: Exit)

2. `void menu_highlight_option(WINDOW * menu, char * option[], int optnum)`
   It is an auxiliary function to `int menu()`. It prints the menu options and highlights the option in focus

   - **Input**: pointer to the window that it should modify, an array of string which are the options and, the number which signs the option to be highlighted
   - **Return**: none. The function prints the options in the screen, highlight the one in focus and updates the screen

3. `void center_string(WINDOW * win, int row, char * str)`
   Receives a string and prints in the horizontal center of the screen in the specified row

   - **Input**: pointer to the window where will the text will be displayed, the row (Y) and the string that will be centered.
   - **Return**: none. Adds the text into the screen (but does not refresh the window, to allow other functions to precisely control the refreshing of the screen)

6.3 game.h

The functions defined in **game.h** mostly follows a naming standard to improve readability and maintainability. The standard naming format is:

`type game_functionality(parameters)`

This allows to easily understand what is the purpose of the function and understand the code rapidly. There are a total of 19 functions defined in **game.h**. The respective more detailed explanation can be seen in the source code, as this is a more of a brief description of the purpose, input paremeters and return types of each function.

1. `void game_play()`
   It works as the **main game loop** and the most crucial function. It works as a main loop for the game.

   - **Input**: none
   - **Return**: none

2. `void init_colors()`
   Initialize all the colors and colors pairs that will be used in the game

   - **Input**: none
   - **Return**: none

3. `bool game_win_create()`
   Creates all the game visual structure by initializing a main window and subwindows, and add static text and elements to the respective window, such as borders and boxes around the windows.

   - **Input**: none, due to the windows being global variables
   - **Return**: Boolean value. It returns TRUE only if all the windows were created successfully, otherwise it returns FALSE.

4. `Matrix game_playfield_matrix(int height, int width)`
   Allocates dynamically a two-dimensional integer array which will be responsible for tracking the states of the game, namely the locked pieces.

   - **Input**: Pointer to a two-dimensional array (`typedef int** Matrix`), the height and the width of the array
   - **Return**: returns the pointer to the first element of the two-dimensional array with size heigth × width

5. `void game_playfield_update(Matrix game_matrix)`
   Updates the playfield window according to the state of the matrix.

   - **Input**: Pointer to a two-dimensional integer array
   - **Return**: none

6. `void game_draw_piece(int y, int x, int type, int orientation, int del)`
   Generates or erases a piece in the screen according to the position, type, orientation.

   - **Input**: The Y position, the X position, the type of the piece, the orientation and if the piece should be drawn or erased
   - **Return**: none. It updates the playfield with the current state of the matrix

7. `void game_update_next(int type)`
   Displays the next piece that will be placed into the playfield in the window win_next.

   - **Input**: Integer which describes the type of the piece
   - **Return**: none. It updates the win_next window with the next piece

8. `bool game_check_position(Matrix game_matrix, int y, int x, int type, int orientation)`
   Checks if a piece collides with the border of the playfield or other pieces. It is primarily used to check if a piece can move or rotate to a new position (left, right or down).

   - **Input**: Pointer to a 2D matrix where the locked pieces are stored, the horizontal and vertical position, the type of the piece and its orientation
   - **Return**: boolean value. Returns TRUE if the position of the piece does not collide with the borders or overlap with other pieces, otherwise returns FALSE

9. `int game_drop_piece(Matrix game_matrix, int type)`
   It is one of the core functions of the game and where the keyboards inputs are registered. It is responsible for ”dropping” a piece into the playfield, allowing the user to manipulate the piece according to the commands. Additionally, it allows the user to quit or pause the game.

   - **Input**: Pointer to the game matrix and type of piece which will be dropped
   - **Return**: returns an integer which is either a status or a score. If the user drops a piece softly or hard, the function will return the amount of earned points. Or if the user presses quit in either the game or pause menu, it will return a status (-1) to indicate the intention of exiting the game to the main loop

10. `void game_lock_piece(Matrix game_matrix, int y, int x, int type, int orientation)`
When it is called, it registers the piece with the respective position, type and orientation to the game matrix. As previously explained, when the piece is in the playfield, it is not registered in the matrix itself, only when it becomes locked.

- **Input**: pointer to a two-dimensional array, the vertical and horizontal position of the piece, the type and the orientation.
- **Return**: none (does not return a value), but modifies the value of the game matrix

11. `bool game_check_row(Matrix game_matrix, int row)`
Checks if a row is full in the game matrix (a row is considered full if all positions does not contain empty spots, i.e. a zero).

- **Input**: pointer to the matrix and the number of the row
- **Return**: boolean value. It returns FALSE if there is a empty spot (value 0), TRUE if every spot if filled

12. `void game_remove_row(Matrix game_matrix, int row)`
Removes a row if it is full and shifts all pieces above.

- **Input**: pointer to a matrix and the number of the row
- **Return**: none, directly modifies the values stored in the game matrix

13. `STATS game_check_playfield(Matrix game_matrix, STATS score)`
Scans the visible playfield area for completed lines. If there are lines completed, it removes them and counts the amount of erased lines (maximum is four at once)

- **Input**: pointer to a matrix and a structure STATS which stores the points and number of erased lines
- **Return**: returns the incremented value of the structure STATS

14. `void game_stats_update(STATS score)`
Updates the points and number of cleared lines in the statistics window

- **Input**: structure STATS which stores the points and lines
- **Return**: none

15. `int game_pause()`
Pauses the game and displays options to the user

- **Input**: None (not necessary as the window variables are global)
- **Return**: Returns the integer which describes the user choice (1 to continue or -1 to quit the game)

16. `bool game_check_gameover(Matrix game_matrix)`
Checks if there are any pieces in the buffer area, which signals that some piece reached the top of the playfield, which means that the game is over

- **Input**: pointer to a matrix
- **Return**: returns a boolean value. If some piece is in the buffer zone, returns TRUE (i.e. that the game is over), otherwise FALSE (the game is not over)

17. `void game_over_display()`
Creates a temporary window inside the playfield to display the gameover text when triggered

- **Input**: None (the window variable is global)
- **Return**: None (does not return value), but displays the gameover message in the terminal

18. `void game_save_score(STATS score)`
Creates the scores.txt file if necessary or appends the final score, number of cleared lines, date and time to the file

- **Input**: structure STATS with the final score and number of cleared lines
- **Return**: none

19. `void game_win_delete(Matrix game_array)`
Frees the two-dimensional array, erases the window and deletes them

- **Input**: pointer to a two-dimensional array
- **Return**: none

6.4 leaderboard.h

For this module, there are two categories of functions:

1. **Display functions**: functions that will display data into the screen
2. **File input and output functions**: read, sort and create files, as well as creating and deleting linked lists responsible for reading the data

1. `void scoreboard()`
Works as the main function of the scoreboard window.

- **Input**: none
- **Return**: none

2. `void print_sort_title(WINDOW * win, int row, int sort)`
Prints the respective title of the leaderboard table (which is according to the sorting) with bold and underline at the center of a row.

- **Input**: pointer of the window where the title will be printed, the row and the sorting mode
- **Return**: none

3. `void print_headers(WINDOW * win, int row)`
Prints the header of the table.

- **Input**: pointer to the window where the header will be displayed and the row
- **Return**: none

4. `void print_row(WINDOW * win, GAME_DATA data, int row)`
Transforms the entry into a string and prints it into the window in the specified row.

- **Input**: pointer to the window, the structure GAME_DATA and the row number
- **Return**: none

5. `void print_table(WINDOW * win, int sort, int len)`
   Calls the function to load the entries into the linked list according to the sorting type and then prints the title, headers and the sorted entries

   - **Input**: pointer to the window, the sorting type and the length of the list
   - **Return**: none

6. `void win_delete(WINDOW * win)`
   Erases the scoreboard window and deletes the windos structures.

   - **Input**: pointer of the window to be erased and deleted
   - **Return**: none

7. `NODE * linked_list_create(NODE * head, GAME_DATA data)`
   Creates the first node of the linked list if necessary and inserts the other nodes to the end of the list.

   - **Input**: a struct node pointer
   - **Return**: pointer to the first element of the list

8. `void linked_list_delete(NODE * head)`
   Deletes a linked list recursively

   - **Input**: pointer of type NODE to the first element of a linked list
   - **Return**: none

9. `NODE * load_sorted_list(NODE * head, int sort, int len)`
   Loads the entries of a linked list with maximum size of 10 entries, according to the sorting method.

   - **Input**: initial pointer to a linked list (the linked list is created inside the function), the type of sorting and the length of the list
   - **Return**: pointer to the first node of the linked list

10. `int sort_compare(const void* a, const void* b)`
    Comparison function necessary for the `qsort()` function. In this case, it compares between the points of two different entries

    - **Input**: two void pointers for the respective element, which should be typecasted into the necessary type
    - **Return**: returns the integer resultant of the comparison

11. `bool sort_file(WINDOW * win, char * filename, char * out, int * len)`
    Sorts the scores files in a descending order

    - **Input**: the window where the error messages will be displayed, the filename of the input file, the filename of the output and a pointer to an integer variable which tracks the number of entries (or length of the scores file)
    - **Return**: boolean value. Returns TRUE if the operation was completed successfully, otherwise returns FALSE and displays a message in the window warning which error occurred. Also, returns the length of the list of the parameter passed by address

6.5 info.h

1. `void win_info_display()`
   Works as the main loop of the info window. It creates the info window, gets the user input in order to switch between pages or exit

   - **Input**: user input via keyboard
   - **Return**: none

2. `void pages(WINDOW * win, int page, int width, int heigth)`
   Defines ”pages” which can be switched according to the function call

   - **Input**: the window where the page will be printed, the number of the page, the width and heigth of the window
   - **Return**: none

3. `void centered_title(WINDOW * win, int row, char * str)`
   Receives a string, applies bold and underline, and display on the center of the row

   - **Input**: the window where the title will be displayed, the row and the string
   - **Return**: none

4. `void text_wrapper(WINDOW * win, int row, int width, char * str)`
   Allows a text to be displayed in a window without splitting words

   - **Input**: the window where the text will be printed, the starting row, the width of the window and the string
   - **Return**: none

References

[1] D. Gookin, *Programmer's Guide to NCurses*. Wiley Publishing, Inc, 2007.

[2] "ncurses." `https://invisible-island.net/ncurses/man/ncurses.3x.html`.