sep 10, 19 18:04 **sudoku.h** Page 1/1

```c
#ifndef SUDOKU_H
#define SUDOKU_H

#include "board.h"
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

/* *****************************************************************
 *                      ESTRUCTURAS
 * *****************************************************************/
 struct sudoku {
   board_t* board;
 };
 typedef struct sudoku sudoku_t;

/* *****************************************************************
 *                      FUNCIONES DEL SUDOKU
 * *****************************************************************/

/*Inicializador de un sudoku.
Pre: Tablero es un archivo de texto que responde al formato establecido para el trabajo practico.
Devuelve un sudoku funcional*/
void sudoku_init(sudoku_t* sudoku);

int sudoku_put_value(sudoku_t* sudoku, char column, char value, char row);

/*Analiza si el sudoku esta en un estado valido o no.
Devuelve 0 si es correcto. 1 en caso contrario.*/
void sudoku_play(sudoku_t* sudoku, char mode, char* buffer);

/*Libera la memoria asociada al sudoku*/
void sudoku_release(sudoku_t* sudoku);

#endif
```

sep 10, 19 18:04 **sudoku.c** Page 1/1

```c
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "sudoku.h"
#define SIZE 10

/* *****************************************************************
 *                      Functions
 * *****************************************************************/

/*Inicializa un sudoku.*/
void sudoku_init(sudoku_t* sudoku){
  FILE* fp;
  fp = fopen("board.txt","r");
  board_t* board = malloc(sizeof(board_t));
  board_init(board,fp);
  sudoku→board = board;
}

int sudoku_put_value(sudoku_t* sudoku, char column, char value, char row){
  return board_put_value(sudoku→board,value,column,row);
}

void sudoku_play(sudoku_t* sudoku, char mode, char* buffer){
  if(mode ≡ 'g'){
    board_print(sudoku→board,buffer);
  }
  if (mode ≡ 'v'){
    buffer[0] = '\0';
    if (board_verify(sudoku→board)){
      snprintf(buffer,SIZE,"%s","OK\n");
    }else{
      snprintf(buffer,SIZE,"%s","ERROR\n");
    }
  }
  if (mode ≡ 'r'){
    board_clear(sudoku→board);
  }
}

/*Libera la memoria asociada al sudoku*/
void sudoku_release(sudoku_t* sudoku){
  board_release(sudoku→board);
  free(sudoku);
}
```

sep 10, 19 18:04 **socket.h** Page 1/1

```c
#ifndef SOCKET_H
#define SOCKET_H
#define _POSIX_C_SOURCE 200112L

typedef struct{
  int fd;
}socket_t;

int socket_send(socket_t* socket, const void* buffer, size_t lenght);

int socket_receive(socket_t* skt, void* buffer, size_t lenght);

void socket_init(socket_t* socket);

void socket_release(socket_t* socket);

int socket_connect(socket_t* socket,const char* host, const char* service);

int socket_bind_and_listen(socket_t* socket,const char* service);
#endif
```

sep 10, 19 18:04 **socket.c** Page 1/2

```c
#define _POSIX_C_SOURCE 200112L
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include "socket.h"
#define BUF_SIZE 500
#define LISTEN_BACKLOG 50
#define RESPONSE_MAX_LEN 1024

void socket_init(socket_t* skt){
  skt→fd = −1;
}

int socket_send(socket_t* skt, const void* buffer, size_t lenght){
  int bytes_sent = 0;
  const char* sent = buffer;
  bool are_we_connected = (skt→fd ≠ −1);
  while (lenght > bytes_sent ∧ are_we_connected){
    int s = send(skt→fd, &sent[bytes_sent], lenght − bytes_sent, MSG_NOSIGNAL);
    if (s > 0) {
      bytes_sent += s;
    }else {
      printf("Error: %s\n", strerror(errno));
      are_we_connected = false;
     }
  }
  return bytes_sent;
}

int socket_receive(socket_t* skt, void* buffer, size_t lenght){
  bool are_we_connected = (skt→fd ≠ −1);
  char* received = buffer;
  int bytes_received = 0;
  while (lenght > bytes_received ∧ are_we_connected){
    int s = recv(skt→fd, &received[bytes_received], \
    RESPONSE_MAX_LEN − bytes_received − 1, 0);
    if (s > 0) {
      bytes_received += s;
    }else {
      printf("Error: %s\n", strerror(errno));
      are_we_connected = false;
      return −1;
    }
  }
  return bytes_received;
}

void socket_release(socket_t* skt){
  if (skt→fd ≠ −1){
    close(skt→fd);
  }
  free(skt);
}

int socket_connect(socket_t* skt,const char* host, const char* service){
  struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, s;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
```

sep 10, 19 18:04 **socket.c** Page 2/2

```c
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = 0;
  s = getaddrinfo(host,service,&hints,&result);
  if (s ≠ 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    freeaddrinfo(result);
    return 1;
  }
  for (rp = result; rp≠ NULL; rp = rp→ai_next){
    sfd = socket(rp→ai_family,rp→ai_socktype,0);
    if(sfd ≡ −1){
      continue;
    }
    if(connect(sfd,rp→ai_addr,rp→ai_addrlen) ≠ −1){
      break;
    }
    close(sfd);
  }
  if (rp ≡ NULL) {
    freeaddrinfo(result);
        fprintf(stderr, "Could not connect\n");
        return 1;
  }
  freeaddrinfo(result);
  skt→fd = sfd;
  return s;
}

int socket_bind_and_listen(socket_t* skt,const char* service){
  struct addrinfo hints;
  struct addrinfo* results,*rp;
  int sfd,s;
  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  s = getaddrinfo(NULL,service,&hints,&results);
  if (s ≠ 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
  }
  for (rp = results; rp≠ NULL; rp = rp→ai_next){
    sfd = socket(rp→ai_family,rp→ai_socktype,0);
    if(sfd ≡ −1){
      continue;
    }
    if(bind(sfd,rp→ai_addr,rp→ai_addrlen) ≡ 0){
      break;
    }
    close(sfd);
  }
  if (rp ≡ NULL){
    fprintf(stderr, "Could not bind\n");
        exit(EXIT_FAILURE);
  }
  freeaddrinfo(results);
  listen(sfd,LISTEN_BACKLOG);
  int peerFd = accept(sfd,NULL,NULL);
  skt→fd = peerFd;
  return peerFd;
}
```

sep 10, 19 18:04 **server_controller.h** Page 1/1

```c
#include <stdio.h>
#include "sudoku.h"
#include <string.h>
#include "socket.h"

typedef struct{
  socket_t* socket;
  sudoku_t* sudoku;
  char buffer[722];
}server_controller_t;

void server_controller_init(server_controller_t* server);

void server_controller_start(server_controller_t* controller, char* server);

void server_controller_procesate(server_controller_t* controller, char* msg);

void server_controller_release(server_controller_t* controller);

int server_controller_send(server_controller_t* server);


```

sep 10, 19 18:04 **server_controller.c** Page 1/1

```c
#include <stdio.h>
#include <string.h>
#include "server_controller.h"
#define L 40

void server_controller_init(server_controller_t* controller){
  socket_t* socket = malloc(sizeof(socket_t));
  sudoku_t* sudoku = malloc(sizeof(sudoku_t));
  socket_init(socket);
  sudoku_init(sudoku);
  controller→socket = socket;
  controller→sudoku = sudoku;
  controller→buffer[0]='\0';
}

void server_controller_start(server_controller_t* controller, char* server){
  int check = 0;
  socket_bind_and_listen(controller→socket,server);
  char buffer[4] = "  ";
  while(check ≠ -1){
    check = socket_receive(controller→socket,buffer,4);
    if (check ≡ 0){
      break;
    }
    server_controller_procesate(controller,buffer);
    check = server_controller_send(controller);
    if (check ≡ 0){
      break;
    }
  }
}

void server_controller_procesate(server_controller_t* controller, char* msg){
  char* mode = msg;
  if(mode[0] ≡ 'r'){
    sudoku_play(controller→sudoku,mode[0],NULL);
    sudoku_play(controller→sudoku,'g',controller→buffer);
  }else{
    if(mode[0] ≡ 'p'){
      int check = sudoku_put_value(controller→sudoku,msg[2],msg[3],msg[1]);
      if (check ≡ 0){
        sudoku_play(controller→sudoku,'g',controller→buffer);
      }else{
        snprintf(controller→buffer,L,"%s","La celda indicada no es modificable\n");
      }
    }else{
      sudoku_play(controller→sudoku,mode[0],controller→buffer);
    }
  }
}

int server_controller_send(server_controller_t* controller){
  __uint32_t num = strlen(controller→buffer);
  socket_send(controller→socket,&num,4);
  return socket_send(controller→socket,controller→buffer,num);
}

void server_controller_release(server_controller_t* controller){
  sudoku_release(controller→sudoku);
  socket_release(controller→socket);
  free(controller);
}
```

sep 10, 19 18:04 **main.c** Page 1/1

```c
#include <stdio.h>
#include <string.h>
#include "server_controller.h"
#include "client_controller.h"
#define ERROR 1

/* ******************************************************************
 *                              Main
 * *****************************************************************/

int main(int argc, char* argv[]) {
    /* Ejecuta todas las pruebas unitarias. */
  if (argc < 2){
    printf("âM-^@M-^KModo no soportado, el primer parÃ¡metro debe ser server o client\nâM-^@M-^K");
    return ERROR;
  }
  if ( strcmp(argv[1],"server") ≡0 ){
    server_controller_t* controller = malloc(sizeof(server_controller_t));
    server_controller_init(controller);
    server_controller_start(controller,argv[2]);
    server_controller_release(controller);
  }
  if(strcmp(argv[1],"client")≡0){
    client_controller_t* controller = malloc(sizeof(client_controller_t));
    client_controller_init(controller);
    cli_contr_start(controller,argv[2],argv[3]);
    //client_controller_release(controller);
  }
  return 0;
}
```

```c
#ifndef LIST_H
#define LIST_H

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

/* ******************************************************************
 *                        Definitions
 * *****************************************************************/
struct node {
  struct node* next;
  void* data;
  int state;
};
typedef struct node node_t;

typedef struct list{
  node_t* first;
  node_t* last;
  size_t size;
} list_t;

typedef struct{
  list_t* list;
  node_t* before;
  node_t* actual;
}list_iter_t;

/* ******************************************************************
 *                        Functions of node
 * *****************************************************************/
 /*Initializes an empty node*/
 void node_init(node_t* node,char* data);

 /*Reemplaces the value in the node*/
 int node_replace(node_t* node, char* newData,void destroy_data(void *extra));

 /*Restarts the value to 0 if the state is 0*/
 void node_restart(node_t* node);
/* ******************************************************************
 *                        FUNCTIONS OF LIST
 * *****************************************************************/
/*Initializes an empty list*/
void list_init(list_t* list);

/*Returns the size of the list*/
size_t list_size(const list_t *list);

/*Inserts data in the list in the last position
Post: Using a bool,informs if was possible to insert*/
bool list_insert(list_t* list, void* data,int value);

/*Inserts data in the first position*/
bool list_insert_first(list_t* list, void* data);

/*Inserts in the last position*/
bool list_insert_last(list_t* list, void*data);

/*Informs if the list is empty*/
bool list_is_empty(const list_t* list);

/*Deletes the first element in the list and returns it
Pre: If you want to use the externarl iterator, dont use this*/
void* list_delete_first(list_t* list);

```



```c
/*Iterates the list*/
void list_iterate(list_t* list, bool \
(*visit)(void *data, char* extra), char *extra);

/*Restarts the values to 0 if the state is 0*/
void list_restart(list_t* list);

/*Free the memory asociated*/
void list_release(list_t *list, void destroy_data(void *extra));

char* list_return_especific_position_data(list_t* list, int position);

/*Initialize the external iterator*/
void list_iter_init(list_iter_t* iterator, list_t *list);

/*Moves forward the iterator*/
bool list_iter_forward(list_iter_t *iter);

/*Moves forward the iterator "amount" times*/
void list_iter_multiple_forwards(list_iter_t* iter, int amount);

/*Informs if the iterator is at the end of the list*/
bool list_iter_end(const list_iter_t *iter);

/*Returns the element pointed by the iterator*/
void *list_iter_actual(const list_iter_t* iter);

/*Returns the actual node*/
node_t* list_iter_node_actual(list_iter_t* iter);

/*Release the memory asociated*/
void list_iter_release(list_iter_t *iter);

#endif

```

```c
#include "list.h"
#define FIRST 0
#define END 1

/* ******************************************************************
 *                      FUNCIONES DEL NODO
 * *****************************************************************/

/*Initializes an empty node*/
 void node_init(node_t* node, char* data){
  node→data = data;
  node→next = NULL;
  int aux = data[0]−48;
  if (aux ≡ 0){
    node→state = 0;
  }else{
    node→state = 1;
  }
}


 int node_replace(node_t* node, char* newData, void destroy_data(void *extra)){
  char* dataOld = node→data;
  if (node→state ≡ 1){
     return 1;
  }
  if (destroy_data){
      destroy_data(dataOld);
  }
  node→data = newData;
  return 0;
 }

void node_restart(node_t* node){
  if (node→state ≡ 0){
    char* value = malloc(sizeof(char*));
    char* aux  = "0";
    value[0] = aux[0];
    node_replace(node,value,free);
  }
}

/* ******************************************************************
 *                      FUNCIONES DE LA LISTA
 * *****************************************************************/

/*Initializes an empty list*/
void list_init(list_t* list){
  list→size = 0;
  list→first = NULL;
  list→last = NULL;
}

/*Returns the size of the list*/
size_t list_size(const list_t *list){
  return list→size;
}

/*Inserts data in the list in the last position
Post: Using a bool,informs if was possible to insert*/
bool list_insert(list_t* list, void* data,int value){
  node_t* node = malloc(sizeof(node_t));
  node_init(node,data);
  if (list_is_empty(list)){
    list→first = node;
    list→last = node;
```



```c
  }else if (value ≡ FIRST){
    node_t* nodeOld = list→first;
    list→first = node;
    node→next = nodeOld;
  }else{
    list→last→next = node;
    list→last = node;
  }
  list→size++;
  return true;
}

/*Inserts data in the first position*/
bool list_insert_first(list_t* list, void* data){
  return list_insert(list,data,FIRST);
}

/*Inserts in the last position*/
bool list_insert_last(list_t* list, void*data){
  return list_insert(list,data,END);
}

/*Informs if the list is empty*/
bool list_is_empty(const list_t* list){
  return (list→size ≡ 0);
}

/*Deletes the first element in the list and returns it
Pre: If you want to use the externarl iterator, dont use this*/
void* list_delete_first(list_t* list){
  if (list_is_empty(list)){
    return NULL;
  }
  node_t* node = list→first;
  void* data = node→data;
  if (list→size ≡ 1){
    list→first = NULL;
    list→last = NULL;
  }else{
    list→first = list→first→next;
  }
  list→size−−;
  free(node);
  return data;
}

/*Applies the function *visit to all the elements in the list*/
void list_iterate(list_t* list, \
bool (*visit)(void *data, char* extra), char *extra){
  if (¬list_is_empty(list)){
    node_t* actual = list→first;
    while (visit(actual→data,extra) ∧ actual→next ≠ NULL){
      actual = actual→next;
      }
  }
}

void list_restart(list_t* list){
  list_iter_t* iter = malloc(sizeof(list_iter_t));
  list_iter_init(iter,list);
  while(¬list_iter_end(iter)){
    node_restart(list_iter_node_actual(iter));
    list_iter_forward(iter);
  }
  list_iter_release(iter);
}
```

```
133
134 /*Returns the data stored in the position given
135 Pre: Position < list->list*/
136 char* list_return_especific_position_data(list_t* list, int position){
137   int i = 0;
138   node_t* node = list→first;
139   while( i < position ){
140     node = node→next;
141     i++;
142   }
143   return node→data;
144 }
145
146 /*Free the memory asociated*/
147 void list_release(list_t* list, void destroy_data(void * extra)){
148   while (¬list_is_empty(list)){
149     void* data = list_delete_first(list);
150     if (destroy_data){
151       destroy_data(data);
152     }
153   }
154   free(list);
155 }
156
157 /*Initialize the external iterator*/
158 void list_iter_init(list_iter_t* iterator,list_t* list){
159   iterator→list = list;
160   iterator→before = NULL;
161   iterator→actual = list→first;
162 }
163
164 /*Moves forward the iterator*/
165 bool list_iter_forward(list_iter_t *iter){
166   if (list_iter_end(iter)){
167         return false;
168   }
169   iter→before = iter→actual;
170   iter→actual = iter→actual→next;
171   return true;
172 }
173
174 /*Informs if the iterator is at the end of the list*/
175 bool list_iter_end(const list_iter_t *iter){
176     return (iter→actual≡NULL);
177 }
178 /*Moves forward the iterator "amount" times*/
179 void list_iter_multiple_forwards(list_iter_t* iter, int amount){
180   int i = 0;
181   while (i< amount){
182     list_iter_forward(iter);
183     i++;
184   }
185 }
186
187 /*Returns the element pointed by the iterator*/
188 void *list_iter_actual(const list_iter_t* iter){
189   if (list_iter_end(iter)){
190     return NULL;
191   }
192   return iter→actual→data;
193 }
194
195 node_t* list_iter_node_actual(list_iter_t* iter){
196   return iter→actual;
197 }
198
```



```
199 /*Release the memory asociated*/
200 void list_iter_release(list_iter_t *iter){
201     free(iter);
202 }
```

sep 10, 19 18:04 **client_controller.h** Page 1/1

```c
#include <stdio.h>
#include "sudoku.h"
#include <string.h>
#include "socket.h"

typedef struct{
  socket_t* socket;
  sudoku_t* sudoku;
  char buffer[722];
}client_controller_t;

void client_controller_init(client_controller_t* server);

int client_controller_validate(char*buffer);

void client_controller_proc(client_controller_t* contr, char* msg, char* msgp);

void cli_contr_start(client_controller_t*cntr,const char*srvr,const char*sc);

void client_controller_recv(client_controller_t* controller, char* buffer);

void client_controller_send(client_controller_t* controller,char* msgp);

void client_controller_release(client_controller_t* controller);
```

sep 10, 19 18:04 **client_controller.c** Page 1/2

```c
#include <stdio.h>
#include <string.h>
#include "client_controller.h"
#include <unistd.h>
#define SIZE 5


void client_controller_init(client_controller_t* controller){
  socket_t* socket = malloc(sizeof(socket_t));
  sudoku_t* sudoku = malloc(sizeof(sudoku_t));
  socket_init(socket);
  sudoku_init(sudoku);
  controller→socket = socket;
  controller→sudoku = sudoku;
}

void cli_contr_start(client_controller_t* controller,\
const char* server,const char* service){
  if (socket_connect(controller→socket,server,service) ≠ 1){
    while (true){
      char msgp [5] = {0,0,0,0};
      char buffer[20] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,};
      char* arg = fgets(buffer,19,stdin);
      int check = 0;
      if (arg ≡ NULL){
        break;
      }
      msgp[strlen(msgp)−1] = '\0';
      buffer[strlen(buffer)−1]='\0';
      if(strncmp(buffer,"exit",4) ≡ 0){
        printf("Saliendo del juego\n");
        break;
      }
      check = client_controller_validate(buffer);
      if (check ≠ 0){
        if(check ≡ 1){
          printf("Comando no soportado\n");
        }else{
          printf("Error en los indices. Rango soportado: [1,9]\n");
        continue;
        }
      }
      client_controller_proc(controller,buffer,msgp);
      client_controller_send(controller,msgp);
      client_controller_recv(controller,buffer);
    }
  }
}

void client_controller_send(client_controller_t* controller,char* msgp){
  socket_send(controller→socket,msgp,4);
}

void client_controller_recv(client_controller_t* controller, char* buffer){
  __uint32_t lenght;
  socket_receive(controller→socket,&lenght,4);
  socket_receive(controller→socket,buffer,lenght);
  buffer[lenght]='\0';
  printf("%s",buffer);
}

int client_controller_validate(char*buffer){
  if (strlen(buffer) ≠ 12){
    if (strncmp(buffer,"get",3) ≡ 0 ∨ strncmp(buffer,"verify",6) ≡ 0 ∨\
    strncmp(buffer,"reset",5) ≡ 0){
      return 0;
```

```
67      }
68      if(strncmp(buffer,"put",2) ≡ 0){
69        return −1;
70      }
71      return 1;
72    }
73    if(buffer[4] ≤'9' ∧ buffer[9]≤ '9' ∧ buffer[11] ≤'9'){
74      return 0;
75    }
76    return −1;
77  }
78
79  void client_controller_proc\
80  (client_controller_t* controller, char* msg, char* msgp){
81    char mode = msg[0];
82    if(mode ≠ 'p'){
83      snprintf(msgp,SIZE,"%c%c%c%c",msg[0],' ',' ',' ');
84    }else{
85      snprintf(msgp,SIZE,"%c%c%c%c",'p',msg[11],msg[9],msg[4]);
86    }
87  }
88
89  void client_controller_release(client_controller_t* controller){
90    sudoku_release(controller→sudoku);
91    if (controller→socket→fd ≠1){
92      socket_release(controller→socket);
93    }
94    free(controller);
95  }
```



```
1   #ifndef BOARD_H
2   #define BOARD_H
3
4   #include <stdbool.h>
5   #include <stdlib.h>
6   #include <stdio.h>
7   #include "list.h"
8
9   /* ********************************************************************
10   *                        Definitions
11   * ********************************************************************/
12   struct board {
13    list_t* list;
14  };
15   typedef struct board board_t;
16
17  /* ********************************************************************
18   *                          Functions
19   * ********************************************************************/
20   /*The board is inited
21   POST: The file is closed*/
22   void board_init(board_t* board, FILE* fp);
23
24  /*Fills list_iter_actual with the char* string after deleting the " " in it
25  PRE: The iter doesn't point to null*/
26  void board_fill_list(char* str, list_iter_t* iter);
27
28  /*Fills nine list for using in a board_t
29  PRE: The nine row are nodes in the list*/
30  void board_fill_rows(list_t* rows, FILE* fp);
31
32  /*The value is putted in the position given
33   Pre: The positionAndValue's fortmat is <numero> in <fila>,<columna>*/
34  int board_put_value(board_t* board,char valueC, char columnC, char rowC);
35
36  /*Prints the double border*/
37  void board_print_border(char* buffer);
38
39  /*Prints the middle border*/
40  void board_print_middle_border(char* buffer);
41
42  /*Prints the row with the next format
43  U X | X | X U X | X | X U X | X | X U*/
44  void board_print_row(list_t* row,char* buffer);
45
46  /*Prints the board*/
47  void board_print(board_t* board, char* buffer);
48
49  /*Puts all the values in a array = 0*/
50  void restart_array(int* array);
51
52  /*Returns array filled with all the lists in board->columns*/
53  void board_make_array_list(board_t* board, list_t** array);
54
55  /*For internal use of the verify functions. Dont use*/
56  bool board_check_values(int* array, char* value);
57
58  /*Verifies if there is a repeate dvalue in a single row*/
59  bool board_verify_row(list_t* list);
60
61  /*Verifies if there is a repeated value in all the rows*/
62  bool board_verify_rows(board_t* board);
63
64  /*Verifies if there is a repeated value in the sector*/
65  bool board_verify_sector(int* array, list_t** listArray, int x, int y);
66
```

```
67  /*Verifies if there is a repeated value in all of the sectors*/
68  bool board_verify_sectors(board_t* board);
69
70  /*Verifies if there is a repeated value in all the columns*/
71  bool board_verify_columns(board_t* board);
72
73  bool board_verify(board_t* board);
74
75  /*Clears the board. The original values taked in the
76   board_init remains the same*/
77   void board_clear(board_t* board);
78
79  /*Releases the memory asociated to the board*/
80   void board_release(board_t* board);
81   #endif
```



```
1   #include <stdbool.h>
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <string.h>
5   #include "board.h"
6
7   #define ORIGINAL 0
8   #define NEW 1
9
10  /* ****************************************************************
11   *                          Functions
12   * ****************************************************************/
13
14   /*The board is inited
15   POST: The file is closed*/
16   void board_init(board_t* board, FILE* fp){
17    list_t* columns = malloc(sizeof(list_t));
18    list_init(columns);
19    board→list = columns;
20    int i;
21    for(i =0; i<9; i++){
22      list_t* row = malloc(sizeof(list_t));
23      list_init(row);
24      list_insert_last(board→list,row);
25    }
26    board_fill_rows(board→list,fp);
27    fclose(fp);
28   }
29
30  /*Fills list_iter_actual with the char* string after deleting the " " in it
31  PRE: The iter doesn't point to null*/
32  void board_fill_list(char* stringA, list_iter_t* iter){
33    char* word;
34    char* split;
35    char** aux = malloc(sizeof(char**));
36    list_t* list = list_iter_actual(iter);
37    list_init(list);
38    split = __strtok_r(stringA," ",aux);
39    while(split ≠ NULL){
40      word = malloc(sizeof(char));
41      *word = *split;
42      split = __strtok_r(NULL, " ",aux);
43      list_insert_last(list,word);
44    }
45    list_iter_forward(iter);
46    free(aux);
47  }
48
49  /*Fills nine list for using in a board_t
50  PRE: The nine row are nodes in the list*/
51  void board_fill_rows(list_t* rows, FILE* fp){
52    list_iter_t* iter = malloc(sizeof(list_iter_t));
53    list_iter_init(iter,rows);
54    char stringA[18*9+1];
55    while (fgets(stringA,sizeof(stringA),fp) ≠ 0){
56      board_fill_list(stringA,iter);
57    }
58    list_iter_release(iter);
59  }
60
61  /*The value is putted in the position given
62   Pre: The positionAndValue's fortmat is <numero> in <fila>,<columna>*/
63  int board_put_value(board_t* board, char valueC, char columnC, char rowC){
64     int i;
65     int column = columnC −48;
66     char* value = malloc(sizeof(char*));
```

```
    value[0] = valueC;
    int row = rowC -48;
    list_iter_t* iter = malloc(sizeof(list_iter_t));
    list_iter_init(iter,board→list);
    list_iter_multiple_forwards(iter,column-1);
    list_iter_t* iterRow = malloc(sizeof(list_iter_t));
    list_iter_init(iterRow,list_iter_actual(iter));
    list_iter_multiple_forwards(iterRow,row-1);
    node_t* node = list_iter_node_actual(iterRow);
    i = node_replace(node,value,free);
    list_iter_release(iter);
    list_iter_release(iterRow);
    return i;
}

/*Prints the double border*/
void board_print_border(char* buffer){
  strncat(buffer,"U==========U==========U==========U\n",722);
}

/*Prints the middle border*/
void board_print_middle_border(char* buffer){
  strncat(buffer,"U---+---+---U---+---+---U---+---+---U\n",722);
}

/*Prints the row with the next format
U X | X | X U X | X | X U X | X | X U*/
void board_print_row(list_t* row,char* buffer){
  list_iter_t* iter = malloc(sizeof(list_iter_t));
  list_iter_init(iter,row);
  char rowNumbers[10];
  int i;
  for (i = 0; i<9; i++){
    char* number = list_iter_actual(iter);
    if(number ≠ NULL ∧ number[0] ≡ '0'){
      number = " ";
    }
    rowNumbers[i] = number[0];
    list_iter_forward(iter);
  }
  char linea[46];
  linea[45]='\0';
  snprintf(linea,sizeof(linea)
  ,"U %c | %c | %c U %c | %c | %c U %c | %c | %c U\n",\
  rowNumbers[0],\
  rowNumbers[1],rowNumbers[2],rowNumbers[3],  rowNumbers[4],rowNumbers[5],\
  rowNumbers[6],rowNumbers[7],rowNumbers[8]);
  strncat(buffer,linea,722);
}

/*Prints the board*/
void board_print(board_t* board, char* buffer){
  buffer[0] = '\0';
  list_iter_t* iter = malloc(sizeof(list_iter_t));
  list_iter_init(iter,board→list);
  int i;
  for (i = 0; i< 3; i++){
    board_print_border(buffer);
    board_print_row(list_iter_actual(iter),buffer);
    list_iter_forward(iter);
    board_print_middle_border(buffer);
    board_print_row(list_iter_actual(iter),buffer);
    list_iter_forward(iter);
    board_print_middle_border(buffer);
    board_print_row(list_iter_actual(iter),buffer);
    list_iter_forward(iter);
```



```
  }
  board_print_border(buffer);
  list_iter_release(iter);
}

/*Puts all the values in a array = 0*/
void restart_array(int* array){
  for (int i = 0; i<9; i++){
    array[i] = 0;
  }
}

/*Returns array filled with all the lists in board->columns*/
void board_make_array_list(board_t* board, list_t** array){
  list_iter_t* iter = malloc(sizeof(list_iter_t));
  list_iter_init(iter,board→list);
  int i = 0;
  while (¬list_iter_end(iter)){
    array[i] = list_iter_actual(iter);
    list_iter_forward(iter);
    i++;
  }
  list_iter_release(iter);
}

/*For internal use of the verify functions. Dont use*/
bool board_check_values(int* array, char* value){
  int number = value[0]-49;
  if (number < 0){
    return true;
  }
  if (array[number] ≡ 0){
    array[number] = 1;
    return true;
  }else{
    return false;
  }
}

/*Verifies if there is a repeate dvalue in a single row*/
bool board_verify_row(list_t* list){
  list_iter_t* iter = malloc(sizeof(list_iter_t));
  list_iter_init(iter,list);
  int check[9] = {0,0,0,0,0,0,0,0,0};
  restart_array(check);
  while(list_iter_node_actual(iter) ≠ NULL){
    if (¬board_check_values(check,list_iter_actual(iter))){
      list_iter_release(iter);
      return false;
    }
    list_iter_forward(iter);
  }
  list_iter_release(iter);
  return true;
}

/*Verifies if there is a repeated value in all the rows*/
bool board_verify_rows(board_t* board){
  list_iter_t* iter = malloc(sizeof(list_iter_t));
  list_iter_init(iter,board→list);
  while(list_iter_node_actual(iter)≠ NULL){
    if(¬board_verify_row(list_iter_actual(iter))){
      list_iter_release(iter);
      return false;
    }
    list_iter_forward(iter);
```

```
199   }
200   list_iter_release(iter);
201   return true;
202 }
203
204 /*Verifies if there is a repeated value in the sector*/
205 bool board_verify_sector(int* array, list_t** listArray, int x, int y){
206   int i = x;
207   for(int x = i; x < i + 3; x ++){
208     char* value = list_return_especific_position_data(listArray[x],y);
209     if (¬board_check_values(array,value)){
210       return false;
211     }
212     value = list_return_especific_position_data(listArray[x],y+1);
213     if (¬board_check_values(array,value)){
214         return false;
215     }
216     value = list_return_especific_position_data(listArray[x],y+2);
217     if (¬board_check_values(array,value)){
218       return false;
219     }
220   }
221   return true;
222 }
223
224 /*Verifies if there is a repeated value in all of the sectors*/
225 bool board_verify_sectors(board_t* board){
226   list_t* array[9];
227   int check[9] = {0,0,0,0,0,0,0,0,0};
228   board_make_array_list(board,array);
229   for(int i = 0; i <7; i= i+3){
230     restart_array(check);
231     for(int j = 0; j <7; j= j+3){
232       restart_array(check);
233       if(¬board_verify_sector(check,array,j,i)){
234         return false;
235       }
236     }
237   }
238   return true;
239 }
240
241 /*Verifies if there is a repeated value in all the columns*/
242 bool board_verify_columns(board_t* board){
243   list_t* array[9];
244   board_make_array_list(board,array);
245   int check[9] = {0,0,0,0,0,0,0,0,0};
246   for (int i = 0; i < 9; i ++){
247     restart_array(check);
248     for(int j = 0; j < 9; j ++){
249       char* value = list_return_especific_position_data(array[j],i);
250       if (¬board_check_values(check,value)){
251         return false;
252       }
253     }
254   }
255   return true;
256 }
257
258 bool board_verify(board_t* board){
259   if(¬board_verify_sectors(board)){
260     return false;
261   }
262   if(¬board_verify_columns(board)){
263     return false;
264   }
```



```
265   if(¬board_verify_rows(board)){
266     return false;
267   }
268   return true;
269 }
270
271
272 /*Clears the board. The original values taked in the
273  board_init remains the same*/
274  void board_clear(board_t* board){
275   list_iter_t* iter = malloc(sizeof(list_iter_t));
276   list_iter_init(iter,board→list);
277   while(¬list_iter_end(iter)){
278     list_restart(list_iter_actual(iter));
279     list_iter_forward(iter);
280   }
281   list_iter_release(iter);
282  }
283
284 /*Releases the memory asociated to the board*/
285  void board_release(board_t* board){
286   list_iter_t* iterator = malloc(sizeof(list_iter_t));
287   list_iter_init(iterator,board→list);
288   for(int i =0; i<9; i++){
289     list_t* row = list_iter_actual(iterator);
290     list_release(row,free);
291     list_iter_forward(iterator);
292   }
293   list_iter_release(iterator);
294   list_release(board→list,NULL);
295   free(board);
296 }
```

sep 10, 19 18:04 **Table of Content** Page 1/1

Table of Contents