

sep 24, 19 9:58

tablero.h

Page 1/1

```

1  #ifndef _TABLERO_H_
2  #define _TABLERO_H_
3  #include "celda.h"
4  #include "sector.h"
5
6
7  #define NRO_FILAS 9
8  #define NRO_COLUMNAS 9
9  typedef struct {
10     celda_t** celdas;
11     sector_t** sectores;
12 }tablero_t;
13
14 //Crea un tablero de 9 x 9 casilleros a partir de un archivo de texto
15 //que contiene los valores iniciales de las celdas. Estas celdas se
16 //identificaran como prefijadas y por lo tanto inmodificables.
17 //PRE: el arhivo debe tener el formato de una fila por linea, con
18 //los numeros para cada celda separados con espacios y se entiende
19 //como una celda con 0 como una celda vacia.
20 //POST: crea un tablero con los valores del archivo. Devuelve distinto
21 //de 0 si ocurre algun error
22 int tablero_crear(tablero_t* tablero, const char* ruta_archivo);
23
24 //Verifica que todas las celdas del tablero
25 //tengan valores que cumplan con las reglas:
26 // *En cada fila, columna y sector hay solamente
27 // numeros del 1 al 9 distintos o celdas vacias
28 //POST: Devuelve true si las reglas se cumplen para
29 //todas las celdas falso en caso contrario.
30 bool tablero_verificar(tablero_t* tablero);
31
32 //Devuelve una representacion del tablero como cadena
33 char* tablero_obtener(tablero_t* tablero);
34
35 //Modifica la celda en la posicion (fila, columna) del tablero poniendole
36 //el valor recibido.
37 //PRE: el tablero fue creado. Fila y columna deben ser dos valores validos
38 //POST: devuelve 0 si la celda se modifico correctamente, en caso de error:
39 // * 1 si (fila, columna) es una posicion invalida
40 // * 2 si la celda no es modificable
41 int tablero_modificar_celda(tablero_t* tablero, int fila, int col, int valor);
42
43 //Reinicia el tablero recibido a los valores prefijados
44 //en la creacion.
45 //POST: Reinicia el tablero eliminando los valores de las
46 //celdas no fijadas en la creacion y lo devuelve?
47 void tablero_reiniciar(tablero_t* tablero);
48
49 //Destruye el tablero recibido por parametro
50 //liberando todos los recursos utilizados
51 void tablero_destruir(tablero_t* tablero);
52
53 #endif

```

sep 24, 19 9:58

tablero.c

Page 1/4

```

1  #include "tablero.h"
2  #include "impresor.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #define TAMANIO_BUFFER_LECTURA (2 * NRO_COLUMNAS)
7
8  int digito_a_int(const char caracter){
9     return caracter - '0';
10 }
11
12 //Toma los caracteres en las posiciones pares, es decir, evita
13 //los separadores y convierte los que estes en estas posiciones
14 //a entero para crear una celda con ese valor. Si el valor es 0
15 //se entiende que la celda esta vacia. En caso contrario se entiende
16 //que es una de las celdas de pista y por lo tanto, no modificable.
17 celda_t* procesar_fila_leida(char* linea, size_t leidos){
18     celda_t* fila = malloc(sizeof(celda_t) * NRO_COLUMNAS);
19     if (fila == NULL){
20         return NULL;
21     }
22     for (int i = 0; i < leidos; i++){
23         if (i % 2 == 0){
24             int valor = digito_a_int(linea[i]);
25             if (valor == 0){
26                 celda_crear(&fila[i/2], valor, true);
27             }else{
28                 celda_crear(&fila[i/2], valor, false);
29             }
30         }
31     }
32     return fila;
33 }
34
35 int leer_archivo(const char* ruta_archivo, celda_t** filas){
36     FILE* archivo = fopen(ruta_archivo, "rt");
37     if (archivo == NULL){
38         fprintf(stderr, "No se pudo abrir el archivo\n");
39         return 1;
40     }
41     int fila_actual = 0;
42     size_t leidos;
43     char buffer[TAMANIO_BUFFER_LECTURA];
44     while ((leidos = fread(&buffer, 1, TAMANIO_BUFFER_LECTURA, archivo))){
45         if (leidos < TAMANIO_BUFFER_LECTURA-1){ //si la fila no esta completa
46             fprintf(stderr, "El archivo de entrada tiene filas incompletas\n");
47             fclose(archivo);
48             return 2;
49         }
50         celda_t* fila = procesar_fila_leida(buffer, leidos);
51         if (fila == NULL){
52             fprintf(stderr, "No hay memoria suficiente\n");
53             fclose(archivo);
54             return 3;
55         }
56         filas[fila_actual] = fila;
57         fila_actual++;
58     }
59     fclose(archivo);
60     return 0;
61 }
62
63 int agregar_sectores(tablero_t* tablero){
64     sector_t** sectores = malloc(sizeof(sector_t) * NRO_FILAS/3);
65     if (sectores == NULL){
66         return 1;

```

sep 24, 19 9:58

tablero.c

Page 2/4

```

67     }
68     for (int i = 0; i < 3; i++){
69         sector_t* fila_sector = malloc(sizeof(sector_t) * NRO_COLUMNAS/3);
70         if (fila_sector == NULL){
71             free(sectores);
72             return 1;
73         }
74         sectores[i] = fila_sector;
75         for (int j = 0; j < 3; j++){
76             sector_t sector;
77             if (sector_crear(&sector) != 0){
78                 return 1;
79             }
80             sectores[i][j] = sector;
81         }
82     }
83     for (int fila = 0; fila < NRO_FILAS; fila++){
84         for (int col = 0; col < NRO_COLUMNAS; col++){
85             sector_agregar_celda(&sectores[fila/3][col/3], &tablero->celdas[fila][col]
86         );
87     }
88     tablero->sectores = sectores;
89     return 0;
90 }
91
92 int tablero_crear(tablero_t* tablero, const char* ruta_archivo){
93     celda_t** filas = malloc( sizeof(celda_t*) * NRO_FILAS);
94     if ( !filas ){
95         fprintf(stderr, "No hay memoria suficiente\n");
96         return 3;
97     }
98     int salida;
99     if ((salida = leer_archivo(ruta_archivo, filas)) == 1){
100         fprintf(stderr, "Error al leer el archivo\n");
101         free(filas);
102         return salida;
103     }
104     tablero->celdas = filas;
105
106     if (agregar_sectores(tablero) != 0){
107         fprintf(stderr, "Error al crear los sectores\n");
108         free(filas);
109         return 4;
110     }
111
112     return 0;
113 }
114
115 void tablero_reiniciar(tablero_t* tablero){
116     for (int i = 0; i < NRO_FILAS; i++){
117         for (int j = 0; j < NRO_COLUMNAS; j++){
118             if ( celda_es_modificable(&tablero->celdas[i][j]) ) {
119                 celda_poner_numero(&tablero->celdas[i][j], 0);
120             }
121         }
122     }
123 }
124
125 //Cuenta las apariciones del numero recibido en la fila indicada
126 int contar_apariciones_columna(tablero_t* tablero, int numero, int columna){
127     int apariciones = 0;
128     for (int i = 0; i < NRO_FILAS; i++){
129         if (celda_obtener_numero(&tablero->celdas[i][columna]) == numero){
130             apariciones++;
131         }

```

sep 24, 19 9:58

tablero.c

Page 3/4

```

132     return apariciones;
133 }
134 //Cuenta las apariciones del numero recibido en la columna indicada
135 int contar_apariciones_fila(tablero_t* tablero, int numero, int fila){
136     int apariciones = 0;
137     for (int i = 0; i < NRO_COLUMNAS; i++){
138         if (celda_obtener_numero(&tablero->celdas[fila][i]) == numero){
139             apariciones++;
140         }
141     }
142     return apariciones;
143 }
144
145 bool verificar_celda(tablero_t* tablero, int fila, int columna){
146     int numero = celda_obtener_numero(&tablero->celdas[fila][columna]);
147     if (numero == 0){ //esta vacia
148         return true;
149     }
150     if (contar_apariciones_columna(tablero, numero, columna) > 1){
151         return false;
152     }
153     if (contar_apariciones_fila(tablero, numero, fila) > 1){
154         return false;
155     }
156     if (sector_contar_apariciones(&tablero->sectores[fila/3][columna/3], numero) > 1)
157     {
158         return false;
159     }
160     return true;
161 }
162
163 bool tablero_verificar(tablero_t* tablero){
164     for (int i = 0; i < NRO_FILAS; i++){
165         for (int j = 0; j < NRO_COLUMNAS; j++){
166             if (!verificar_celda(tablero, i, j)){
167                 return false;
168             }
169         }
170     }
171     return true;
172 }
173
174 int tablero_modificar_celda(tablero_t* tablero, int fil, int col, int valor){
175     if ((fil < 0) || (fil >= NRO_FILAS) || (col < 0) || (col >= NRO_COLUMNAS)){
176         return 1;
177     }
178     celda_t* celda = &tablero->celdas[fil][col];
179
180     if (!celda_es_modificable(celda)){
181         return 2;
182     }
183     celda_poner_numero(celda, valor);
184     return 0;
185 }
186
187 char* tablero_obtener(tablero_t* tablero){
188     return impresor_imprimir_celdas((const celda_t **)tablero->celdas, \
189     NRO_FILAS, NRO_COLUMNAS);
190 }
191
192 void tablero_destruir(tablero_t* tablero){
193     for (int i = 0; i < NRO_FILAS/3; i++){
194         for (int j = 0; j < NRO_COLUMNAS/3; j++){

```

sep 24, 19 9:58

tablero.c

Page 4/4

```

197     sector_destruir(&tablero->sectores[i][j]);
198     }
199     free(tablero->sectores[i]);
200 }
201 free(tablero->sectores);
202 for (int i = 0; i < NRO_FILAS; i++){
203     free(tablero->celdas[i]);
204 }
205 free(tablero->celdas);
206 }

```

sep 24, 19 9:58

sudoku.c

Page 1/1

```

1  #include "servidor.h"
2  #include "cliente.h"
3  #include <stdio.h>
4
5  #define MSJ_ERROR_ARG "Modo no soportado, el primer parámetro debe ser "\
6      "server o client\n"
7  int main(int argc, char **argv){
8      if (argc == 4){
9          cliente_t cliente;
10         if (cliente_crear(&cliente, argv[2], argv[3]) != 0){
11             return 1;
12         }
13         cliente_recibir_comandos(&cliente);
14         cliente_destruir(&cliente);
15     }else if (argc == 3){
16         servidor_t servidor;
17         if (servidor_crear(&servidor, "board.txt") != 0){
18             return 1;
19         }
20         servidor_escuchar(&servidor, argv[2]);
21     }else{
22         printf("%s\n", MSJ_ERROR_ARG);
23     }
24     return 0;
25 }

```

sep 24, 19 9:58

## socket\_tcp.h

Page 1/1

```

1  #ifndef _SOCKET_TCP_H_
2  #define _SOCKET_TCP_H_
3  #include <stdlib.h>
4
5  #define COLA_CONECCIONES 20
6  typedef struct socket_tcp{
7      int fd;
8  }socket_tcp_t;
9
10 //Crea un socket IPV4 del tipo TCP.
11 //POST: crea el socket, en caso de error imprime un mensaje y deja el
12 //socket recibido inalterado.
13 void socket_tcp_crear(socket_tcp_t* self);
14
15 //Envia tantos bytes como indique la longitud, de la informacion almacenada
16 //en el buffer. Lo envia el socket conectado en <socket_tcp_aceptar>
17 //PRE: el buffer debe contener tanto espacio como longitud del mensaje que se
18 //quiere mandar.
19 //POST: Devuelve la cantidad de bytes que fueron enviados o un numero negativo
20 //en caso de error.
21 int socket_tcp_enviar(socket_tcp_t* self, const void* buffer, size_t longitud);
22
23 //Recibe informacion hasta recibir <longitud> bytes. Esta debe ser
24 //menor o igual a la capacidad del buffer.
25 //PRE: el socket debe estar creado y debe ser resultado de una conexion
26 //aceptada y el buffer y la longitud deben ser suficientes para el mensaje
27 //que se quiera recibir.
28 //POST: devuelve la cantidad de bytes recibidos.
29 int socket_tcp_recibir(socket_tcp_t* self, void* buffer, size_t longitud);
30
31 //Conecta el socket a una direccion con el host y servicio especificado.
32 //PRE: el socket fue creado.
33 //POST: se conecta al host indicado y devuelve 0. En caso de error:
34 // * 1: Error en la obtencion de direcciones
35 // * 2: No se pudo conectar.
36 int socket_tcp_conectar(socket_tcp_t* self, const char* host, \
37     const char* servicio);
38
39 //Crea un socket pasivo que escuche conexiones en el servicio/puerto
40 //indicado.
41 //PRE: el socket fue creado. Y el servicio debe ser válido para los permisos
42 //dados al programa.
43 //POST: se crea un socket escuchando a la direccion con una cola de conexiones
44 //de valor COLA_CONECCIONES.
45 int socket_tcp_bind_and_listen(socket_tcp_t* self, const char* servicio);
46
47 //Acepta una conceccion entrante en el socket recibido. Y guarda en un
48 //socket nuevo el socket al cual se acepto
49 //PRE: este debe haber sido puesto a escuchar una direccion con bind_and_listen
50 //el socket cliente no es necesario que este inicializado, dado que se
51 //inicializara de forma correcta en caso de que se acepte correctamente
52 //la conceccion.
53 //POST: devuelve 0 si se aceptó correctamente. -1 en caso de error
54 int socket_tcp_aceptar(socket_tcp_t* self, socket_tcp_t* socket_cliente);
55
56 //Cierra el socket liberando los recursos
57 //Devuelve 1 en caso de error, 0 si no lo hubo.
58 int socket_tcp_destruir(socket_tcp_t* self);
59
60 #endif

```

sep 24, 19 9:58

## socket\_tcp.c

Page 1/3

```

1  #define _POSIX_C_SOURCE 200112L
2  #include "socket_tcp.h"
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netdb.h>
6  #include <unistd.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <errno.h>
10
11
12
13
14 void socket_tcp_crear(socket_tcp_t* self){
15     int fd = socket(AF_INET, SOCK_STREAM, 0);
16     if (fd == -1){
17         return;
18     }
19     self->fd = fd;
20 }
21
22 void setear_addrinfo_tcp_ipv4(struct addrinfo* hints){
23     memset(hints, 0, sizeof(struct addrinfo));
24     hints->ai_family = AF_INET;
25     hints->ai_socktype = SOCK_STREAM;
26 }
27
28 int socket_tcp_conectar(socket_tcp_t* self, const char* host, \
29     const char* servicio){
30     struct addrinfo hints;
31     struct addrinfo* resultados, *dir_act;
32
33     setear_addrinfo_tcp_ipv4(&hints);
34     hints.ai_flags = 0;
35
36     int estado = getaddrinfo(host, servicio, &hints, &resultados);
37     if (estado != 0){
38         freeaddrinfo(resultados);
39         return 1;
40     }
41
42     for (dir_act = resultados; dir_act != NULL; dir_act = dir_act->ai_next){
43         if (connect(self->fd, dir_act->ai_addr, dir_act->ai_addrlen) != -1){
44             break;
45         }
46     }
47     if (!dir_act){
48         freeaddrinfo(resultados);
49         return 2;
50     }
51     freeaddrinfo(resultados);
52     return 0;
53 }
54
55
56 int socket_tcp_bind_and_listen(socket_tcp_t* self, const char* servicio){
57     struct addrinfo hints;
58     struct addrinfo* resultados, *dir_act;
59     setear_addrinfo_tcp_ipv4(&hints);
60     hints.ai_flags = AI_PASSIVE;
61
62     int variable = 1;
63     int salida = setsockopt(self->fd, SOL_SOCKET, SO_REUSEADDR, \
64         &variable, sizeof(variable));
65     if (salida < 0) {
66         return 4;

```

sep 24, 19 9:58

socket\_tcp.c

Page 2/3

```

67     }
68
69     int estado = getaddrinfo(NULL, servicio, &hints, &resultados);
70     if (estado != 0){
71         freeaddrinfo(resultados);
72         return 1;
73     }
74     //Podria ser una funcion auxiliar con un puntero a funcion
75     for (dir_act = resultados; dir_act != NULL; dir_act = dir_act->ai_next){
76         if (bind(self->fd, dir_act->ai_addr, dir_act->ai_addrlen) != -1){
77             break;
78         }
79     }
80     freeaddrinfo(resultados);
81     if (!dir_act){
82         return 2;
83     }
84     if (listen(self->fd, COLA_CONECCIONES) != 0){
85         return 3;
86     }
87
88     return 0;
89 }
90
91 int socket_tcp_recibir(socket_tcp_t* self, void* buffer, size_t longitud){
92     ssize_t recibidos = 0;
93     do{
94         char* buff_nuevo = (char*)buffer;
95         ssize_t nuevos = recv(self->fd, &buff_nuevo[recibidos], \
96             longitud-recibidos, 0);
97         if (nuevos < 0){
98             return -1;
99         } else if (nuevos == 0) {
100             break;
101         }
102         recibidos += nuevos;
103     }while (recibidos < longitud);
104     return recibidos;
105 }
106
107 int socket_tcp_enviar(socket_tcp_t* self, const void* buffer, size_t longitud){
108     ssize_t enviados = 0;
109     do{
110         char* buff_nuevo = (char*) buffer;
111         ssize_t nuevos = send(self->fd, &buff_nuevo[enviados], \
112             longitud-enviados, MSG_NOSIGNAL);
113         if (nuevos < 0){
114             return -1;
115         }
116         enviados += nuevos;
117     }while (enviados < longitud);
118     return enviados;
119 }
120
121 int socket_tcp_aceptar(socket_tcp_t* self, socket_tcp_t* socket_cliente){
122     int fd_cliente = accept(self->fd, NULL, NULL);
123     if (fd_cliente < 0){
124         return -1;
125     }
126     socket_cliente->fd = fd_cliente;
127     return 0;
128 }
129
130 int socket_tcp_destruir(socket_tcp_t* self){
131     shutdown(self->fd, 2);
132     if (close(self->fd) == -1){

```

sep 24, 19 9:58

socket\_tcp.c

Page 3/3

```

133         return 1;
134     }
135     return 0;
136 }

```

sep 24, 19 9:58

servidor.h

Page 1/1

```

1  #ifndef _SERVIDOR_H_
2  #define _SERVIDOR_H_
3  #include "tablero.h"
4  #include "socket_tcp.h"
5
6
7
8  typedef struct servidor{
9      tablero_t tablero;
10     socket_tcp_t socket_tcp;
11     socket_tcp_t socket_cliente;
12     bool esta_conectado;
13 }servidor_t;
14
15 //Crea e inicia un servidor de sudoku con el tablero inicial leido de
16 //el archivo localizado en <ruta_archivo>.
17 //PRE: debe existir un archivo en <ruta_archivo>, y debe tener la configuracion
18 //válida para crear un sudoku.
19 //POST: crea el servidor. Devuelve distinto de 0 en caso de error.
20 int servidor_crear(servidor_t* servidor, const char* ruta_archivo);
21
22 //Deja el servidor en modo escucha para que espere los comandos del cliente.
23 //Utiliza el servicio /puerto indicado en el parametro.
24 int servidor_escuchar(servidor_t* servidor, const char* servicio);
25
26 //Destruye el servidor liberando sus recursos.
27 void servidor_destruir(servidor_t* servidor);
28
29 #endif

```

sep 24, 19 9:58

servidor.c

Page 1/2

```

1  #include "servidor.h"
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <arpa/inet.h>
6
7  #define LONG_MSJ_LONGITUD 4
8  #define MSJ_OK "OK\n"
9  #define LONG_MSJ_OK strlen(MSJ_OK)
10 #define MSJ_ERROR "ERROR\n"
11 #define LONG_MSJ_ERROR strlen(MSJ_ERROR)
12 #define MSJ_CELDA_NO_MODIFICABLE "La celda indicada no es modificable\n"
13 #define LONG_MSJ_CELDA_NO_MOD strlen(MSJ_CELDA_NO_MODIFICABLE)
14
15 int servidor_crear(servidor_t* servidor, const char* ruta_archivo){
16     tablero_t tablero;
17     if (tablero_crear(&tablero, ruta_archivo) != 0){
18         fprintf(stderr, "Error al crear tablero\n");
19         return 1;
20     }
21     servidor->tablero = tablero;
22     servidor->esta_conectado = false;
23     return 0;
24 }
25
26 void enviar_al_cliente(servidor_t* servidor, const char* mensaje, size_t len){
27     unsigned int long_int = htonl(len);
28     socket_tcp_enviar(&servidor->socket_cliente, &long_int, LONG_MSJ_LONGITUD);
29     socket_tcp_enviar(&servidor->socket_cliente, mensaje, len);
30 }
31
32 void enviar_tablero(servidor_t* servidor){
33     char* representacion = tablero_obtener(&servidor->tablero);
34     enviar_al_cliente(servidor, representacion, strlen(representacion));
35     free(representacion);
36 }
37
38 //Recibe los comandos del cliente segun el protocolo. Si el socket
39 //del cliente se cierra devuelve -1 para indicar que la conexion fue terminada
40 int recibir_comandos(servidor_t* servidor, char* comando){
41     int recibidos;
42     memset(comando, '\0', 4);
43     recibidos = socket_tcp_recibir(&servidor->socket_cliente, comando, 1);
44     if (recibidos < 0){
45         return 1;
46     }else if (recibidos == 0){
47         return -1;
48     }
49     if (comando[0] == 'P'){
50         recibidos = socket_tcp_recibir(&servidor->socket_cliente,
51                                     &comando[1], 3);
52         if (recibidos < 0){ return 1; }
53     }
54     return 0;
55 }
56
57 void verificar(servidor_t* servidor){
58     if (tablero_verificar(&servidor->tablero)){
59         enviar_al_cliente(servidor, MSJ_OK, LONG_MSJ_OK);
60     }else{
61         enviar_al_cliente(servidor, MSJ_ERROR, LONG_MSJ_ERROR);
62     }
63 }
64
65 void poner(servidor_t* servidor, int fila, int col, int valor){
66     //Si la celda no es modificable

```

sep 24, 19 9:58

servidor.c

Page 2/2

```

67  if (tablero_modificar_celda(&servidor->tablero, fila, col, valor) == 2){
68      enviar_al_cliente(servidor, MSJ_CELDA_NO_MODIFICABLE, \
69          LONG_MSJ_CELDA_NO_MOD);
70  }else{
71      enviar_tablero(servidor);
72  }
73  }
74
75  void ejecutar_comando(servidor_t* servidor, const char* comando){
76      if (comando[0] == 'V'){
77          verificar(servidor);
78      }else if (comando[0] == 'R'){
79          tablero_reiniciar(&servidor->tablero);
80          enviar_tablero(servidor);
81      }else if (comando[0] == 'G'){
82          enviar_tablero(servidor);
83      }else if (comando[0] == 'P'){
84          poner(servidor, comando[1] -1, comando[2] -1, comando[3]);
85      }
86  }
87  int servidor_escuchar(servidor_t* servidor, const char* servicio){
88      socket_tcp_crear(&servidor->socket_tcp);
89      if (socket_tcp_bind_and_listen(&servidor->socket_tcp, servicio) != 0){
90          fprintf(stderr, "Error al poner el socket en modo escucha\n");
91          return 1;
92      }
93      servidor->esta_conectado = true;
94      if (socket_tcp_aceptar(&servidor->socket_tcp, \
95          &servidor->socket_cliente) < 0){
96          fprintf(stderr, "Error en aceptar\n");
97          return 2;
98      }
99      char comando[4];
100      while (1) {
101          if (recibir_comandos(servidor, comando) < 0){
102              break;
103          }
104          ejecutar_comando(servidor, comando);
105      }
106      servidor_destruir(servidor);
107      return 0;
108  }
109
110  void servidor_destruir(servidor_t* servidor){
111      tablero_destruir(&servidor->tablero);
112      if (servidor->esta_conectado){
113          socket_tcp_destruir(&servidor->socket_tcp);
114      }
115      return;
116  }

```

sep 24, 19 9:58

sector.h

Page 1/1

```

1  #ifndef _SECTOR_H_
2  #define _SECTOR_H_
3  #include "celda.h"
4
5  #define CAPACIDAD 9
6
7  typedef struct{
8      celda_t** celdas;
9      int cant_guardados;
10 }sector_t;
11
12 //Crea un sector valido con la CAPACIDAD establecida. Devuelve 0 si se creo
13 // correctamente 1 en caso contrario
14 int sector_crear(sector_t* sector);
15
16 //Agrega una celda al sector recibido. NO verifica si la celda ya fue
17 //agregada.
18 //Devuelve distinto de 0 si hubo un error, por ejemplo si el sector esta lleno.
19 int sector_agregar_celda(sector_t* sector, celda_t* celda);
20
21 //Cuenta cuantas celdas tienen ese numero almacenado y lo devuelve
22 int sector_contar_apariciones(sector_t* sector, int numero);
23
24 //Destruye el sector, liberando sus recursos. No modifica ni elimina
25 //las celdas a las cuales tiene referencia.
26 void sector_destruir(sector_t* sector);
27
28
29 #endif

```

sep 24, 19 9:58

sector.c

Page 1/1

```

1  #include "sector.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  #define CAPACIDAD 9
6  int sector_crear(sector_t* sector){
7      celda_t** celdas = malloc(sizeof(celda_t)*CAPACIDAD);
8      if (sector == NULL){
9          return 1;
10     }
11     sector->celdas = celdas;
12     sector->cant_guardados = 0;
13     return 0;
14 }
15
16 int sector_agregar_celda(sector_t* sector, celda_t* celda){
17     if (sector->cant_guardados == CAPACIDAD){
18         return 1;
19     }
20     sector->celdas[sector->cant_guardados] = celda;
21     sector->cant_guardados++;
22     return 0;
23 }
24
25 int sector_contar_apariciones(sector_t* sector, int numero){
26     int contador = 0;
27     for (int i = 0; i < sector->cant_guardados; i++){
28         if (celda_obtener_numero(sector->celdas[i]) == numero){
29             contador++;
30         }
31     }
32     return contador;
33 }
34
35 void sector_destruir(sector_t* sector){
36     free(sector->celdas);
37 }

```

sep 24, 19 9:58

impresor.h

Page 1/1

```

1  #ifndef _IMPRESOR_H_
2  #define _IMPRESOR_H_
3  #include "celda.h"
4
5  //Recibe una lista de listas de celdas, el numero de filas y
6  //de columnas que tiene el archivo y devuelve una representacion
7  //en forma de cadena del mismo
8  char* impresor_imprimir_celdas(const celda_t** celdas, \
9      int nro_filas, int nro_col);
10
11 #endif

```



sep 24, 19 9:58

impresor.c

Page 1/1

```

1  #include "impresor.h"
2  #include <string.h>
3  #include <stdlib.h>
4
5  #define HEADER "U=====U=====U=====U\n"
6  #define LONG_HEADER strlen(HEADER)+1
7  #define SEP_MEDIO "U---+---+---U---+---+---U---+---+---U\n"
8  #define LONG_SEP_MEDIO strlen(SEP_MEDIO)
9  #define SEPARADOR_SECTOR 'U'
10 #define SEPARADOR_CELDA '|'
11
12 char procesar_celda(const celda_t* celda){
13     if (!celda_esta_vacia(celda)){
14         return celda_obtener_numero(celda) + '0';
15     }
16     return ' ';
17 }
18
19 void procesar_fila(const celda_t** celdas, int fila,\
20     int nro_col, char* repr_fila){
21     int col;
22     for (col = 0; col <= nro_col*4; col++){
23         if (col % 12 == 0){
24             repr_fila[col] = SEPARADOR_SECTOR;
25         }else if (col % 4 == 0){
26             repr_fila[col] = SEPARADOR_CELDA;
27         }else if (col % 2 == 0){
28             repr_fila[col] = procesar_celda(\
29                 &celdas[(fila - 1) / 2][(col-2) / 4]);
30         }else{
31             repr_fila[col] = ' ';
32         }
33     }
34     repr_fila[col] = '\n';
35 }
36
37 char* impresor_imprimir_celdas(const celda_t** celdas, int nro_filas,\
38     int nro_col){
39     char* repr_tablero = malloc(sizeof(char)*(LONG_HEADER+1)*2*(nro_col+1)+1);
40     strncpy(repr_tablero, HEADER, LONG_HEADER);
41     for (int fila = 1; fila < nro_filas*2; fila++){
42         if (fila % 6 == 0){
43             strncat(repr_tablero, HEADER, LONG_HEADER);
44         }else if (fila % 2 == 0){
45             strncat(repr_tablero, SEP_MEDIO, LONG_SEP_MEDIO);
46         }else{
47             char repr_fila[LONG_HEADER-1];
48             procesar_fila(celdas, fila, nro_col, repr_fila);
49             strncat(repr_tablero, repr_fila, LONG_HEADER-1);
50         }
51     }
52     strncat(repr_tablero, HEADER, LONG_HEADER);
53     return repr_tablero;
54 }

```

sep 24, 19 9:58

cliente.h

Page 1/1

```

1  #ifndef _CLIENTE_H_
2  #define _CLIENTE_H_
3  #include "socket_tcp.h"
4
5  typedef struct cliente{
6     socket_tcp_t socket_tcp;
7 }cliente_t;
8
9  //Crea un cliente y lo conecta al servidor en el puerto y host indicado.
10 //PRE: el servidor debe estar iniciado y escuchando en el puerto y host que
11 //se intenta conectar.
12 //POST: crea y conceta el cliente.
13 int cliente_crear(cliente_t* cliente, const char* host, const char* puerto);
14
15 //Pone al cliente en modo escucha para recibir comandos por STDIN, hasta que
16 //se ingrese el comando de salida: "exit".
17 //PRE: el cliente debe estar creado correctamente. Devuelve 1
18 //POST: verifica y ejecuta los comandos recibidos.
19 void cliente_recibir_comandos(cliente_t* cliente);
20
21 //Destruye el cliente liberando los recursos.
22 void cliente_destruir(cliente_t* cliente);
23
24 #endif

```

sep 24, 19 9:58

cliente.c

Page 1/3

```

1  #define _POSIX_C_SOURCE 200809L
2  #include "cliente.h"
3  #include "socket_tcp.h"
4  #include <stdio.h>
5  #include <ctype.h>
6  #include <string.h>
7  #include <arpa/inet.h>
8
9  #define TAMANIO_BUFFER_COMANDOS 16
10 #define TAMANIO_MSJ_SERVIDOR 4
11 #define MSJ_VERIFICAR "V"
12 #define MSJ_REINICIAR "R"
13 #define MSJ_OBTENER "G"
14 #define MSJ_PONER "P"
15
16 #define MSJ_COMANDO_INVALIDO "Comando invalido, intente nuevamente\n"
17 #define MSJ_INDICE_INVALIDO "Error en los A-ndices. Rango soportado: [1,9]\n"
18 #define MSJ_VALOR_INVALID "Error en el valor ingresado. Rango soportado: [1,9]\n"
19 #define MSJ_INGRESO_COMANDOS "Ingrese un comando: "
20
21
22 int cliente_crear(cliente_t* cliente, const char* host, const char* puerto){
23     socket_tcp_crear(&cliente->socket_tcp);
24     if (socket_tcp_conectar(&cliente->socket_tcp, host, puerto) != 0){
25         fprintf(stderr, "No se pudo crear el cliente, error al conectar\n");
26         socket_tcp_destruir(&cliente->socket_tcp);
27         return 1;
28     }
29     return 0;
30 }
31
32 void enviar_al_servidor(cliente_t* cliente, const char* mensaje){
33     socket_tcp_enviar(&cliente->socket_tcp, mensaje, strlen(mensaje));
34 }
35
36 char* recibir_mensaje_servidor(cliente_t* cliente){
37     unsigned int tamano;
38     socket_tcp_recibir(&cliente->socket_tcp, &tamano, TAMANIO_MSJ_SERVIDOR);
39     tamano = ntohs(tamano);
40     char* mensaje = malloc(sizeof(char)*tamano + 1);
41     if (mensaje == NULL){
42         return NULL;
43     }
44     socket_tcp_recibir(&cliente->socket_tcp, mensaje, tamano);
45     mensaje[tamano] = '\0';
46     return mensaje;
47 }
48
49
50
51 void realizar_comando(cliente_t* cliente, const char* mensaje_enviar){
52     enviar_al_servidor(cliente, mensaje_enviar);
53     char* mensaje_recibido = recibir_mensaje_servidor(cliente);
54     if (mensaje_recibido == NULL) return;
55     printf("%s", mensaje_recibido);
56     free(mensaje_recibido);
57 }
58
59 int esta_en_rango(int numero){
60     return (numero > 0) ^ (numero <= 9);
61 }
62
63 int verificar_valores(int valor, int fila, int columna){
64     if (!esta_en_rango(fila) || !esta_en_rango(columna)){
65         fprintf(stderr, "%s", MSJ_INDICE_INVALIDO);
66         return 1;

```

sep 24, 19 9:58

cliente.c

Page 2/3

```

67     }else if (!esta_en_rango(valor)){
68         fprintf(stderr, "%s", MSJ_VALOR_INVALID);
69         return 1;
70     }
71     return 0;
72 }
73
74 char* crear_comando_poner(int valor, int fila, int columna){
75     char* comando = malloc(sizeof(char) * 5);
76     if (comando == NULL){
77         return NULL;
78     }
79     comando[0] = 'P';
80     comando[1] = (unsigned char) fila;
81     comando[2] = (unsigned char) columna;
82     comando[3] = (unsigned char) valor;
83     comando[4] = '\0';
84     return comando;
85 }
86
87
88 char* parsear_comando(char* comando){
89     char* parametros[4];
90     char* comando_poner = NULL;
91     for (int i = 0; i < 4; i++){
92         char* param = strtok_r(NULL, " ", &comando);
93         parametros[i] = param;
94     }
95     int valor = atoi(parametros[0]);
96     int fila = atoi(parametros[2]);
97     int columna = atoi(parametros[3]);
98     if (verificar_valores(valor, fila, columna) == 0){
99         comando_poner = crear_comando_poner(valor, fila, columna);
100     }
101     return comando_poner;
102 }
103 void cliente_recibir_comandos(cliente_t* cliente){
104     char* comando = NULL;
105     char* guardado;
106     size_t tamano;
107     while (getline(&comando, &tamano, stdin) > 0){
108         if (strlen(comando) > TAMANIO_BUFFER_COMANDOS){
109             fprintf(stderr, "%s\n", MSJ_COMANDO_INVALIDO);
110             continue;
111         }
112         comando[0] = toupper(comando[0]);
113         char * primera_palabra = strtok_r(comando, " ", &guardado);
114         if (strcmp(primera_palabra, "Verify\n") == 0){
115             realizar_comando(cliente, MSJ_VERIFICAR);
116         }else if (strcmp(primera_palabra, "Get\n") == 0){
117             realizar_comando(cliente, MSJ_OBTENER);
118         }else if (strcmp(primera_palabra, "Put\n") == 0){
119             char* comando_poner = parsear_comando(guardado);
120             if (comando_poner == NULL) continue;
121             realizar_comando(cliente, comando_poner);
122             free(comando_poner);
123         }else if (strcmp(primera_palabra, "Reset\n") == 0){
124             realizar_comando(cliente, MSJ_REINICIAR);
125         }else if (strcmp(primera_palabra, "Exit\n") == 0){
126             break;
127         }else{
128             fprintf(stderr, "%s\n", MSJ_COMANDO_INVALIDO);
129         }
130     }
131     free(comando);
132 }

```

sep 24, 19 9:58

cliente.c

Page 3/3

```
133
134 void cliente_destruir(cliente_t* cliente){
135     socket_tcp_destruir(&cliente->socket_tcp);
136 }
```

sep 24, 19 9:58

celda.h

Page 1/1

```
1  #ifndef _CELDA_H_
2  #define _CELDA_H_
3
4  #include <stdbool.h>
5
6
7  typedef struct{
8      int numero;
9      bool modificable;
10     bool validez;
11 }celda_t;
12
13 //Inicia una celda con el numero indicado por parametro.
14 //PRE: debe estar creada la celda_t. El numero debe ser del 0 al 9
15 //POST: inicializa la celda con el numero indicado y configurando
16 // si es modificable o no.
17 int celda_crear(celda_t* celda, int numero, bool es_modificable);
18
19 //Devuelve el numero contenido en la celda recibida.
20 int celda_obtener_numero(const celda_t* celda);
21
22 //Devuelve si la celda recibida es modificable
23 bool celda_es_modificable(celda_t* celda);
24
25 //Devuelve True si la celda esta vacia (tiene un 0), False si no lo esta.
26 bool celda_esta_vacia(const celda_t* celda);
27
28
29 //Intenta actualizar el numero de la celda recibida.
30 //PRE: la celda debe ser modificable y el numero debe ser del 0 al 9
31 //POST: actualiza el numero de la celda. Si este no es valido o la celda
32 //no es modificable imprime un mensaje por STDOUT y deja la celda intacta
33 void celda_poner_numero(celda_t* celda, int numero);
34
35 #endif
```

sep 24, 19 9:58

celda.c

Page 1/1

```

1  #include "celda.h"
2  #include <stdio.h>
3
4
5  bool es_numero_valido(int numero){
6      return (numero ≥ 0) ^ (numero ≤ 9);
7  }
8
9  int celda_crear(celda_t* celda, int numero, bool es_modificable){
10     if ( ¬es_numero_valido(numero) ){
11         return 1;
12     }
13     celda→numero = numero;
14     celda→modificable = es_modificable;
15     return 0;
16 }
17
18
19 int celda_obtener_numero(const celda_t* celda){
20     return celda→numero;
21 }
22
23 bool celda_esta_vacia(const celda_t* celda){
24     return celda→numero ≡ 0;
25 }
26
27 bool celda_es_modificable(celda_t* celda){
28     return celda→modificable;
29 }
30
31 void celda_poner_numero(celda_t* celda, int numero){
32     if ( ¬es_numero_valido(numero) ){
33         return;
34     } else if ( ¬celda→modificable ){
35         return;
36     }
37
38     celda→numero = numero;
39 }

```

sep 24, 19 9:58

Table of Content

Page 1/1

1	<b>Table of Contents</b>				
2	1 <i>tablero.h</i> .....	sheets	1 to	1 ( 1) pages	1- 1 54 lines
3	2 <i>tablero.c</i> .....	sheets	1 to	3 ( 3) pages	2- 5 207 lines
4	3 <i>sudoku.c</i> .....	sheets	3 to	3 ( 1) pages	6- 6 26 lines
5	4 <i>socket_tcp.h</i> .....	sheets	4 to	4 ( 1) pages	7- 7 61 lines
6	5 <i>socket_tcp.c</i> .....	sheets	4 to	5 ( 2) pages	8- 10 137 lines
7	6 <i>servidor.h</i> .....	sheets	6 to	6 ( 1) pages	11- 11 30 lines
8	7 <i>servidor.c</i> .....	sheets	6 to	7 ( 2) pages	12- 13 117 lines
9	8 <i>sector.h</i> .....	sheets	7 to	7 ( 1) pages	14- 14 30 lines
10	9 <i>sector.c</i> .....	sheets	8 to	8 ( 1) pages	15- 15 38 lines
11	10 <i>impresor.h</i> .....	sheets	8 to	8 ( 1) pages	16- 16 12 lines
12	11 <i>impresor.c</i> .....	sheets	9 to	9 ( 1) pages	17- 17 55 lines
13	12 <i>cliente.h</i> .....	sheets	9 to	9 ( 1) pages	18- 18 25 lines
14	13 <i>cliente.c</i> .....	sheets	10 to	11 ( 2) pages	19- 21 137 lines
15	14 <i>celda.h</i> .....	sheets	11 to	11 ( 1) pages	22- 22 36 lines
16	15 <i>celda.c</i> .....	sheets	12 to	12 ( 1) pages	23- 23 40 lines