sep 10, 19 15:51 **tablero.h** Page 1/1

```c
#ifndef _TABLERO_H_
#define _TABLERO_H_
#include "celda.h"
#include "sector.h"


#define NRO_FILAS 9
#define NRO_COLUMNAS 9
typedef struct {
  celda_t** celdas;
  sector_t** sectores;
}tablero_t;

//Crea un tablero de 9 x 9 casilleros a partir de un archivo de texto
//que contiene los valores iniciales de las celdas. Estas celdas se
//identificaran como prefijadas y por lo tanto inmodificables.
//PRE: el arhivo debe tener el formato de una fila por linea, con
//los numeros para cada celda separados con espacios y se entiende
//como una celda con 0 como una celda vacia.
//POST: crea un tablero con los valores del archivo. Devuelve distinto
//de 0 si ocurre algun error
int tablero_crear(tablero_t* tablero, const char* ruta_archivo);

//Verifica que todas las celdas del tablero
//tengan valores que cumplan con las reglas:
// *En cada fila, columna y sector hay solamente
//  numeros del 1 al 9 distintos o celdas vacias
//POST: Devuelve true si las reglas se cumplen para
//todas las celdas falso en caso contrario.
bool tablero_verificar(tablero_t* tablero);

//Devuelve una representacion del tablero como cadena
char* tablero_obtener(tablero_t* tablero);

//Modifica la celda en la posicion (fila, columna) del tablero poniendole
//el valor recibido.
//PRE: el tablero fue creado. Fila y columna deben ser dos valores validos
//POST: devuelve 0 si la celda se modifico correctamente, en caso de error:
// * 1 si (fila, columna) es una posicion invalida
// * 2 si la celda no es modificable
int tablero_modificar_celda(tablero_t* tablero, int fila, int col, int valor);

//Reinicia el tablero recibido a los valores prefijados
//en la creacion.
//POST: Reinicia el tablero eliminando los valores de las
//celdas no fijadas en la creacion Â¿y lo devuelve?
void tablero_reiniciar(tablero_t* tablero);

//Destruye el tablero recibido por parametro
//liberando todos los recursos utilizados
void tablero_destruir(tablero_t* tablero);

#endif
```

sep 10, 19 15:51 **tablero.c** Page 1/4

```c
#include "tablero.h"
#include "impresor.h"
#include <stdio.h>
#include <stdlib.h>

#define TAMANIO_BUFFER_LECTURA 2 * NRO_COLUMNAS

int digito_a_int(const char caracter){
  return caracter - '0';
}

//Toma los caracteres en las posiciones pares, es decir, evita
//los separadores y convierte los que estes en estas posiciones
//a entero para crear una celda con ese valor. Si el valor es 0
//se entiende que la celda esta vacia. En caso contrario se entiende
//que es una de las celdas de pista y por lo tanto, no modificable.
celda_t* procesar_fila_leida(char* linea, size_t leidos){
  celda_t* fila = malloc(sizeof(celda_t) * NRO_COLUMNAS);
  if (fila == NULL){
    return NULL;
  }
  for (int i = 0; i < leidos; i++){
    if (i % 2 == 0){
      int valor = digito_a_int(linea[i]);
      if (valor == 0){
        celda_crear(&fila[i/2], valor, true);
      }else{
        celda_crear(&fila[i/2], valor, false);
      }
    }
  }
  return fila;
}

int leer_archivo(const char* ruta_archivo, celda_t** filas){
  FILE* archivo = fopen(ruta_archivo, "r");
  if (archivo == NULL){
    //fprintf(stderr,"No se pudo abrir el archivo\n");
    return 1;
  }
  int fila_actual = 0;
  size_t leidos;
  char buffer[TAMANIO_BUFFER_LECTURA];
  while ((leidos = fread(&buffer, 1, TAMANIO_BUFFER_LECTURA, archivo))) {
    if (leidos < TAMANIO_BUFFER_LECTURA-1) { //si la fila no esta completa
      //fprintf(stderr,"El archivo de entrada tiene filas incompletas\n");
      free(filas);
      fclose(archivo);
      return 2;
    }
    celda_t* fila = procesar_fila_leida(buffer, leidos);
    if (fila == NULL){
      //fprintf(stderr,"No hay memoria suficiente\n");
      free(filas);
      fclose(archivo);
      return 3;
    }
    filas[fila_actual] = fila;
    fila_actual++;
  }
  fclose(archivo);
  return 0;
}

int agregar_sectores(tablero_t* tablero){
  sector_t** sectores = malloc(sizeof(sector_t)* NRO_FILAS/3);
```

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



```
132     }
133   }
134   return apariciones;
135 }
136 //Cuenta las apariciones del numero recibido en la columna indicada
137 int contar_apariciones_fila(tablero_t* tablero, int numero, int fila){
138   int apariciones = 0;
139   for (int i = 0; i < NRO_COLUMNAS; i++){
140     if (celda_obtener_numero(&tablero→celdas[fila][i]) ≡ numero){
141       apariciones++;
142     }
143   }
144   return apariciones;
145 }
146
147
148 bool verificar_celda(tablero_t* tablero, int fila, int columna){
149   int numero = celda_obtener_numero(&tablero→celdas[fila][columna]);
150   if (numero ≡ 0){ //esta vacia
151     return true;
152   }
153   if(contar_apariciones_columna(tablero, numero, columna)>1){
154     return false;
155   }
156   if(contar_apariciones_fila(tablero, numero, fila)>1){
157     return false;
158   }
159   if(sector_contar_apariciones(&tablero→sectores[fila/3][columna/3], numero)>1)
    {
160     return false;
161   }
162
163   return true;
164 }
165
166 bool tablero_verificar(tablero_t* tablero){
167   for (int i = 0; i < NRO_FILAS; i++){
168     for (int j = 0; j < NRO_COLUMNAS; j++){
169       if (¬verificar_celda(tablero, i,j)){
170         return false;
171       }
172     }
173   }
174   return true;
175 }
176
177
178 int tablero_modificar_celda(tablero_t* tablero, int fil, int col, int valor){
179   if ((fil < 0) ∨ (fil ≥ NRO_FILAS) ∨ (col < 0) ∨ (col ≥ NRO_COLUMNAS)){
180     return 1;
181   }
182   celda_t* celda = &tablero→celdas[fil][col];
183
184   if (¬celda_es_modificable(celda)){
185     return 2;
186   }
187   celda_poner_numero(celda, valor);
188   return 0;
189 }
190
191 char* tablero_obtener(tablero_t* tablero){
192   return impresor_imprimir_celdas((const celda_t **)tablero→celdas,\
193   NRO_FILAS, NRO_COLUMNAS);
194 }
195
196 void tablero_destruir(tablero_t* tablero){
```

sep 10, 19 15:51 **tablero.c** Page 4/4

```
197   for (int i = 0; i < NRO_FILAS/3; i++){
198     for (int j = 0; j < NRO_COLUMNAS/3; j++){
199       sector_destruir(&tablero→sectores[i][j]);
200     }
201     free(tablero→sectores[i]);
202   }
203   free(tablero→sectores);
204   for (int i = 0; i < NRO_FILAS; i++){
205     free(tablero→celdas[i]);
206   }
207   free(tablero→celdas);
208 }
```

sep 10, 19 15:51 **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 parÃ¡metro 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   //fclose(stdin);
25   //fclose(stdout);
26   //fclose(stderr);
27   return 0;
28 }
```

sep 10, 19 15:51 **socket_tcp.h** Page 1/1

```c
#ifndef _SOCKET_TCP_H_
#define _SOCKET_TCP_H_
#include <stdlib.h>

#define COLA_CONECCIONES 20
typedef struct socket_tcp{
  int fd;
}socket_tcp_t;

//Crea un socket IPV4 del tipo TCP.
//POST: crea el socket, en caso de error imprime un mensaje y deja el
//socket recibido inalterado.
void socket_tcp_crear(socket_tcp_t* self);

//Envia tantos bytes como indique la longitud, de la informacion almacenada
//en el buffer. Lo envia el socket conectado en <socket_tcp_aceptar>
//PRE: el buffer debe contener tanto espacio como longitud del mensaje que se
//quiere mandar.
//POST: Devuelve la cantidad de bytes que fueron enviados o un numero negativo
//en caso de error.
int socket_tcp_enviar(socket_tcp_t* self, const void* buffer, size_t longitud);

//Recibe informacion hasta recibir <longitud> bytes. Esta debe ser
//menor o igual a la capacidad del buffer.
//PRE: el socket debe estar creado y debe ser resultado de una conexion
//aceptada y el buffer y la longitud deben ser suficientes para el mensaje
//que se quiera recibir.
//POST: devuelve la cantidad de bytes recibidos.
int socket_tcp_recibir(socket_tcp_t* self, void* buffer, size_t longitud);

//Conecta el socket a una direccion con el host y servicio especificado.
//PRE: el socket fue creado.
//POST: se conecta al host indicado y devuelve 0. En caso de error:
// * 1: Error en la obtencion de direcciones
// * 2: No se pudo conectar.
int socket_tcp_conectar(socket_tcp_t* self, const char* host, \
  const char* servicio);

//Crea un socket pasivo que escuche conecciones en el servicio/puerto
//indicado.
//PRE: el socket fue creado. Y el servicio debe ser vÃ¡lido para los permisos
//dados al programa.
//POST: se crea un socket escuchando a la direccion con una cola de conecciones
//de valor COLA_CONECCIONES.
int socket_tcp_bind_and_listen(socket_tcp_t* self, const char* servicio);

//Acepta una conceccion entrante en el socket recibido. Y guarda en un
//socket nuevo el socket al cual se acepto
//PRE: este debe haber sido puesto a escuchar una direccion con bind_and_listen
//el socket cliente no es necesario que este inicializado, dado que se
//inicializara de forma correcta en caso de que se acepte correctamente
//la coneccion.
//POST: devuelve 0 si se aceptÃ³ correctamente. -1 en caso de error
int socket_tcp_aceptar(socket_tcp_t* self, socket_tcp_t* socket_cliente);

//Cierra el socket liberando los recursos
//Devuelve 1 en caso de error, 0 si no lo hubo.
int socket_tcp_destruir(socket_tcp_t* self);

#endif
```

sep 10, 19 15:51 **socket_tcp.c** Page 1/3

```c
#define _POSIX_C_SOURCE 200112L
#include "socket_tcp.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>




void socket_tcp_crear(socket_tcp_t* self){
  int fd = socket(AF_INET, SOCK_STREAM, 0);
  if (fd ≡ -1){
    //printf("No se pudo crear el socket\n");
    return;
  }
  self→fd = fd;
}

void setear_addrinfo_tcp_ipv4(struct addrinfo* hints){
  memset(hints, 0, sizeof(struct addrinfo));
  hints→ai_family = AF_INET;
  hints→ai_socktype = SOCK_STREAM;
}

int socket_tcp_conectar(socket_tcp_t* self, const char* host,\
  const char* servicio){
  struct addrinfo hints;
  struct addrinfo* resultados, *dir_act;

  setear_addrinfo_tcp_ipv4(&hints);
  hints.ai_flags = 0;

  int estado = getaddrinfo(host, servicio, &hints, &resultados);
  if (estado ≠ 0){
    //printf("Error en la obtencion de direcciones\n");
    freeaddrinfo(resultados);
    return 1;
  }

  for (dir_act = resultados; dir_act ≠ NULL; dir_act = dir_act→ai_next){
    if (connect(self→fd, dir_act→ai_addr, dir_act→ai_addrlen) ≠ -1){
      break;
    }
  }
  if (¬dir_act){
    //printf("No se pudo conectar\n");
    freeaddrinfo(resultados);
    return 2;
  }
  freeaddrinfo(resultados);
  return 0;
}


int socket_tcp_bind_and_listen(socket_tcp_t* self, const char* servicio){
  struct addrinfo hints;
  struct addrinfo* resultados, *dir_act;
  setear_addrinfo_tcp_ipv4(&hints);
  hints.ai_flags = AI_PASSIVE;

  int variable = 1;
    int salida = setsockopt(self→fd, SOL_SOCKET, SO_REUSEADDR, \
```

```
67        &variable, sizeof(variable));
68      if (salida < 0) {
69        //printf("Error al aplicar configuraciones: %s\n", strerror(errno));
70        return 4;
71      }
72
73    int estado = getaddrinfo(NULL, servicio, &hints, &resultados);
74    if (estado ≠ 0){
75      //printf("Error en la obtencion de direcciones\n");
76      freeaddrinfo(resultados);
77      return 1;
78    }
79    //Podria ser una funcion auxiliar con un puntero a funcion
80    for (dir_act = resultados; dir_act ≠ NULL; dir_act = dir_act→ai_next){
81      if (bind(self→fd, dir_act→ai_addr, dir_act→ai_addrlen) ≠ -1){
82        break;
83      }
84    }
85    freeaddrinfo(resultados);
86    if (¬dir_act){
87      //printf("No se pudo conectar. Error: %s\n", strerror(errno));
88      return 2;
89    }
90    if (listen(self→fd, COLA_CONECCIONES) ≠ 0){
91      //printf("Error en el listen\n");
92      return 3;
93    }
94
95    return 0;
96  }
97
98  int socket_tcp_recibir(socket_tcp_t* self, void* buffer, size_t longitud){
99    ssize_t recibidos = 0;
100   do{
101     char* buff_nuevo = (char*)buffer;
102     ssize_t nuevos = recv(self→fd, &buff_nuevo[recibidos], \
103       longitud-recibidos, 0);
104     if (nuevos < 0){
105       //printf("Error en la recepcion del mensaje: %s\n", strerror(errno));
106       return -1;
107     } else if (nuevos ≡ 0) {
108       //printf("Error: se cerro el socket del cual se recibian datos\n");
109       break;
110     }
111     recibidos += nuevos;
112   }while (recibidos < longitud);
113   return recibidos;
114 }
115
116 int socket_tcp_enviar(socket_tcp_t* self, const void* buffer, size_t longitud){
117   ssize_t enviados = 0;
118   do{
119     ssize_t nuevos = send(self→fd, buffer, longitud, MSG_NOSIGNAL);
120     if (nuevos < 0){
121       //printf("Error en el envio del mensaje: %s\n", strerror(errno));
122       return -1;
123     }
124     enviados += nuevos;
125   }while (enviados < longitud);
126   return enviados;
127 }
128
129 int socket_tcp_aceptar(socket_tcp_t* self, socket_tcp_t* socket_cliente){
130   int fd_cliente = accept(self→fd, NULL, NULL);
131   if (fd_cliente < 0){
132     //printf("Error al aceptar una coneccion: %s\n", strerror(errno));
```



```
133     return -1;
134   }
135   socket_cliente→fd = fd_cliente;
136   return 0;
137 }
138
139 int socket_tcp_destruir(socket_tcp_t* self){
140   shutdown(self→fd, 2);
141   if (close(self→fd) ≡ -1){
142     //printf("No se pudo cerrar el socket\n");
143     return 1;
144   }
145   return 0;
146 }
```

sep 10, 19 15:51 **servidor.h** Page 1/1

```c
#ifndef _SERVIDOR_H_
#define _SERVIDOR_H_
#include "tablero.h"
#include "socket_tcp.h"



typedef struct servidor{
  tablero_t tablero;
  socket_tcp_t socket_tcp;
  socket_tcp_t socket_cliente;
  bool esta_conectado;
}servidor_t;

//Crea e inicia un servidor de sudoku con el tablero inicial leido de
//el archivo localizado en <ruta archivo>.
//PRE: debe existir un archivo en <ruta_archivo>, y debe tener la configuracion
//vÃ¡lida para crear un sudoku.
//POST: crea el servidor. Devuelve distinto de 0 en caso de error.
int servidor_crear(servidor_t* servidor, const char* ruta_archivo);

//Deja el servidor en modo escucha para que espere los comandos del cliente.
//Utiliza el serivicio /puerto indicado en el parametro.
int servidor_escuchar(servidor_t* servidor, const char* servicio);

//Destruye el servidor liberando sus recursos.
void servidor_destruir(servidor_t* servidor);

#endif
```

sep 10, 19 15:51 **servidor.c** Page 1/2

```c
#include "servidor.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define LONG_MSJ_LONGITUD 4
#define MSJ_OK "OK\n"
#define LONG_MSJ_OK strlen(MSJ_OK)+1 //Por el '\0'
#define MSJ_ERROR "ERROR\n"
#define LONG_MSJ_ERROR strlen(MSJ_ERROR)+1
#define MSJ_CELDA_NO_MODIFICABLE "La celda indicada no es modificable\n"
#define LONG_MSJ_CELDA_NO_MOD strlen(MSJ_CELDA_NO_MODIFICABLE)+1

int servidor_crear(servidor_t* servidor, const char* ruta_archivo){
  tablero_t tablero;
  if (tablero_crear(&tablero, ruta_archivo) ≠ 0){
    //fprintf(stderr, "Error al crear tablero\n");
    return 1;
  }
  servidor→tablero = tablero;
  servidor→esta_conectado = false;
  return 0;
}

void enviar_al_cliente(servidor_t* servidor, const char* mensaje, size_t len){
  unsigned int long_int = htonl(len);
  socket_tcp_enviar(&servidor→socket_cliente, &long_int, LONG_MSJ_LONGITUD);
  socket_tcp_enviar(&servidor→socket_cliente, mensaje, len);
}

void enviar_tablero(servidor_t* servidor){
  char* representacion = tablero_obtener(&servidor→tablero);
  enviar_al_cliente(servidor, representacion, strlen(representacion)+1);
  free(representacion);
}

//Recibe los comandos del cliente segun el protocolo. Si el socket
//del cliente se cierra devuelve -1  para indicar que la conexion fue terminada
int recibir_comandos(servidor_t* servidor, char* comando){
  int recibidos;
  memset(comando, '\0', 4);
  recibidos = socket_tcp_recibir(&servidor→socket_cliente, comando, 1);
  if (recibidos < 0){
    return 1;
  }else if (recibidos ≡ 0){
    return -1;
  }
  if (comando[0] ≡ 'P'){
    recibidos = socket_tcp_recibir(&servidor→socket_cliente, &comando[1], 3);
    if (recibidos < 0){ return 1;}
  }
  return 0;
}

void verificar(servidor_t* servidor){
  if (tablero_verificar(&servidor→tablero)){
    enviar_al_cliente(servidor, MSJ_OK, LONG_MSJ_OK);
  }else{
    enviar_al_cliente(servidor, MSJ_ERROR, LONG_MSJ_ERROR);
  }
}

void poner(servidor_t* servidor, int fila, int col, int valor){
  //Si la celda no es modificable
  if (tablero_modificar_celda(&servidor→tablero, fila, col, valor) ≡ 2){
```

sep 10, 19 15:51 **servidor.c** Page 2/2

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

sep 10, 19 15:51 **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 10, 19 15:51 **sector.c** Page 1/1

```c
#include "sector.h"
#include <stdlib.h>
#include <stdio.h>

#define CAPACIDAD 9
int sector_crear(sector_t* sector){
  celda_t** celdas = malloc(sizeof(celda_t*)*CAPACIDAD);
  if (sector == NULL){
    //fprintf(stderr, "Error al crear el sector\n");
    return 1;
  }
  sector->celdas = celdas;
  sector->cant_guardados = 0;
  return 0;
}

int sector_agregar_celda(sector_t* sector, celda_t* celda){
  if (sector->cant_guardados == CAPACIDAD){
    //fprintf(stderr, "El sector esta lleno\n");
    return 1;
  }
  sector->celdas[sector->cant_guardados] = celda;
  sector->cant_guardados++;
  return 0;
}

int sector_contar_apariciones(sector_t* sector, int numero){
  int contador = 0;
  for (int i = 0; i < sector->cant_guardados; i++){
    if (celda_obtener_numero(sector->celdas[i]) == numero){
      contador++;
    }
  }
  return contador;
}

void sector_destruir(sector_t* sector){
  free(sector->celdas);
}
```

sep 10, 19 15:51 **impresor.h** Page 1/1

```c
#ifndef _IMPRESOR_H_
#define _IMPRESOR_H_
#include "celda.h"

//Recibe una lista de listas de celdas, el numero de filas y
//de columnas que tiene el archivo y devuelve una representacion
//en forma de cadena del mismo
char* impresor_imprimir_celdas(const celda_t** celdas, \
  int nro_filas, int nro_col);

#endif
```

sep 10, 19 15:51 **impresor.c** Page 1/1

```c
#include "impresor.h"
#include <string.h>
#include <stdlib.h>

#define HEADER "U==========U==========U==========U\n"
#define LONG_HEADER strlen(HEADER)+1
#define SEP_MEDIO "U---+---+---U---+---+---U---+---+---U\n"
#define LONG_SEP_MEDIO strlen(SEP_MEDIO)
#define SEPARADOR_SECTOR 'U'
#define SEPARADOR_CELDA '|'

char procesar_celda(const celda_t* celda){
  if (¬celda_esta_vacia(celda)){
    return celda_obtener_numero(celda) + '0';
  }
  return ' ';
}

void procesar_fila(const celda_t** celdas, int fila,\
  int nro_col, char* repr_fila){
  int col;
  for (col = 0; col ≤ nro_col*4; col++){
    if (col % 12 ≡ 0){
      repr_fila[col] = SEPARADOR_SECTOR;
    }else if (col % 4 ≡ 0){
      repr_fila[col] = SEPARADOR_CELDA;
    }else if (col % 2 ≡ 0){
      repr_fila[col] = procesar_celda(&celdas[(fila − 1) / 2 ][(col−2) /4 ]);
    }else{
      repr_fila[col] = ' ';
    }
  }
  repr_fila[col] = '\n';
  //repr_fila[col+2] = '\0';
}

char* impresor_imprimir_celdas(const celda_t** celdas, int nro_filas,\
  int nro_col){
  char* repr_tablero = malloc(sizeof(char)*(LONG_HEADER+1)*2*(nro_col+1)+1);
  strncpy(repr_tablero, HEADER, LONG_HEADER);
  for (int fila = 1; fila < nro_filas*2; fila++){
    if (fila % 6 ≡ 0){
      strncat(repr_tablero, HEADER, LONG_HEADER);
    }else if (fila % 2 ≡ 0){
      strncat(repr_tablero, SEP_MEDIO, LONG_SEP_MEDIO);
    }else{
      char repr_fila[LONG_HEADER−1];
      procesar_fila(celdas, fila, nro_col, repr_fila);
      strncat(repr_tablero, repr_fila, LONG_HEADER−1);
    }
  }
  strncat(repr_tablero, HEADER, LONG_HEADER);
  return repr_tablero;
}
```

sep 10, 19 15:51 **cliente.h** Page 1/1

```c
#ifndef _CLIENTE_H_
#define _CLIENTE_H_
#include "socket_tcp.h"

typedef struct cliente{
  socket_tcp_t socket_tcp;
}cliente_t;

//Crea un cliente y lo conecta al servidor en el puerto y host indicado.
//PRE: el servidor debe estar inciado y escuchando en el puerto y host que
//se intenta conectar.
//POST: crea y conceta el cliente.
int cliente_crear(cliente_t* cliente, const char* host, const char* puerto);

//Pone al cliente en modo escucha para recibir comandos por STDIN, hasta que
//se ingrese el comando de salida: "exit".
//PRE: el cliente debe estar creado correctamente.
//POST: verifica y ejecuta los comandos recibidos. Devuelve 1
void cliente_recibir_comandos(cliente_t* cliente);

//Destruye el cliente liberando los recursos.
void cliente_destruir(cliente_t* cliente);

#endif
```

```c
#define _POSIX_C_SOURCE 200809L
#include "cliente.h"
#include "socket_tcp.h"
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <arpa/inet.h>

#define TAMANIO_BUFFER_COMANDOS 16
#define TAMANIO_MSJ_SERVIDOR 4
#define MSJ_VERIFICAR "V"
#define MSJ_REINICIAR "R"
#define MSJ_OBTENER "G"
#define MSJ_PONER "P"

#define MSJ_COMANDO_INVALIDO "Comando invalido, intente nuevamente\n"
#define MSJ_INDICE_INVALIDO "Error en los Â-ndices. Rango soportado: [1,9]\n"
#define MSJ_VALOR_INVAL "Error en el valor ingresado. Rango soportado: [1,9]\n"
#define MSJ_INGRESO_COMANDOS "Ingrese un comando: "


int cliente_crear(cliente_t* cliente, const char* host, const char* puerto){
  socket_tcp_crear(&cliente→socket_tcp);
  if (socket_tcp_conectar(&cliente→socket_tcp, host, puerto) ≠ 0){
    //fprintf(stderr, "No se pudo crear el cliente, error al conectar\n");
    socket_tcp_destruir(&cliente→socket_tcp);
    return 1;
  }
  return 0;
}

void enviar_al_servidor(cliente_t* cliente, const char* mensaje){
  socket_tcp_enviar(&cliente→socket_tcp, mensaje , strlen(mensaje));
}

char* recibir_mensaje_servidor(cliente_t* cliente){
  unsigned int tamanio;
  socket_tcp_recibir(&cliente→socket_tcp, &tamanio, TAMANIO_MSJ_SERVIDOR);
  tamanio = ntohl(tamanio);
  char* mensaje = malloc(sizeof(char)*tamanio);
  if (mensaje ≡ NULL){
    return NULL;
  }
  socket_tcp_recibir(&cliente→socket_tcp, mensaje, tamanio);

  return mensaje;
}

void realizar_comando(cliente_t* cliente, const char* mensaje_enviar){
  enviar_al_servidor(cliente, mensaje_enviar);
  char* mensaje_recibido = recibir_mensaje_servidor(cliente);
  printf("%s", mensaje_recibido);
  free(mensaje_recibido);
}

int esta_en_rango(int numero){
  return (numero > 0) ∧ (numero ≤ 9);
}

int verificar_valores(int valor, int fila, int columna){
  if (¬esta_en_rango(fila) ∨ ¬esta_en_rango(columna)){
    fprintf(stderr, "%s", MSJ_INDICE_INVALIDO);
    return 1;
  }else if (¬esta_en_rango(valor)){
    fprintf(stderr, "%s", MSJ_VALOR_INVAL);
    return 1;
  }
  return 0;
}

char* crear_comando_poner(int valor, int fila, int columna){
  char* comando = malloc(sizeof(char) * 5);
  if (comando ≡ NULL){
    return NULL;
  }
  comando[0] = 'P';
  comando[1] = (unsigned char) fila;
  comando[2] = (unsigned char) columna;
  comando[3] = (unsigned char) valor;
  comando[4] = '\0';
  return comando;
}


char* parsear_comando(char* comando){
  char* parametros[4];
  char* comando_poner = NULL;
  for (int i = 0; i < 4; i++){
    char* param = strtok_r(NULL, ",", &comando);
    parametros[i] = param;
  }
  int valor = atoi(parametros[0]);
  int fila = atoi(parametros[2]);
  int columna = atoi(parametros[3]);
  if (verificar_valores(valor, fila, columna) ≡ 0){
    comando_poner = crear_comando_poner(valor, fila, columna);
  }
  return comando_poner;
}
void cliente_recibir_comandos(cliente_t* cliente){
  //char* comando = malloc(sizeof(char) * TAMANIO_BUFFER_COMANDOS);

  // if (comando == NULL){
  //  return;
  // }
  char* comando = NULL;
  char* guardado;
  size_t tamanio;
  //printf("%s\n", MSJ_INGRESO_COMANDOS);
  while (getline(&comando, &tamanio, stdin) > 0){
    if (strlen(comando) > TAMANIO_BUFFER_COMANDOS){
      fprintf(stderr, "%s\n", MSJ_COMANDO_INVALIDO);
      continue;
    }
    comando[0] = toupper(comando[0]);
    char * primera_palabra = strtok_r(comando, " ", &guardado);

    if (strcmp(primera_palabra, "Verify\n") ≡ 0){
      realizar_comando(cliente, MSJ_VERIFICAR);
    }else if (strcmp(primera_palabra, "Get\n") ≡ 0){
      realizar_comando(cliente, MSJ_OBTENER);
    }else if (strcmp(primera_palabra, "Put") ≡ 0){
      char* comando_poner = parsear_comando(guardado);
      if (comando_poner ≡ NULL) continue;
      realizar_comando(cliente, comando_poner);
      free(comando_poner);
    }else if (strcmp(primera_palabra, "Reset\n") ≡ 0){
      realizar_comando(cliente, MSJ_REINICIAR);
    }else if (strcmp(primera_palabra, "Exit\n") ≡ 0){
      break;
    }else{
      fprintf(stderr, "%s\n", MSJ_COMANDO_INVALIDO);
```

sep 10, 19 15:51 **cliente.c** Page 3/3

```
133       }
134     }
135     free(comando);
136   }
137
138   void cliente_destruir(cliente_t* cliente){
139     socket_tcp_destruir(&cliente→socket_tcp);
140   }
```

sep 10, 19 15:51 **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 10, 19 15:51 **celda.c** Page 1/1

```c
#include "celda.h"
#include <stdio.h>


bool es_numero_valido(int numero){
  return (numero ≥ 0) ∧ (numero ≤ 9);
}

int celda_crear(celda_t* celda, int numero, bool es_modificable){
  if ( ¬es_numero_valido(numero) ){
    //printf("El numero no es valido\n");
    return 1;
  }
  celda→numero = numero;
  celda→modificable = es_modificable;
  return 0;
}


int celda_obtener_numero(const celda_t* celda){
  return celda→numero;
}

bool celda_esta_vacia(const celda_t* celda){
  return celda→numero ≡ 0;
}

bool celda_es_modificable(celda_t* celda){
  return celda→modificable;
}

void celda_poner_numero(celda_t* celda, int numero){
  if ( ¬es_numero_valido(numero) ){
    //printf("El numero no es valido\n");
    return;
  } else if ( ¬celda→modificable ){
    //printf("La celda no es modificable\n");
    return;
  }

  celda→numero = numero;
}
```

sep 10, 19 15:51 **Table of Content** Page 1/1

Table of Contents