Práctica 2
Sudoku

Fecha de entrega: 12 de mayo de 2019

El Sudoku (en japonés, 数独) es un rompecabezas matemático de colocación que se hizo popular en Japón a finales del siglo XX y que cuenta con muchos adeptos en todo el mundo desde hace años.

Su resolución requiere paciencia y ciertas dotes lógicas, y tradicionalmente consiste en rellenar un tablero de 9×9 celdas, es decir 81 casillas, dividido en regiones o cajas de 3×3 con las cifras del 1 al 9 partiendo de algunos números ya dispuestos en algunas de las casillas.

La única regla que hay que respetar a la hora de resolver un Sudoku es que no se repita ninguna cifra en una misma fila, columna o región. Suena sencillo, ¿verdad?, pero que no te engañe, el grado de dificultad puede llegar a ser diabólico.

A continuación tienes un Sudoku de nivel de dificultad bajo y su solución.

		3			1	7	2	
					4			
			7			1	4	9
	1	4	8					5
2	8						7	4
7					2	6	8	
9	5	2			8			
			3					
	6	7	9			4		

Tablero de partida

4	9	3	5	8	1	7	2	6
1	7	6	2	9	4	8	5	3
5	2	8	7	3	6	1	4	9
6	1	4	8	7	9	2	3	5
2	8	5	6	1	3	9	7	4
7	3	9	4	5	2	6	8	1
9	5	2	1	4	8	3	6	7
8	4	1	3	6	7	5	9	2
3	6	7	9	2	5	4	1	8

Juego resuelto

El objetivo de esta práctica es desarrollar una aplicación que nos permita resolver sudokus, llevar un registro de jugadores con sus puntuaciones y registrar nuevos sudokus para resolver.

1. Descripción de la funcionalidad de la aplicación

En esta práctica hay que ir desarrollando de manera incremental un programa que permita resolver sudokus así como mantener un registro de jugadores. Además los usuarios podrán incorporar nuevos sudokus con los que desafiar a otros jugadores.

La aplicación, nada más arrancar, deberá cargar desde archivos de texto, en listas en memoria, el registro de jugadores así como la colección de sudokus entre los que se podrá elegir para jugar.

Si las cargas se han podido realizar, se presentará un menú principal que, de forma cíclica y hasta que no se solicite finalizar la ejecución, permitirá elegir entre:

1.- Jugar

2.- Ver jugadores ordenados por identificador

3.- Ver jugadores ordenados por puntos

4.- Incorporar sudoku

0.- Salir

Si alguna carga no tiene éxito la aplicación deberá finalizar.

Opción *Jugar*

Si el usuario elige la opción de jugar se le mostrará la lista de sudokus entre los que puede elegir. El sudoku elegido se cargará desde archivo y después se mostrará un menú secundario que también actuará de forma cíclica hasta que el usuario elija abortar la resolución del sudoku y volver al menú principal, y cuyas opciones son:

1.- Ver posibles valores de una casilla vacía

2.- Colocar valor en una casilla

3.- Borrar valor de una casilla

4.- Reiniciar el tablero

5.- Autocompletar celdas simples

0.- Abortar la resolución y volver al menú principal

Además de salir de este menú secundario a demanda del jugador (opción 0), también se saldrá automáticamente cuando el jugador haya terminado de rellenar el tablero. En este segundo caso, antes regresar al menú principal, se solicitará al jugador su identificador y se actualizará su información en la lista que contiene el registro de jugadores. La actualización puede consistir en aumentar su puntuación con los nuevos puntos obtenidos (si el jugador se encontraba ya en la lista) o en incorporarlo a la lista con la puntuación obtenida.

Mientras que el usuario esté jugando, el tablero actualizado estará siempre a la vista del usuario en la pantalla.

A continuación se describen las acciones asociadas a cada opción del menú secundario:

- La opción 1 (Ver posibles valores de una casilla vacía) sirve para visualizar por pantalla todos los valores válidos con los que podría rellenarse la casilla vacía indicada por el jugador.
- La opción 2 (Colocar valor en una casilla) permite colocar el valor proporcionado por el jugador en la casilla vacía que haya indicado. Sólo se permite colocar un valor que forme parte del conjunto de valores posibles para esa casilla vacía.
- La opción 3 (Borrar valor de una casilla) vacía la casilla indicada. Sólo se pueden vaciar casillas rellenadas por el jugador, es decir, no se puede vaciar una casilla fija (casilla dada como valor de partida del juego).
- La opción 4 (Reiniciar el tablero) permite reanudar el juego con el tablero de partida.
- La opción 5 (Autocompletar celdas simples) hace que la aplicación rellene automáticamente aquellas casillas vacías que tienen un único valor candidato y que, por tanto, puede considerarse definitivo para ese estado del tablero.
- La opción 0 (Abortar la resolución y volver al menú principal) permite al usuario dejar de jugar y volver al menú principal.

Opción *Ver jugadores ordenados por identificador*

Esta opción visualiza el registro de jugadores (identificadores y puntos) tal cual está por defecto, es decir, ordenado crecientemente por identificador.

Opción *Ver jugadores ordenados por ranking*

Esta opción visualiza el registro de jugadores (identificadores y puntos) ordenado decrecientemente por puntos, y a igualdad de puntos ordenada crecientemente por identificador.

Opción *Incorporar sudoku*

Esta opción permite al usuario introducir el nombre del fichero que contiene un nuevo sudoku para resolver así como los puntos que permitiría conseguir, y lo incorpora en la lista que contiene la colección de sudokus. No se permite incorporar nuevos sudokus cuyo fichero coincida con el de uno ya existente en la lista.

Opción *Salir*

Tras elegir esta opción se actualizará el fichero que contiene el registro de jugadores y el que contiene la colección de sudokus con los contenidos de las correspondientes listas. Una vez hecho esto, se cerrará la aplicación.

2. Versión 1

En esta primera versión al arrancar la aplicación se cargará únicamente la colección de sudokus. Si la carga se puede realizar, se presentará un menú principal que, de forma cíclica y hasta que no se solicite finalizar la ejecución, permitirá elegir entre un número de opciones reducido a:

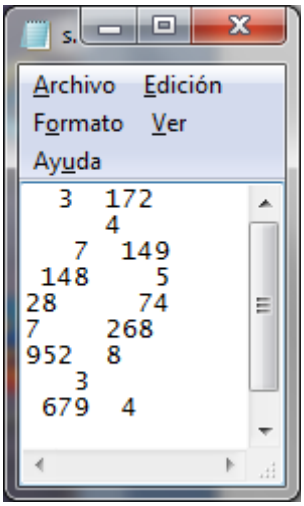
1.- Jugar

0.- Salir

En esta primera versión de la práctica la opción Jugar debe comportarse como se ha indicado en el apartado 1 salvo que no se realizará la actualización de la lista de jugadores en caso de éxito. En lugar de actualizar la lista (que ni siquiera se habrá cargado) simplemente se mostrará por pantalla un mensaje indicando los puntos obtenidos (caso de haber completado el sudoku). Por su parte, la opción Salir simplemente finalizará la ejecución pues la colección de sudokus cargada no habrá sufrido ningún cambio.

La colección de sudokus estará compuesta por un máximo de 20 sudokus. La información disponible acerca de cada sudoku será el nº de puntos que permite obtener al resolverlo y el nombre del archivo que contiene una representación del tablero del sudoku. La colección de sudokus se encontrará en el archivo `listaSudokus.txt` que a lo sumo tendrá la información de 20 sudokus. Cada línea de dicho archivo contiene la información de un sudoku (primero el nombre del archivo y después el nº de puntos). La colección almacenada en el archivo estará ordenada ascendentemente por el nº de puntos y, a igualdad en el nº de puntos, ascendentemente por el nombre del archivo[1].

Por lo que respecta al formato y contenido de los archivos que albergan los tableros de los sudokus, cada línea contiene una cadena de 9 caracteres que representa una fila del tablero. La siguiente imagen es la del archivo que contendría el tablero de partida del sudoku mostrado al inicio de este enunciado.



[1] Esta ordenación no es relevante para esta primera versión de la práctica pero sí lo será para la segunda.

2.1.Detalles de implementación

Se deben desarrollar módulos para los diferentes tipos y sus operaciones asociadas. A modo de breve análisis de los diferentes tipos necesarios téngase en cuenta que:

- Hay que representar la información asociada a la lista de sudokus seleccionables para jugar, lista que será cargada desde archivo. Cada elemento de esta lista corresponderá a un sudoku seleccionable y la información a representar para cada elemento será el nº de puntos que permite conseguir el sudoku y el nombre del archivo donde está el tablero con el que se comienza.
- Una vez seleccionado un sudoku para jugar la aplicación deberá cargarlo en memoria desde el correspondiente archivo. Se debe disponer de un tipo que permita representar el tablero del sudoku en memoria. El tablero será una matriz cuadrada, de 9x9 casillas. En cada casilla deberá haber información acerca del estado de la casilla (vacía, fija, rellenada) y también del nº existente en la misma (lo habrá si es fija o rellenada). Para hacer más eficiente la visualización de los posibles valores de una casilla vacía y el control del valor que se coloca en una casilla vacía se incluirá también en la casilla el conjunto de valores posibles de la misma. El conjunto de valores posibles para una casilla vacía deberá ser actualizado convenientemente cada vez que se produzca algún cambio en el estado de otra casilla que se encuentre en su misma fila, columna o región.
- Finalmente, se contará con un tipo que permita representar la información completa del juego: aparte del propio tablero hay que tener constancia del sudoku elegido (para poder reiniciar el tablero) y saber si el tablero está completamente relleno (para poner fin al juego de forma automática).

Módulo Conjunto

Incluirá un tipo de datos `tConjunto` que permita representar conjuntos de valores enteros en el intervalo [1, 9]. Las variables de tipo `tConjunto` se utilizarán para representar los conjuntos de valores posibles de las casillas vacías. El módulo deberá incluir además la implementación de las siguientes operaciones:

- `void cjto_vacio(tConjunto & c)`: inicializa el conjunto `c` al conjunto vacío.
- `void cjto_lleno(tConjunto & c)`: inicializa el conjunto `c` al conjunto formado por todos los valores del intervalo [1, 9].
- `bool pertenece(const tConjunto & c, int e)`: devuelve un booleano que indica si el elemento `e` (nº entero ∈ [1, 9]) se encuentra en el conjunto `c`.
- `void annadeElemento(tConjunto & c, int e)`: mete el elemento `e` (nº entero ∈ [1, 9]) en el conjunto `c`.
- `void borraElemento(tConjunto & c, int e)`: saca el elemento `e` (nº entero ∈ [1, 9]) del conjunto `c`.
- `int numElems(const tConjunto & c)`: devuelve el nº de elementos que hay en el conjunto `c`.
- `bool esUnitario(const tConjunto & c, int & e):` devuelve un booleano que indica si el conjunto `c` tiene un único elemento y, de ser así, lo devuelve.

- `void mostrar(const tConjunto & c)`: visualiza por pantalla los elementos del conjunto `c`.

Módulo Casilla

Incluirá el tipo de datos `tCasilla` que representa la información para una casilla del tablero: estado de la casilla (vacía, fija, rellenada), nº existente en la misma (en caso de que el estado sea fija o rellenada) y el conjunto de valores posibles de la casilla de tipo `tConjunto` (en caso de ser vacía). Se incluirán, al menos, las siguientes operaciones:

- `void iniciaCasilla(tCasilla & casilla)`: inicializa el estado de la casilla dada a vacío y su conjunto de valores posibles al conjunto {1,2,3,4,5,6,7,8,9}.
- `void rellenaCasilla(tCasilla & casilla, char c, bool fija=false)`[2]: rellena el estado y el nº de `casilla` de acuerdo con los valores de `fija` y `c`. Si `c` es el carácter espacio en blanco se tratará de una casilla vacía; si `c` ∈ ['0', '9'] el nº que almacena esa casilla será el entero equivalente y `fija` indicará si se trata de una casilla fija (valor `true` para `fija`) o rellenada (valor por defecto `false`).
- `void dibujaCasilla(const tCasilla & casilla)`: pinta en pantalla la casilla dada. Se mostrarán sobre fondo azul las casillas fijas y sobre fondo rojo las casillas rellenadas. El resto, sobre el fondo negro por defecto.
- `bool esSimple(const tCasilla & casilla, int & numero)`: devuelve un valor que indica si una casilla vacía tiene un único valor posible y, en caso afirmativo, también lo devuelve; devuelve `false` para casillas fijas o rellenadas.

A la hora de dibujar las casillas fijas y rellenadas hay que cambiar el color de fondo (por defecto es negro). A continuación explicamos cómo hacerlo utilizando rutinas que son específicas de Visual Studio, por lo que debemos ser conscientes de que el programa no será portable a otros compiladores.

Visual Studio incluye una biblioteca, `Windows.h`, que tiene, entre otras, rutinas para la consola. Una de ellas es `SetConsoleTextAttribute()`, que permite ajustar los colores de fondo y primer plano. Incluye esa biblioteca y el siguiente subprograma:

```
void colorFondo(int color){
     HANDLE handle = GetStdHandle(STD_OUTPUT_HANDLE);
     SetConsoleTextAttribute(handle, 15 | (color << 4));
}
```

Al invocarlo con un color para el fondo (0 = negro; 1 = azul; 4 = rojo) lo establecerá, y usará el blanco (15) como color de primer plano.

Módulo Tablero

[2] Este subprograma se utilizará tanto al rellenar cada casilla durante la carga del tablero desde archivo como cuando se actualice una casilla (bien para colocar un valor, bien para borrarlo). En el primer caso se invocará con valor `true` para el tercer parámetro; en el segundo caso se usará el valor `false` por defecto para dicho parámetro.

Contendrá un tipo de datos `tTablero` que representa el tablero de un sudoku mediante una matriz cuadrada cuyos elementos son de tipo `tCasilla`. Se incluirán, al menos, las siguientes operaciones:

- `void iniciaTablero(tTablero tablero)`: inicializa el tablero `tablero` de forma que todas las casillas sean vacías y tengan como valores posibles todos los del intervalo [1, 9].
- `bool cargarTablero(string nombreFichero, tTablero tablero)`: recibe el tablero `tablero` inicializado como hace el subprograma anterior, lo actualiza con la información almacenada en el archivo de nombre `nombreFichero` que contiene el tablero del sudoku a jugar y devuelve un booleano que indica si la carga se ha podido realizar. Los valores posibles de las casillas deberán quedar convenientemente actualizados.
- `void dibujarTablero(const tTablero tablero)`: muestra por pantalla el tablero dado.
- `bool ponerNum(tTablero tablero, int fila, int col, int c)`: coloca el nº `c` en la casilla de coordenadas (`fila`,col) (`fila` y `col` estarán en el intervalo [1,9]) del tablero `tablero`; el booleano que devuelve indica si la acción ha sido posible (para ello la casilla debe ser vacía y `c` debe ser uno de los valores posibles de la misma). Si es posible colocar el número dado en la casilla indicada deberán actualizarse convenientemente los valores posibles de las casillas que puedan verse afectadas (las de la misma fila, columna y submatriz que la casilla dada).
- `bool borrarNum(tTablero tablero, int fila, int col):` borra el número que haya colocado en la casilla de coordenadas (`fila`,col) (`fila` y `col` estarán en el intervalo [1,9]) del tablero `tablero` y la casilla volverá a estar en estado vacío; el booleano que devuelve indica si la acción ha sido posible (debe ser una casilla que haya sido previamente rellenada). Si es posible borrar el número de la casilla indicada deberán actualizarse convenientemente los valores posibles de las casillas que puedan verse afectadas (las de la misma fila, columna y submatriz que la casilla dada).
- `bool tableroLleno (const tTablero tablero)`: devuelve un valor booleano que indica si el tablero dado está relleno por completo.
- `void mostrarPosibles(const tTablero tablero, int fila, int col):` muestra los valores posibles de la casilla del tablero dado que tiene coordenadas (`fila`,col) (`fila` y `col` estarán en el intervalo [1,9])
- `void rellenarSimples(tTablero tablero)`: en cada casilla que tiene un único valor posible se pone dicho valor y se actualizan convenientemente los valores posibles de las casillas que puedan verse afectadas.

Módulo Juego

Incluirá dos tipos: `tSudoku` y `tJuego`. `tSudoku` representa la información de un sudoku seleccionable de la colección de sudokus disponibles: nº de puntos que permite conseguir el sudoku y el nombre del archivo donde está el tablero con el que se comienza. `tJuego` permite representar la información del sudoku elegido (`tSudoku`), el tablero de dicho sudoku (`tTablero`) y un valor que indique si el tablero está completamente relleno o no. Además se incluirán, al menos, las siguientes operaciones:

- `void iniciaJuego(tJuego & juego, const tSudoku & sudoku)`: recibe en `sudoku` la información del sudoku elegido para jugar e inicializa el parámetro `juego` a un juego no acabado con dicha información como registro del sudoku a jugar y con un tablero inicializado según se ha indicado en el módulo anterior.
- `void mostrarJuego(const tJuego & juego)`: muestra por pantalla la información del sudoku que se va a jugar así como el tablero del mismo.
- `bool cargaJuego(tJuego & juego, const tSudoku & sudoku)`: actualiza el tablero del parámetro `juego` con el contenido del archivo cuyo nombre figura en el parámetro de tipo `tSudoku` recibido.
- `int jugarUnSudoku(const tSudoku & sudoku)`: dada la información del sudoku elegido lleva a cabo todas las acciones correspondientes a haber elegido la opción 1 de esta versión y devuelve la puntuación obtenida por el jugador (0 si aborta la resolución antes de rellenar el tablero o los puntos asociados al sudoku elegido en caso de resolverlo).

Módulo ListaSudokus

Incluirá un tipo de datos `tListaSudokus` que permita guardar en memoria la información de una lista de hasta `MAX_SUDOKUS` sudokus seleccionables (`MAX_SUDOKUS` = 20). Cada elemento de esta lista será, por tanto, de tipo `tSudoku`. Deberá incluir además la implementación de, al menos, las siguientes operaciones:

- `void creaListaVacia(tListaSudokus & lista)`: inicializa `lista` a una lista vacía.
- `bool cargar(tListaSudokus & lista)`: guarda en `lista` el contenido del archivo `listaSudokus.txt`; devuelve un booleano que indica si la carga se ha podido realizar.
- `void mostrar(const tListaSudokus & lista)`: visualiza por pantalla la lista de sudokus dada.

3. Versión 2

En esta segunda versión se desarrollará la funcionalidad completa de la aplicación descrita en el apartado 1.

La lista de jugadores se cargará desde el archivo `listaJugadores.txt` donde en cada línea aparece la información de un jugador: primero su identificador y después el número de puntos que ha conseguido hasta el momento. El listado de jugadores del archivo está ordenado alfabéticamente por el identificador de los jugadores y ese orden se debe mantener en memoria. No puede haber dos jugadores en la lista con el mismo identificador. En esta versión el archivo `listaJugadores.txt` guardará información para un máximo de 10 jugadores.

3.1.Detalles de implementación

Se deben desarrollar módulos para los nuevos tipos que esta versión requiere y para sus operaciones asociadas:

- Hay que representar la información de cada jugador: identificador y puntos conseguidos.
- Hay que representar la información asociada a la lista de jugadores, lista que será cargada desde archivo.

Así mismo, teniendo en cuenta el incremento de funcionalidad que supone la versión 2, habrá que actualizar el módulo ListaSudokus incorporando nuevas operaciones que permitan, p.e., añadir un nuevo sudoku a la lista de sudokus, actualizar el archivo `listaSudokus.txt` que contiene la colección de sudokus, etc. Como ya se ha indicado previamente, la lista de sudokus que se carga desde el archivo estará ordenada ascendentemente por el nº de puntos y, a igualdad en el nº de puntos, ascendentemente por el nombre del archivo. Esta ordenación debe mantenerse durante la ejecución, aspecto que debe ser tenido en cuenta en operaciones como la de añadir un nuevo sudoku. También hay que tener en cuenta que en la lista no puede haber dos sudokus que coincidan en el nombre del fichero.

Módulo ListaSudokus

En este módulo, ya existente, deberán incorporarse, al menos, las siguientes operaciones:

- `bool guardar(const tListaSudokus & lista)`: almacena en el archivo `listaSudokus.txt` el contenido de `lista` y devuelve un valor booleano indicando si la acción fue posible. Debe respetar el formato indicado para el archivo.
- `bool registrarSudoku(tListaSudokus & lista)`: solicita los datos de un nuevo sudoku (nombre del fichero y puntos que permite conseguir) y si no existe un sudoku en `lista` con igual nombre de fichero lo inserta en la posición adecuada respetando el orden existente. Se devuelve un booleano que indica si se pudo registrar un nuevo sudoku, para lo cual también hay que tener en cuenta si la lista está o no llena.
- `bool buscarFichero(const tListaSudokus & lista, string nombreFich)`: devuelve un boleano que indica si existe o no un sudoku en `lista` con nombre de fichero igual a `nombreFich`.
- `int buscarPos(const tListaSudokus & lista, const tSudoku & sudoku)`: devuelve la posición de `lista` en la que debería insertarse `sudoku` para respetar el orden existente en la lista. Debe implementar una búsqueda binaria.

Módulo Jugador

Incluirá el tipo `tJugador` que representa la información de un jugador y, al menos, las siguientes operaciones:

- `string toString(tJugador jugador)`: devuelve la información del jugador dado en un formato de cadena de caracteres (una cadena de caracteres donde primero aparece el identificador y después los puntos conseguidos por el jugador, separados por uno o más espacios en blanco)

- `void modificarJugador(tJugador & jugador, int puntos)`: añade `puntos` puntos a los puntos que lleva conseguidos el jugador dado.
- `bool operator<(const tJugador & opIzq, const tJugador & opDer)`: sobrecarga del operador < para datos del tipo `tJugador` (se compara en base al identificador).
- `bool menor(const tJugador & j1, const tJugador & j2)`: devuelve `true` si el jugador `j1` tiene menos puntos que el jugador `j2`, o si tienen los mismos puntos pero el identificador del jugador `j2` es menor que el del jugador `j1`; `false` en caso contrario.

Módulo ListaJugadores

Incluirá un tipo de datos `tListaJugadores` que permita guardar en memoria la información de una lista de hasta `MAX_JUGADORES` jugadores (`MAX_JUGADORES` = 10). Cada elemento de esta lista será, por tanto, de tipo `tJugador`. Deberá incluir además la implementación de, al menos, las siguientes operaciones:

- `void creaListaVacia(tListaJugadores & lista)`: inicializa `lista` a una lista vacía.
- `bool cargar(tListaJugadores & lista)`: guarda en `lista` el contenido del archivo `listaJugadores.txt`; devuelve un booleano que indica si la carga se ha podido realizar.
- `void mostrar(const tListaJugadores & lista):` visualiza por pantalla la lista de jugadores dada.
- `bool guardar(const tListaJugadores & lista):` almacena en el archivo `listaJugadores.txt` el contenido de `lista` y devuelve un valor booleano indicando si la acción fue posible. Debe respetar el formato indicado para el archivo.
- `void puntuarJugador(tListaJugadores & lista, int puntos)`: solicita que se introduzca el identificador de un jugador por teclado y se actualiza su información en `lista`. La actualización puede consistir en aumentar su puntuación con los nuevos puntos obtenidos (si el jugador se encontraba ya en la lista) o en incorporarlo a la lista con la puntuación obtenida (si no está llena).
- `bool buscar(const tListaJugadores & lista, string id, int & pos):` busca al jugador con identificador `id` en `lista`; devuelve `true` y la posición (`pos`) en la que se encuentra si el jugador está en la lista; devuelve `false` y la posición (`pos`) en la que debería estar si el jugador no está en la lista. Debe implementar una búsqueda binaria.
- `tListaJugadores ordenarPorRanking(const tListaJugadores & lista)`: devuelve una copia de la lista dada ordenada por ranking (decrecientemente por puntos, y a igualdad de puntos crecientemente por identificador).

4. Versión 3

La funcionalidad de esta última versión de la práctica es la misma que la de la versión 2 pero se usará memoria dinámica para la lista de jugadores (en concreto la

lista de jugadores usará un array dinámico de punteros a datos de tipo `tJugador`) y se deberá implementar de forma recursiva algún algoritmo concreto.

En esta versión no hay un número límite de jugadores almacenados en el archivo `listaJugadores.txt`.

4.1.Detalles de implementación

A continuación se indican aquellos módulos que sufren cambios en esta última versión así como indicaciones para la detección de fugas de memoria.

Módulo Jugador

Deberá incorporarse la definición de un tipo `tJugadorPtr` que represente punteros a datos de tipo `tJugador`.

Módulo ListaJugadores

Deberá cambiarse la definición del tipo `tListaJugadores` de forma que ahora esté sustentado por un array dinámico cuyas componentes sea de tipo `tJugadorPtr` e incluya además la dimensión que en cada momento de la ejecución tiene el array dinámico. La dimensión inicial será de 5 elementos. De esta forma la dimensión del array que sustenta la lista ya no será un impedimento a la hora de incorporar a un nuevo jugador: si la lista está llena simplemente se redimensionará el array a uno mayor y seguidamente se incluirá el nuevo jugador en el punto adecuado de la lista.

Deben incorporarse, al menos, las siguientes operaciones:

- `void ampliar(tListaJugadores & lista)`: amplía la dimensión del array dinámico de `lista` al doble de la que tiene. Los datos de los jugadores que ya existen en la lista deben mantenerse.
- `void borrarListaJugadores(tListaJugadores & lista)`: libera la memoria dinámica usada por `lista`.

Además deben actualizarse adecuadamente las implementaciones de todos los subprogramas de este módulo que puedan verse influenciados por la nueva definición del tipo `tListaJugadores`. Adicionalmente debe sustituirse la implementación iterativa de la búsqueda binaria del subprograma `bool buscar(const tListaJugadores & lista, string id, int & pos)` desarrollada en la versión 2 por una implementación recursiva.

Detección de fugas de memoria

La aplicación no puede tener fugas de memoria. Para llevar un control de las fugas de memoria se debe hacer lo siguiente:

- Para que al terminar la ejecución de la aplicación se muestre información sobre la basura que haya podido quedarse sin liberar hay que añadir al inicio de la función `main` el comando:
  ```
  _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
  ```
- Además, para que como parte de dicha información se muestre el nombre del módulo y la línea de código, debe añadirse al proyecto un archivo

`checkML.h` con las siguientes directivas de Visual Studio e incluir dicho archivo en los archivos de código fuente (archivos .cpp) del proyecto. El contenido de dicho archivo es el siguiente:

```
// archivo checkML.h
#ifdef _DEBUG
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#ifndef DBG_NEW
#define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
#define new DBG_NEW
#endif

#endif
```

5. Entrega de la práctica

La práctica se entregará en el Campus Virtual por medio de la tarea **Entrega de la Práctica 2**, que permitirá subir un archivo comprimido con el código fuente de los diferentes archivos .h y .cpp que componen la solución de la práctica. Uno de los dos miembros del grupo será el encargado de subirlo, no lo suben los dos.

Recordad poner el nombre de los miembros del grupo en un comentario al principio del archivo de código fuente.