

# Práctica 4

## *Gestor de partidas online*

Fecha límite de entrega: 8 de mayo

### 1. Funcionalidad de la aplicación

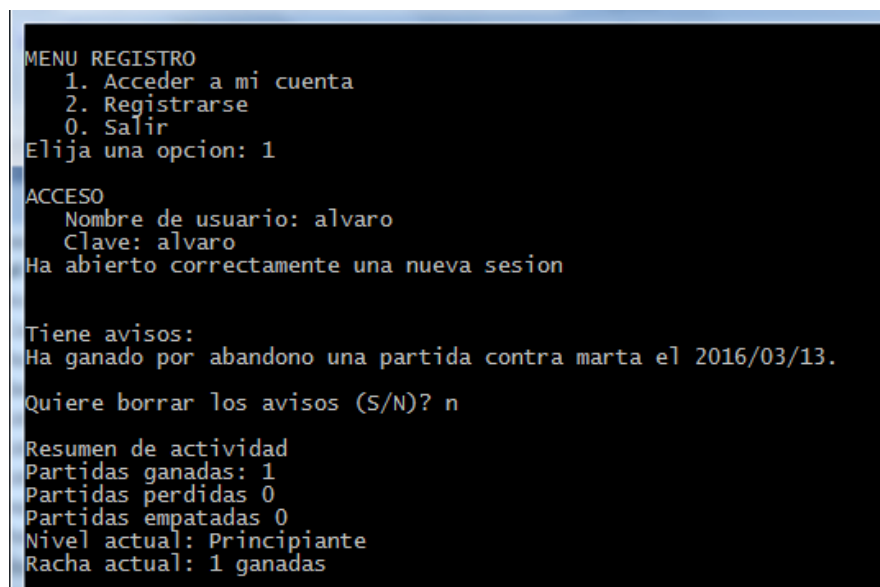
Se quiere desarrollar un gestor de partidas de 2 jugadores (en concreto del juego Conecta 4, cuyo módulo podéis encontrar en el campus) que permite a nuevos usuarios registrarse, y a los usuarios ya registrados buscar adversario para iniciar una partida, y jugar en partidas que tenga en curso. En esta modalidad de juego cada usuario se conecta cuando quiere jugar, pudiendo tener simultáneamente varias partidas en curso, hasta un máximo establecido.

Los usuarios se clasifican en categorías y las partidas se abren con adversarios de la misma categoría. Cuando un participante ha ganado suficientes partidas consecutivas asciende de categoría, por otra parte si pierde muchas partidas consecutivas baja de categoría.

Una partida puede terminar bien porque se acaba el juego, o bien porque un jugador decide abandonar. En este último caso, se da como ganador al adversario. Cuando un jugador termina una partida, el gestor avisará al adversario enviándole un aviso, que verá cuando se conecte.

#### 1.1. Registro e inicio de sesión en el gestor

El menú principal del sistema permite realizar dos opciones (mientras no se elija salir): Iniciar sesión, para usuarios ya registrados, o registrarse. En ambos casos se pide un nombre de usuario y contraseña.



```
MENU REGISTRO
1. Acceder a mi cuenta
2. Registrarse
0. Salir
Elija una opción: 1

ACCESO
Nombre de usuario: alvaro
Clave: alvaro
Ha abierto correctamente una nueva sesion

Tiene avisos:
Ha ganado por abandono una partida contra marta el 2016/03/13.

Quiere borrar los avisos (S/N)? n

Resumen de actividad
Partidas ganadas: 1
Partidas perdidas 0
Partidas empatadas 0
Nivel actual: Principiante
Racha actual: 1 ganadas
```

Figura 1: Pantalla del menú de registro del gestor

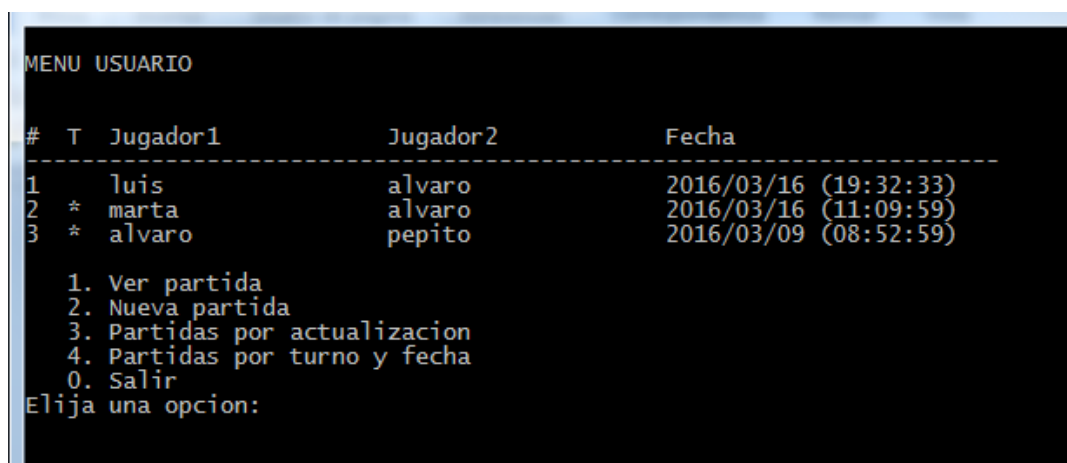
- Cuando se elige la opción de **Acceder a mi cuenta**, se comprueba que el usuario exista en la lista de usuarios del sistema y que la contraseña sea correcta. Si una de estas dos condiciones no se cumple, se muestra un error y el menú principal aparece de nuevo.

- Cuando se elige la opción de **Registrarse** se comprueba si el usuario que se quiere registrar ya existe en la lista de usuarios del sistema. Si el nuevo usuario ya existe se mostrará un error y se volverá al menú principal.
- La opción **Salir** termina la ejecución de la aplicación.

Una vez iniciada la sesión, se muestra, en primer lugar los avisos que pueda tener el usuario del gestor, y el resumen de actividad: partidas ganadas, perdidas, empatadas, nivel, racha y si está esperando (ver figura 1). Y a continuación la pantalla del menú principal de la sesión.

## 1.2. Menú principal de una sesión de usuario

Este menú muestra las partidas en curso del usuario de la sesión ordenadas por fecha de actualización, numeradas y con una marca que indica si tiene el turno.



#	T	Jugador1	Jugador2	Fecha
1		luis	alvaro	2016/03/16 (19:32:33)
2	*	marta	alvaro	2016/03/16 (11:09:59)
3	*	alvaro	pepito	2016/03/09 (08:52:59)

1. Ver partida
2. Nueva partida
3. Partidas por actualizacion
4. Partidas por turno y fecha
0. Salir

Elija una opcion:

Figura 2: Pantalla principal del usuario de una sesión del gestor

Se dispone de las siguientes opciones (mientras no se elija salir):

- **Ver partida:** Solicita al usuario el número de la partida a mostrar, comprueba dicho número, y si no ha habido error, muestra en una nueva pantalla la partida junto con las opciones que se pueden realizar sobre ella (ver sección 1.3).
- **Nueva partida:** Permite al usuario solicitar el inicio de una nueva partida. El gestor comprobará que no está ya esperando una partida y que no ha alcanzado el número máximo de partidas en curso permitido, y buscará un contrincante del mismo nivel que también quiera iniciar una nueva partida. Si lo encuentra creará la partida y la añadirá a la lista de partidas en curso de ambos usuarios. Si no, el usuario se queda en espera.
- **Partidas por actualización:** Muestra las partidas en curso ordenadas por la fecha de la última actualización.
- **Partidas por turno y fecha:** Muestra las partidas en curso ordenadas por turno (primero aquellas en las que le toca jugar) y la fecha de la última actualización.
- **Salir:** Vuelve al menú anterior.

### 1.3. Pantalla de una partida

Cuando el usuario elige la opción **Ver partida**, se solicita el número de la partida y se muestra en una nueva pantalla la partida seleccionada y las opciones disponibles:



Figura 3: Pantalla donde se muestra la partida

Las opciones del menú (a continuación vuelve al menú anterior):

- **Jugar:** Si es su turno, solicita la jugada al usuario y la ejecuta.
- **Abandonar:** Al abandonar una partida no acabada, el usuario pierde la misma.
- **Salir:** Vuelve al menú anterior.

## 2. Visión general de la implementación: módulos de la aplicación y relación entre ellos

La aplicación requiere implementar el tipo de datos `tGestor` que representa el gestor de partidas y usuarios, y que implementará todas las funcionalidades de la aplicación.

El tipo de datos `tGestor` tendrá, entre otros campos, la lista de partidas (tipo `tListaPartidas`) y la lista de usuarios (`tListaUsuarios`). Por tanto tenemos que implementar también los tipos `tPartida` y `tUsuario`.

**Importante:** el gestor sólo almacena una copia de cada partida, que se encuentra en la lista de partidas. Por su parte, un usuario no almacena sus partidas, sino que almacena una lista de registros (del tipo `tParIdEn`) que contendrán, el identificador de partida y el índice de dicha partida en la lista de partidas. Por tanto, `tUsuario` tendrá un campo (del tipo `tListaAccesoPartidas`) para la lista de acceso a las partidas en curso del usuario.

## 2.1. Organización en módulos

La práctica deberá organizarse en módulos, con un módulo independiente para cada tipo principal. Se deben definir al menos los siguientes módulos:

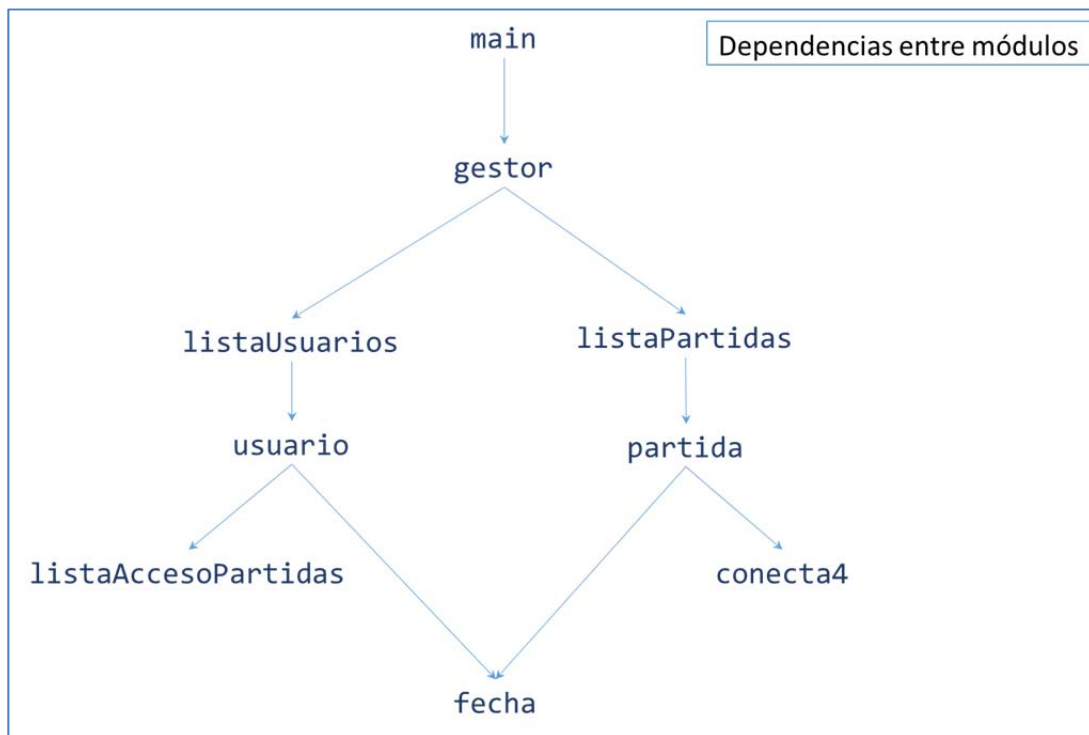


Figura 4: Grafo de dependencias entre módulos

### Módulo Conecta 4

Los archivos `conecta4.h` y `conecta4.cpp` puedes encontrarlos en el campus junto con este enunciado. Cópialos junto a los demás archivos de la práctica 4 y añádelos al proyecto. Estos archivos no debes modificarlos, para utilizarlos examina el archivo de cabecera.

### Módulo Fecha

Declara el tipo `tFecha`, una función para consultar la fecha actual y una función para mostrar las fechas con formato Año/Mes/Día, con o sin hora (ver los detalles en la sección 2.2).

### Módulo Partida

Declara el tipo de datos `tPartida` para guardar la información de una partida:

- Un identificador **único** (`string`). Para que el identificador sea único, se compone de la concatenación de la fecha y los adversarios (p.e. `143456443_pepe_marta`).
- Los 2 adversarios (`string`). Corresponderán con identificadores de usuarios en el sistema.
- La fecha de inicio y de la última actualización, de tipo `tFecha` (ver la sección 2.2).
- Estado de la partida, del tipo enumerado `tEstadoPartida` (`EnCurso`, `Terminada`).
- El estado del juego, del tipo `tConecta4`, implementado en el módulo `conecta4`.

Ofrece al menos los siguientes subprogramas:

- `void nueva(tPartida & partida, const string & jugador1, const string & jugador2)`: Recibe los identificadores de los adversarios y devuelve una partida con todos sus datos rellenos.
- `bool aplicarJugada(tPartida & partida, int col)`: Aplica la jugada `col` al juego, y si se pudo realizar, devuelve `true` y actualiza los demás campos de la partida.
- `void abandonar(tPartida & partida)`: Modifica el estado de la partida a `Terminada`, y la fecha de la última actualización.
- `void guardar(const tPartida & partida, ofstream & archivo)`: Dado un flujo de archivo de salida (ya abierto), escribe en el flujo los datos de la partida (ver detalles y ejemplo en la sección 2.2).
- `bool cargar(tPartida & partida, ifstream & archivo)`: Dado un flujo de archivo de entrada (ya abierto), lee los datos que corresponden a una partida en `partida`. Devuelve `false` si la partida no se ha podido cargar correctamente.

## Módulo ListaPartidas

Declara el tipo de datos `tListaPartidas` para gestionar una lista de partidas. **Importante:** esta lista no tiene ningún orden especial, las partidas se van añadiendo al final de la lista según se crean. Debes implementarla como una lista de tamaño variable (array con contador).

Ofrece al menos los siguientes subprogramas:

- `void guardar(const tListaPartidas & partidas, ofstream & archivo)`: Guarda la lista de partidas en el flujo (ya abierto) `archivo` (ver detalles en la sección 2.2).
- `bool cargar(tListaPartidas & partidas, ifstream & archivo)`: Carga la lista de partidas desde el flujo (ya abierto) `archivo`. Devuelve `false` si la lista no se ha podido cargar correctamente.
- `bool insertar(tListaPartidas & partidas, const tPartida & partida)`: Dada una partida, si hay espacio en la lista, la añade al final de la lista y devuelve `true`. Si no se ha podido insertar devuelve `false`.

## Módulo ListaAccesoPartidas

Declara los tipos de datos `tParIdEn` y `tListaAccesoPartidas` para gestionar la lista de partidas en curso de los usuarios. El tipo `tParIdEn` debe tener los siguientes dos campos:

- **Identificador**: El identificador de la partida.
- **Enlace**: El índice en el que se encuentra la partida en la lista de partidas del gestor.

El tipo `tParIdEn` es tan sencillo que no tiene operaciones asociadas ni módulo.

La lista `tListaAccesoPartidas` será una lista de tamaño variable con un número máximo de elementos establecido (`MAX_PAR_US = 10`). **Importante:** esta lista no tiene ningún orden especial, pudiendo ser ordenada de distintas formas según convenga.

Ofrece al menos los siguientes subprogramas:

- `void iniciar(tListaAccesoPartidas & lista)`: Inicia la lista vacía.

- `bool llena(const tListaAccesoPartidas & lista)`: Comprueba si la lista está llena.
- `bool insertar(tListaAccesoPartidas & lista, tParIdEn par)`: Dado un elemento, si la lista no está llena lo inserta al final de la lista y devuelve `true`. En otro caso devuelve `false`.
- `bool eliminar(tListaAccesoPartidas & lista, const string & id)`: Dado un identificador de partida, lo busca en la lista y si lo encuentra lo elimina de la lista (¡sin dejar huecos!). Si no lo encuentra, devuelve `false`, en otro caso devuelve `true`. **Ojo:** esta operación sólo supone que el elemento se elimina de la lista de partidas en curso del jugador, pero la partida sigue existiendo en la lista de partidas del gestor.

## Módulo Usuario

Declara el tipo de datos `tUsuario` para guardar la información de un jugador:

- El nombre o identificador del usuario (`string`). Debe ser **único** en el gestor, es decir, en la lista de usuarios del gestor.
- La contraseña del usuario (`string`).
- El nivel de juego, de tipo enumerado `tNivel` (Principiante, Medio, Experto).
- La última racha: el número de partidas ganadas o perdidas consecutivamente. Las partidas que acaban en tablas reinician la racha.
- Si está esperando un adversario para iniciar una nueva partida.
- Número de partidas ganadas, perdidas y empatadas
- Fecha de la última conexión.
- Los avisos que le envíe el gestor: un `string` en el que se van concatenando los avisos que recibe.
- La lista de acceso a las partidas que tiene abiertas, del tipo `tListaAccesoPartidas`.

Ofrece al menos los siguientes subprogramas:

- `void iniciar(tUsuario & usuario, const string & id, const string & pas)`: Recibe un identificador de usuario y una contraseña e inicia todos los campos del usuario.
- `void guardar(const tUsuario & usuario, ofstream & archivo)`: Dado un flujo de archivo (ya abierto), escribe en el flujo los datos del usuario (ver detalles y ejemplo en la sección 2.2). **Importante:** la lista de partidas en curso no se guarda.
- `bool cargar(tUsuario & usuario, ifstream & archivo)`: Dado un flujo de archivo (ya abierto), lee los datos que corresponde a un usuario en `usuario`. **Importante:** el fichero no contiene los datos de las partidas en curso (hay que iniciarla a la lista vacía).
- `bool iniciarSesion(tUsuario & usuario, const string & clave)`: Recibe una contraseña (clave) y un usuario y devuelve si la contraseña es correcta o no. Además, en caso de ser correcta, actualiza la fecha de la última conexión.
- `void actualizarAvisos(tUsuario & usu, const string & aviso)`: Actualiza el campo avisos añadiendo al final del string una nueva línea con el nuevo aviso.
- `void limpiarAvisos(tUsuario & usu)`: Actualiza el campo avisos dejando el string vacío.

- `bool nuevaPartida(tUsuario & usuario, const tParIdEn & par)`: Si el usuario no ha utilizado todas las partidas en curso que puede tener, inserta en su lista de partidas en curso la información de acceso a la partida `par`, actualiza el estado de espera y devuelve `true`. En otro caso devuelve `false`.
- `void aplicarFinPartida(tUsuario & usuario, const string & idPar, tResultado resultado)`: modifica los campos afectados por el final de una partida en curso del usuario con el resultado dado (valor del enumerado {Gana, Pierde, Empata}): partidas ganadas, perdidas, empatadas, racha, nivel y lista de partidas en curso. **Importante:** El usuario sube de nivel cada vez que consigue ganar NRN partidas consecutivas (`racha == NRN`), y baja de nivel si lleva una racha de NRN partidas perdidas (`racha == -NRN`). Un empate reinicia la racha. Define la constante `NRN = 5`.

## Módulo ListaUsuarios

Declara el tipo de datos `tListaUsuarios` para gestionar la lista de usuarios. **Importante:** esta lista se encuentra **ordenada por el identificador del usuario**. Debes implementarla como una lista de tamaño variable (array con contador).

Ofrece al menos los siguientes subprogramas:

- `bool cargar(tListaUsuarios & usuarios, ifstream & archivo)`: Carga la lista de usuarios desde el flujo (ya abierto) `archivo` (ver detalles y ejemplo en la sección 2.2).
- `void guardar(const tListaUsuarios & usuarios, ofstream & archivo)`: Guarda la lista de usuarios en el flujo (ya abierto) `archivo`.
- `bool buscar(const tListaUsuarios & usuarios, const string & idUser, int & pos)`: Dado un identificador de usuario y la lista, devuelve, si dicho identificador existe en la lista, su posición y el valor `true`, y si no existe en la lista, la posición que le correspondería y el valor `false`.
- `bool insertar(tListaUsuarios & usuarios, const tUsuario & usuario, int & pos)`: Añade un usuario en la posición de la lista que le corresponde, si hay sitio para ello y no hay otro usuario con el mismo identificador. Además devuelve un booleano indicando si la operación tuvo éxito o no, y la posición donde se ha insertado.
- `bool buscarUsuarioEsperando(const tListaUsuarios & usuarios, tNivel nivel, int & pos)`: Devuelve `true` y la posición del primer usuario del nivel dado que está esperando para iniciar una partida. Si no existe devuelve `false`.

## Módulo Gestor

Declara el tipo de datos `tGestor` para guardar la información de un gestor:

- La lista de partidas del sistema que será de tipo `tListaPartidas` y que guarda la única copia de las partidas que existirá en la aplicación.
- La lista de usuarios del sistema que será de tipo `tListaUsuarios`.
- El usuario de la sesión (índice de la posición en la lista de usuarios).
- La partida seleccionada por el usuario de la sesión (índice en la lista de partidas).

Ofrece al menos los siguientes subprogramas, que serán utilizados en `mainP4`:

- `bool arrancar(tGestor & gestor)`: Inicializa el gestor e intenta arrancarlo cargando la información del sistema. Para ello abre los archivos y carga la lista de usuarios y de partidas. Si tiene éxito, forma la lista de partidas en curso de cada uno de los usuarios y devuelve `true`. El nombre de los archivos es: `partidas.txt` y `usuarios.txt` (ver sección 2.2). Se usa una vez al inicio del `main`.
- `void generarAccesos(tGestor & gestor)`: Recorre la lista de partidas, y para las que todavía están en curso, añade el par {identificador, índice} a la lista de partidas en curso de cada uno de los adversarios de la partida.
- `void apagar(tGestor & gestor)`: Esta operación apaga el gestor, abriendo los ficheros para guardar las listas de usuarios y de partidas del sistema. Los nombres de los archivos son los mismos que al arrancar (durante las pruebas puede ser útil usar otro archivo para guardar las listas al apagar el gestor). Se usa una vez al final del `main`.

Funciones que se usan en el menú registro del `main`:

- `bool iniciarSesion(tGestor & gestor, const string & idUsu, const string & clave)`: Recibe los datos de usuario necesarios para validar la cuenta (id y contraseña) y comprueba si el usuario existe y la contraseña coincide. Devuelve cierto y actualiza el usuario de la sesión si la operación tuvo éxito.
- `bool crearCuenta(tGestor & gestor, const string & idUsu, const string & clave)`: Recibe los datos de usuario necesarios para crear una cuenta (id y contraseña) y si el id de usuario no existe y hay espacio en la lista de usuarios, crea la cuenta del usuario. Devuelve cierto y actualiza el usuario de la sesión si la operación tuvo éxito.
- `bool tieneAvisos(const tGestor & gestor, string & aviso)`: Comprueba si el usuario de la sesión tiene avisos, y en caso afirmativo los devuelve en `aviso`.
- `void limpiarAvisos(tGestor & gestor)`: actualiza los avisos del usuario de la sesión a la cadena vacía.
- `string resumenActividad(const tGestor & gestor)`: Devuelve un `string` con los datos de la actividad del usuario de la sesión, las partidas ganadas, perdidas y empatadas, el nivel, la racha y si está esperando para una nueva partida (ver figura 1).

Funciones que se usan en el menú usuario del `main`:

- `void ordenar_Fecha(tGestor & gestor)`: Ordena por la fecha de la última actualización la lista de partidas en curso del usuario de la sesión.
- `void ordenar_Turno(tGestor & gestor)`: Ordena por turno (primero aquellas en las que le toca jugar al usuario de la sesión) la lista de partidas en curso del usuario de la sesión. Utiliza la ordenación por inserción.
- `int partidasUsuario(const tGestor & gestor)`: Devuelve el número de partidas en curso del usuario de la sesión.
- `string cabecera(const tGestor & gestor, int posEnCurso)`: Dada una posición de la lista de partidas en curso del usuario de la sesión, devuelve un `string` que contiene la información que se mostrará en la lista de partidas en curso: marca (\*) si tiene el turno, nombres de los jugadores (`jugador1`, `jugador2`) y fecha de la última actualización (ver figura 2).



- **bool nuevaPartida(tGestor & gestor):** Si el usuario de la sesión no ha alcanzado el máximo número de partidas en curso y no está ya en espera, busca un usuario que esté esperando para una nueva partida, que sea del mismo nivel que el usuario de la sesión. Si lo encuentra crea la nueva partida e inserta en ambos adversarios los datos de acceso. En otro caso deja al usuario de la sesión en estado de espera para una nueva partida.
- **void apuntaPartida(tGestor & gestor, int posParEnCurso):** Dada una posición de la lista de partidas en curso del usuario de la sesión, guarda en partida seleccionada, el índice de dicha partida en la lista de partidas.

Funciones que se usan en el menú partida del main:

- **void mostrarPartida(const tGestor & gestor):** Muestra el estado de la partida en curso (ver figura 3).
- **bool esSuTurno(const tGestor & gestor):** Comprueba si es el turno del usuario de la sesión en la partida seleccionada.
- **void jugarPartida(tGestor & gestor, int col):** Aplica la jugada col a la partida, y en caso de que la partida termine, actualiza a los adversarios, según termine el juego con Ganador o Bloqueo. Además envía un aviso al contrincante indicando que ha perdido o empatado y la fecha (ver figura 2). (**Ojo:** la partida no se elimina de la lista de partidas del gestor, en la práctica 5 se pasará a una lista de partidas terminadas).
- **void abandonarPartida(tGestor & gestor):** abandona la partida seleccionada del usuario, y actualiza la información de los adversarios teniendo en cuenta que el que abandona pierde y el otro gana. Además avisa al contrincante indicando que ha ganado con la fecha (ver figura 2). (**Ojo:** la partida no se elimina de la lista de partidas del gestor, en la práctica 5 se pasará a un archivo de partidas terminadas).

## Módulo Principal (mainP4)

Habrás además un fichero `mainP4.cpp` adicional que contendrá el `main` de la aplicación y todos los menús, que comunican al usuario con el `Gestor`. Una vez iniciado el gestor, mostrará el menú inicial de registro, y una vez se identifica un usuario, mostrará el menú principal del usuario. Cuando un usuario cierra su sesión al salir del menú principal del usuario, el programa seguirá en el menú principal de registro de usuarios para un nuevo inicio de sesión, hasta que se cierre la aplicación (momento en que se apaga el gestor). En general se debe definir una función por cada opción de los menús.

Define al menos los siguientes subprogramas:

- **void menuRegistro(Gestor & gestor):** Muestra y gestiona el menú inicial de la aplicación (ver figura 1).
- **void mostrarDatosUsu(tGestor & gestor):** Si el usuario de la sesión tiene avisos, los muestra y pregunta si quiere eliminarlos. Y a continuación muestra su resumen de actividad (ver figura 1).
- **void menuUsuario(tGestor & gestor):** Muestra y gestiona el menú principal de una sesión de usuario (ver figura 2).

- `void mostrarPartidasEnCurso(tGestor & gestor):` Muestra las cabeceras de cada una de las partidas en curso del usuario de la sesión.
- `void elegirPartida(tGestor & gestor):` Solicita un número válido de partida (de entre la lista mostrada), y lo apunta como partida seleccionada del usuario de la sesión.
- `void menuPartida(tGestor & gestor):` Muestra y gestiona el menú de una partida (ver figura 3).
- `void leerJugada(int & col):` solicita y devuelve una columna válida.

## 2.2. Detalles de implementación

A la hora de completar la práctica, deberás seguir las siguientes indicaciones.

### El tipo fecha

Las fechas se representarán en el formato UNIX, es decir, como un entero con el número de segundos transcurridos desde el 1 de enero de 1970. Debes por tanto, en el módulo Fecha declarar el tipo `tFecha` mediante:

```
typedef time_t tFecha; → será necesario incluir la librería ctime
```

Deberás definir además las siguientes funciones:

- Para consultar la fecha actual:

```
tFecha fechaActual() { return time(0); }
```

- Para mostrar la fecha en formato Año/Mes/Día, con o sin hora (en formato Hora/Mins/Segs)

```
string stringFecha(tFecha fecha, bool hora) {  
    ostringstream resultado;  
    tm ltm;  
    localtime_s(&ltm, &fecha);  
    resultado << setfill('0') << setw(4) << 1900 + ltm.tm_year << '/'  
              << setw(2) << 1 + ltm.tm_mon << '/' << setw(2) << ltm.tm_mday;  
    if (hora) resultado << " (" << setw(2) << ltm.tm_hour << ':' << setw(2)  
              << ltm.tm_min << ':' << setw(2) << ltm.tm_sec << ')';  
  
    return resultado.str();  
}
```

La lectura (escritura) de la fecha de (en) fichero se realiza usando el operador habitual de extracción (inserción) de los enteros.

### El tipo stringstream

En C++ es posible operar con un string como lo hacemos con los flujos de E/S. En esta práctica puede resultarnos útil generar strings como si estuviéramos mandando información a un flujo de salida `ostream`.

Por ejemplo:

```
#include <sstream> // Es necesario incluir la biblioteca sstream
...
string resultado;
string nombre= "Pepe";
ostringstream flujo; // Flujo de salida
flujo << "Hola " << nombre << '\n'; // Pasamos datos al flujo
resultado = flujo.str(); // string del flujo, contendrá la cadena "Hola Pepe\n"
```

## Ficheros de datos

En el campus, en el archivo comprimido practica4.zip, dispones de los archivos de datos: `partidas.txt` que contiene la lista de partidas del sistema por orden de creación, y `usuarios.txt` que contiene la lista de usuarios del sistema ordenada por orden alfabético del identificador de usuario. **Importante:** la primera línea de ambos archivos es un entero no negativo que indica el número de elementos de que consta la lista guardada en el archivo.

En la figura 5 puedes ver unos archivos de ejemplo.

Cada partida se compone de 12 líneas: identificador (1 línea), jugadores (cada uno en una línea), fecha de inicio y última actualización (ambas en una línea), estado de la partida (1 línea) y el estado del juego (7 líneas).

Cada usuario se compone de varias líneas (5 + las líneas de los avisos): nombre, contraseña (cada uno en una línea), nivel, racha, si está esperando (los tres en una línea), número de partidas ganadas, perdidas y empatadas (los tres en una línea), la fecha de la última conexión (1 línea) y los avisos del gestor (número variable de líneas). El campo con los avisos que envía el gestor es un `string` compuesto de varias líneas, y en el archivo marcamos el final con la línea centinela "`_X_X_X_`" (Es conveniente definir funciones para leer y guardar este campo).

**Importante:** La lista de acceso a las partidas en curso del usuario no se guarda en el archivo de usuarios. Esta información será generada por el gestor una vez cargados los dos archivos de datos, dentro de la función `arrancar()`, se llamará a la función `generarAccesos()` que se encarga de añadir por cada partida en curso, los datos de acceso (identificador y posición) a cada uno de los adversarios.

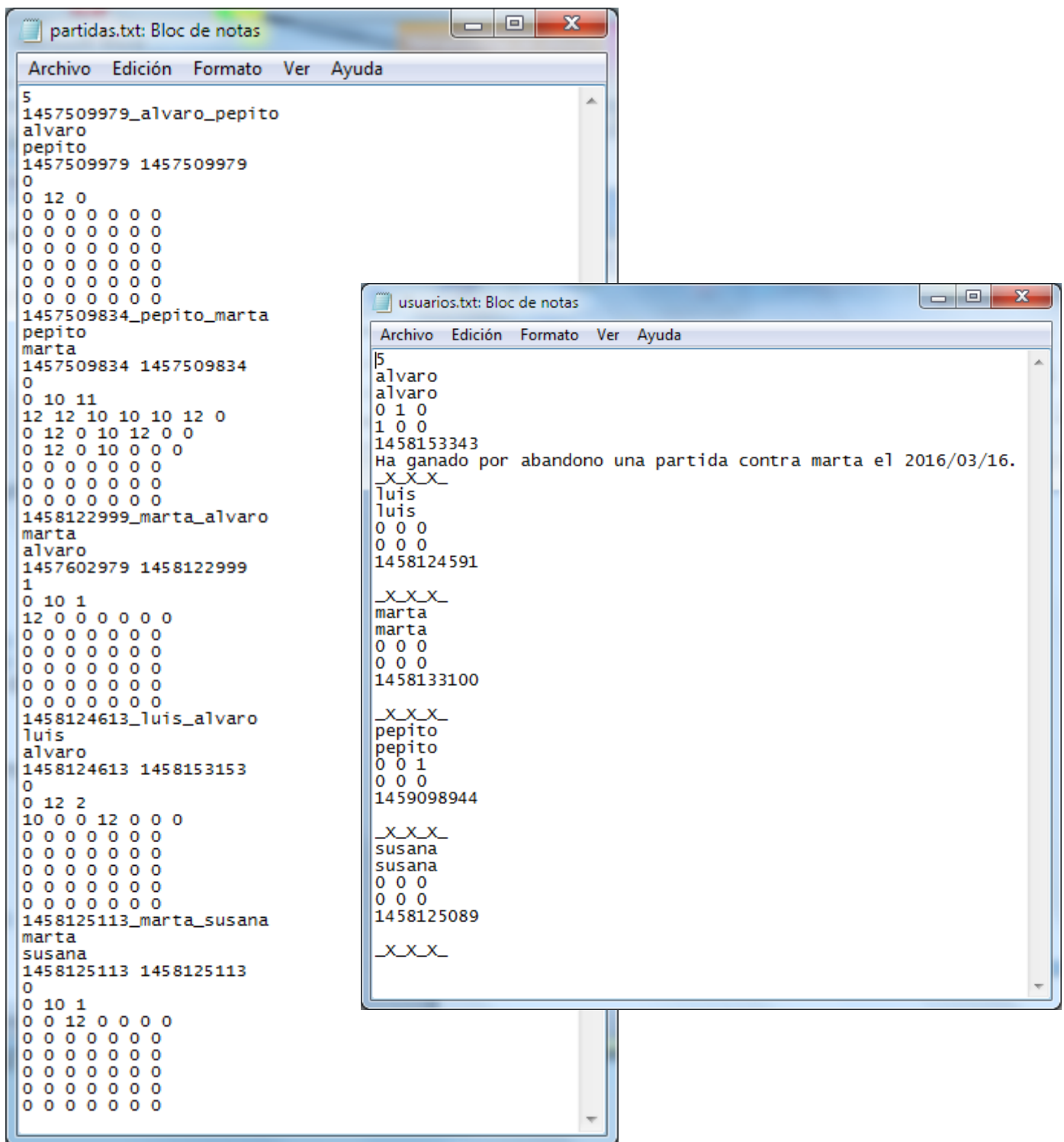


Figura 5: Ejemplo de ficheros de datos para la lista de partidas y la de usuarios

## Cargar y guardar

**Importante:** La carga y guardado en ficheros no se realiza en un único módulo. Cada módulo se encarga de cargar/guardar los datos que le corresponden.

Para esto:

- La apertura y cierre de un fichero se realiza en un único sitio: el subprograma que inicia la operación de cargar/guardar.
- Este subprograma coordina la operación invocando al resto de subprogramas de cargar/guardar, los cuales reciben el fichero (ifstream / ofstream) abierto **como parámetro por referencia** para cargar/guardar lo que corresponda.

Vamos a poner un ejemplo con la carga de la lista de partidas desde fichero. La operación de carga de la lista de partidas se inicia en el arranque del gestor, que intenta abrir el archivo y si lo consigue invoca al subprograma cargar del módulo de la lista de partidas:

```
bool cargar(tListaPartidas & partidas, ifstream & archivo)
```

A este subprograma hay que pasarle como parámetro de E/S el flujo de entrada que hemos obtenido al abrir el fichero. Ahora, este subprograma, una vez leído el número de partidas que hay en el fichero, invoca, para cada partida, al subprograma que gestiona la carga de una partida que está en el módulo partida:

```
bool cargar(tPartida & partida, ifstream & archivo)
```

De nuevo a este subprograma hay que pasarle como parámetro de E/S el flujo de entrada que recibió el cargar del módulo lista de partidas. Ahora, este subprograma lee los campos básicos de la partida en orden (identificador, ...), sin embargo, cuando llegue el turno de leer el estado del juego tendrá que llamar al subprograma cargar del módulo del juego conecta 4:

```
void cargar(tConecta4 & conect4, ifstream & archivo)
```

De nuevo a este subprograma hay que pasarle como parámetro de E/S el flujo de entrada que recibió el cargar del módulo partida.

### **Indicaciones importantes**

- Las listas de tamaño variable deberán implementarse con la estructura de array con contador vista en clase.
- Siempre que se pueda, las operaciones de búsqueda serán binarias.
- Todos los módulos (excepto el principal) se implementarán con archivos .h y .cpp. En el archivo .h se incluirán los tipos de datos públicos del módulo y los prototipos de las funciones públicas con comentarios describiendo qué hace cada función, indicando parámetros de entrada, salida y entrada/salida.
- Los módulos ListaUsuarios, ListaPartidas y ListaAccesoPartidas únicamente realizan operaciones con listas, y nunca deben escribir nada en pantalla. Pueden devolver valores al programa que los llame para que éste escriba en pantalla lo que sea necesario.

## **3. Entrega de la práctica**

La práctica se entregará a través del Campus Virtual, en la tarea **Entrega de la Práctica 4** donde debes subir el archivo practica4.zip con todos los archivos .h y .cpp del proyecto.

Asegúrate de poner el nombre de los miembros del grupo en un comentario al principio de cada archivo. Fin del plazo de entrega: **8 de mayo a las 23:55**.