El Solitario 2.0

Fundamentos de la Programación II

Grados de la Facultad de Informática (UCM)

Normas de realización de la práctica

1. La fecha límite para entregar la práctica es el **26 de abril a las 22:00**.
2. Debe entregarse a través del Campus Virtual, en la actividad Entrega práctica (versión 2).
3. Escribe tu nombre completo en un comentario al inicio de cada fichero de código que entregues.
4. Esta práctica es una extensión de la versión 1. Corrige antes los errores indicados que cometiste en esa primera versión.
5. En el Campus Virtual encontrarás el fichero `memoryleaks.h` que ayuda en la detección de pérdidas de memoria (*memory leaks*). Deberás incluirlo en todos tus ficheros que hagan gestión de memoria dinámica. También encontrarás un fichero de datos `partidas.txt` para ser cargado por tu programa al comenzar a ejecutarse.

Primero resuelve el problema. Entonces, escribe el código.

— John Johnson

Comentar el código es como limpiar el cuarto de baño;
nadie quiere hacerlo, pero el resultado es siempre
una experiencia más agradable para uno mismo y sus invitados.

— Ryan Campbell

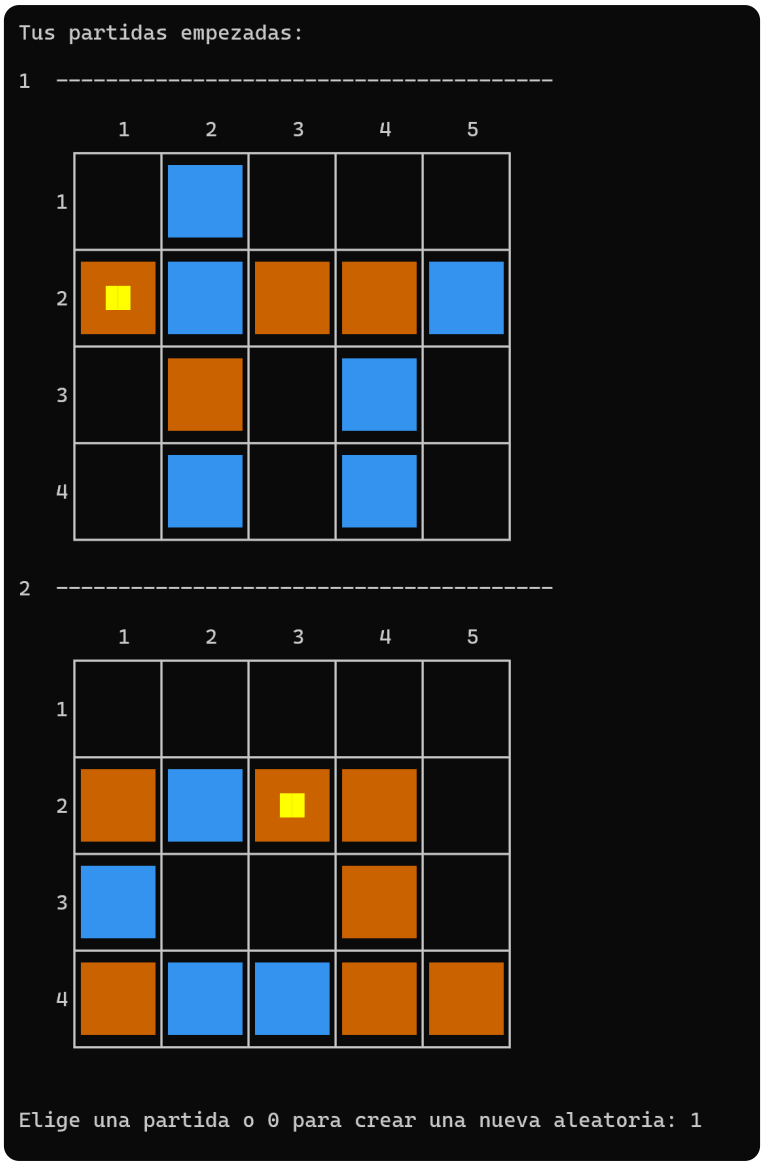
Objetivo

En esta segunda versión de la práctica vamos a extender la primera versión añadiendo un *gestor de partidas* de diferentes usuarios. Una partida es una instancia del juego *Solitario* con la que posiblemente ya se ha comenzado a jugar.

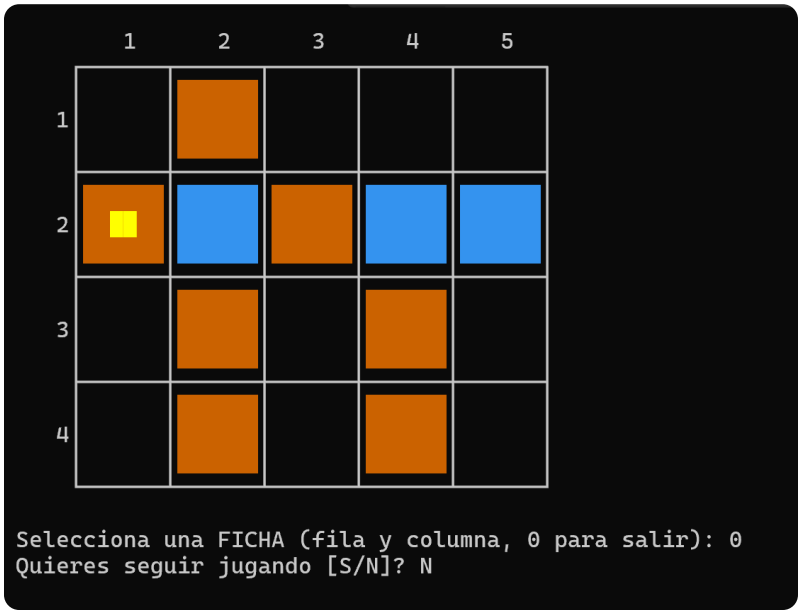
La aplicación comenzará solicitando al usuario su identificador, con el cual hará *login* en la aplicación:

```
Usuario (FIN para terminar): 123E
```

Se recuperarán sus partidas comenzadas pero no terminadas y se le mostrará el estado actual de esas partidas, solicitándole que elija una de ellas para continuar la partida o podrá elegir que se cree una nueva partida con un tablero aleatorio (del que podrá elegir el número de movimientos para resolverlo).



Con una partida ya seleccionada el usuario podrá jugar con ella de la misma manera que se hacía en la primera versión de la práctica, con la diferencia de que el usuario podrá abandonar la partida en cualquier momento, si esta no se ha terminado.



Si abandona la partida, se le preguntará si desea seguir jugando o no.

- Si contesta que sí, se le volverá a mostrar sus partidas comenzadas, con la última a la que jugó correspondientemente actualizada, para que elija continuar una partida o crear una nueva (como cuando hizo *login*).
- Si contesta que no, será como si hiciera *logout*, y se mostrará de nuevo el prompt de *login* para que un nuevo usuario pueda acceder a la aplicación.

Si durante el transcurso del juego el usuario llega a terminar una partida, con ganador o bloqueo, se le mostrará el resultado de la partida y se le preguntará si quiere seguir jugando, con un comportamiento análogo al del párrafo anterior, con la diferencia de que la partida terminada ya no se mostrará entre las partidas del usuario en el caso de que quiera seguir jugando.

Si un usuario sin partidas comenzadas hace *login* (no existía para la aplicación), se le ofrecerá directamente la opción de comenzar con un tablero aleatorio:

```
Usuario (FIN para terminar): 456M

No tienes partidas. Vamos a crear un juego aleatorio
Indica el numero de pasos para crear el tablero aleatorio:
```

Lo mismo ocurrirá si un usuario que está dentro de la aplicación termina todas sus partidas comenzadas y quiere seguir jugando.

Si un usuario deja de seguir jugando, es decir, hace *logout*, y no tiene partidas comenzadas, entonces su información será borrada de la aplicación.

Si en el prompt del *login* se escribe `FIN` entonces el programa terminará, guardando en el fichero `partidas.txt` la información de los usuarios que han utilizado la aplicación y tienen partidas comenzadas. La información de ese fichero será cargada la siguiente vez que se ejecute el programa.

Se os ha proporcionado un fichero inicial `partidas.txt` con la información de algunos usuarios. El formato de este fichero se describe más adelante en la operación `cargar` del módulo `GestorPartidas`.

Implementación

Para el diseño modular de esta nueva funcionalidad del programa vamos a añadir los módulos que se describen a continuación.

Módulo `Lista`

Este módulo sirve para representar listas genéricas de valores. Estas listas serán implementadas mediante arrays dinámicos cuyo tamaño pueda crecer si el array actual está ya lleno y queremos insertar un elemento más.

El tipo de datos `Lista` consta al menos de las siguientes operaciones:

```
template <typename T>
class Lista {
public:
   Lista();   // constructor lista vacía
   ~Lista();  // destructora
   T & operator[](int i);
   T const& operator[](int i) const;
   void push_back(T elem);
   void pop_back();
   int size() const;
   bool empty() const;
private:
   T* datos;
   int num_elems;
   int capacidad;
};
```

El comportamiento de las operaciones es el siguiente:

- `Lista()` es el constructor que crea la lista vacía, sin elementos.
- `~Lista()` es la destructora que libera la memoria utilizada por el array dinámico.
- `operator[](i)` devuelve el elemento en la posición `i` de la lista, si existe. Hay dos versiones de este operador: una constante que devuelve el valor de la posición `i` por referencia constante, y otra no constante que devuelve el valor por referencia, lo que permite que ese valor pueda ser modificado.
- `push_back(elem)` añade el elemento `elem` al final de la lista, duplicando el tamaño del array dinámico si es necesario (no había espacio libre en el array para el nuevo elemento).
- `pop_back()` elimina el último elemento de la lista, si la lista no está vacía.
- `size()` devuelve el número de elementos en la lista.
- `empty()` devuelve cierto si la lista está vacía y falso en caso contrario.

El atributo `datos` es un puntero que apunta a la primera posición del array en memoria dinámica utilizado para almacenar los elementos de la lista. El tamaño actual de ese array se guarda en el atributo `capacidad` y el número de posiciones realmente ocupadas (número de elementos actualmente en la lista) se guarda en el atributo `num_elems`. Siempre se cumplirá que `num_elems` $\leq$ `capacidad`.

Módulo `GestorPartidas`

Es el módulo más importante en esta extensión de la práctica. Sirve para almacenar la información de los usuarios conocidos, asociando a cada uno de ellos la lista de sus partidas comenzadas. En concreto mantendrá una lista de pares ⟨ Usuario, Lista de partidas ⟩, ordenada de menor a mayor por usuario, para que las búsquedas por usuario sean eficientes. Para evitar copias de objetos, en particular copias de objetos de la clase `Juego`, hace un uso intensivo de punteros.

Este módulo define `Usuario` como un alias del tipo `std::string` (los usuarios podrían ser más complicados, teniendo asociada por ejemplo una contraseña que hubiera que validar para entrar en la aplicación, pero para simplificar por ahora serán solamente un identificador).

```
using Usuario = std::string;
```

También define el tipo de datos `GestorPartidas`, que consta al menos de las siguientes operaciones:

```
class GestorPartidas {
public:
   GestorPartidas();  // crea un gestor vacío
   ~GestorPartidas(); // destructora
   bool cargar(std::istream&/*ent/sal*/ entrada);
   bool salvar(std::ostream&/*ent/sal*/ salida);
   void login(Usuario const& usuario);
   void logout();
   bool tiene_partidas() const;
   void mostrar_partidas() const;
   int insertar_aleatoria(int movimientos);
   Juego & partida(int part);
   void eliminar_partida(int part);
private:
   static const int NOLOGIN = -1;
   int usuario_activo; // posición del usuario que ha hecho login
   struct UsuarioPartidas {
      Usuario user;
      Lista<Juego*> partidas;
   };
   Lista<UsuarioPartidas*> usuarios;
};
```

El comportamiento de las operaciones es el siguiente:

- `GestorPartidas()` es el constructor que crea una lista de pares vacía.

- `~GestorPartidas()` es la destructora, que tiene que liberar todos los objetos creados en memoria dinámica por métodos de esta clase (objetos de la clase `Juego` y del registro `UsuarioPartidas`).
- `cargar(entrada)` rellena la lista de pares con la información en el flujo de entrada recibido como argumento. El flujo corresponderá al fichero `partidas.txt` abierto. El formato de este fichero es el siguiente: primero aparece un número con el número de usuarios cuya información está almacenada en el fichero, y a continuación aparecen tantas descripciones de usuarios como indique ese número. Cada descripción de usuario comienza con el identificador de usuario (un valor de tipo `Usuario`) en una línea. En la línea siguiente aparece un número que representa el número de partidas comenzadas de este usuario. Y en las siguientes líneas aparecen las descripciones de estas partidas, con el formato de los archivos para los juegos de la primera versión de la práctica (número de filas y columnas, contenido de cada celda del tablero, y posición de la meta). Se puede suponer que en el fichero los usuarios aparecen en orden de menor a mayor identificador.
- `salvar(salida)` guarda en el flujo de salida recibido como argumento (correspondiente al fichero `partidas.txt`) la información de la lista de pares con el formato descrito en el método `cargar`.
- `login(usuario)` registra que el usuario `usuario` ha hecho *login* en el sistema, actualizando el valor del atributo `usuario_activo`, que contendrá *la posición* del usuario que ha hecho login en la lista de pares, para no tener que estar buscándolo cada vez. Si el usuario no existe en la lista, se añadirá en la posición que le corresponda según el orden entre usuarios, asociándole una lista de partidas vacía.
- `logout()` registra el hecho de que el usuario activo ha salido del sistema, actualizando el valor del atributo `usuario_activo` a la constante `NOLOGIN`. Además, si el usuario activo no tenía partidas comenzadas, su información es borrada de la lista de pares.
- `tiene_partidas()` devuelve cierto si el usuario activo tiene partidas comenzadas y falso en caso contrario.
- `mostrar_partidas()` muestra por la consola las partidas comenzadas del usuario activo numerándolas desde `1`.
- `insertar_aleatoria(movimientos)` inserta en la lista de partidas asociada al usuario activo un nuevo `Juego` creado con la constructora de juegos aleatorios con argumento `movimientos`. Devuelve la posición de ese nuevo `Juego` en la lista de partidas.
- `partida(part)` devuelve por referencia el objeto `Juego` correspondiente a la partida con índice `part`. Aunque al usuario se le muestren las partidas numeradas desde `1`, internamente estas estarán numeradas desde `0`. Al devolverse el `Juego` por referencia, las modificaciones que se realicen en el objeto al jugar se realizarán directamente sobre un objeto de la lista de partidas (en realidad un objeto dinámico apuntado por una posición en la lista, al ser los valores de esta lista punteros a juegos).
- `eliminar_partida(part)` elimina de la lista de partidas asociada al usuario activo la partida con índice `part` porque será una partida que se ha terminado.

En cuanto a los atributos privados, `usuario_activo` almacena la posición en la lista de pares del último usuario que ha hecho *login* y aún no ha hecho *logout*, si hay alguno. Si no, este atributo valdrá `NOLOGIN`. Y la lista de pares se almacena en el atributo `usuarios` que es una lista de punteros a registros del tipo `UsuarioPartidas`. Al utilizar punteros no se copiarán

valores de tipo `UsuarioPartidas` cuando haya que hacer desplazamientos porque llegue un nuevo usuario o se elimine a un usuario. El registro `UsuarioPartidas`, que representa los pares ⟨ Usuario, Lista de partidas ⟩, tiene dos campos: el usuario y una lista de punteros a objetos de la clase `Juego`. De nuevo se utilizan punteros para no realizar copias cuando se creen o eliminen partidas de la lista.

Módulos `Tablero`, `Movimiento` y `Juego`

Estos módulos prácticamente no se modifican respecto a la primera versión de la práctica. Si no tenías un método `salvar` en `Juego` tendrás que añadirlo, para que pueda ser utilizado desde el método `salvar` del gestor.

Módulo `Main`

Este módulo es el encargado del control de la interacción de los usuarios con el sistema. Tendrá un bucle principal que permitirá que los usuarios hagan *login* y *logout*. Cuando un usuario entre en el sistema, se entrará en otro bucle hasta que el usuario desee dejar de jugar. En este bucle interno, primero se mostrarán las partidas comenzadas del usuario y se le dará la opción de elegir una de ellas o una nueva creada aleatoriamente, para a continuación empezar a jugar, como se hacía en la primera versión de la práctica. La solicitud de datos al usuario tendrá que ser modificada para permitir que el usuario pueda abandonar una partida antes de que esta termine, como se mostró en las capturas al comienzo de este enunciado.

Ten especial cuidado con el uso de memoria dinámica y en particular con las *pérdidas de memoria*. Para que el sistema de depuración de Visual Studio te ayude con esto, incluye en todos los ficheros que hagan uso de memoria dinámica el fichero `memoryleaks.h`, y como primera instrucción de la función `main()` añade esta:

```
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

que hará que al terminar el programa se vuelque en la ventana de salida de Visual Studio información sobre las pérdidas de memoria, en caso de que las haya habido.