Práctica 3
Uno Solo

Fecha de entrega: 28 de marzo

Fuente: Wikimedia.org

1. Historia y breve descripción del juego

El solitario conocido como Uno Solo o Senku es un juego de tablero abstracto en el que normalmente se juega con un tablero con agujeros y clavijas o bolas que se colocan en dichos agujeros.

Según Wikipedia, la primera evidencia del juego data de la corte de Luis XIV, según el grabado hecho en 1687 por Claude Auguste Berey donde aparece Anne de Rohan-Chabot, Princesa de Soubise, jugando a dicho solitario. La edición de agosto de 1687 de la revista literaria francesa "Mercure galant" contenía una descripción del tablero, de las reglas y de problemas de ejemplo.

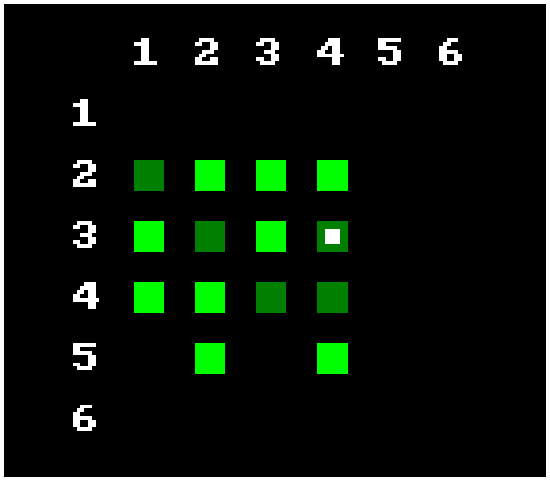
En este solitario, se empieza con un tablero en modo de rejilla con piezas colocadas sobre las casillas. El jugador debe mover una pieza en cada turno. Las piezas sólo pueden moverse "saltando" sobre otra ficha hasta una casilla libre de forma que la pieza sobre la que se salta es eliminada del tablero, como en las damas. Sólo se puede saltar una pieza, y el salto debe ser en horizontal o en vertical, nunca en diagonal. El objetivo del juego es eliminar todas las piezas, dejando sólo una en el tablero. Aunque nosotros implementaremos una variante de este solitario en la que la última pieza debe acabar en una casilla específica.

Existen muchas variantes de este juego que parten de formas concretas del tablero (como por ejemplo el tablero en cruz que muestra la primera imagen). Nuestro tablero podrá tener formas más libres como veremos más adelante.

2. El programa

El programa simulará de forma realista la dinámica del juego, aunque obviamente en modo consola. Utilizará caracteres para dibujar el tablero, así como colores de fondo para las fichas.

Principalmente el programa usa un array bidimensional de tamaño fijo `DIM` x `DIM` celdas para mantener el estado del tablero (en los ejemplos de este enunciado `DIM=6`). Las celdas pueden tener valor `NULA`, `VACIA` o `FICHA`, que representan respectivamente el estado de una celda del tablero que no se puede utilizar en esta partida (es decir, no se pueden colocar fichas sobre él), una celda vacía y una celda en la que hay una ficha.



La imagen anterior representa un tablero donde las celdas en negro son nulas (no forman parte del tablero "jugable" en esta partida), las que están en verde apagado están vacías y las que están en verde brillante son celdas con ficha. La celda que tiene el símbolo en blanco en el centro representa la meta de esta partida.

Tal y como muestra la imagen anterior en cada turno el jugador deberá introducir la ficha que quiere mover introduciendo **primero la fila y luego la columna**. Si la fila y columna corresponden a una celda con ficha que se puede mover, el programa preguntará en qué

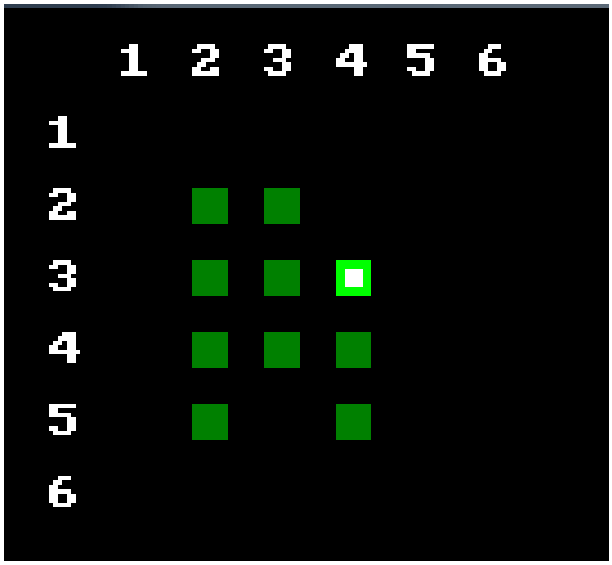
dirección quiere mover la ficha indicándole **sólo aquellas opciones posibles** para dicha ficha. De forma general el jugador introducirá

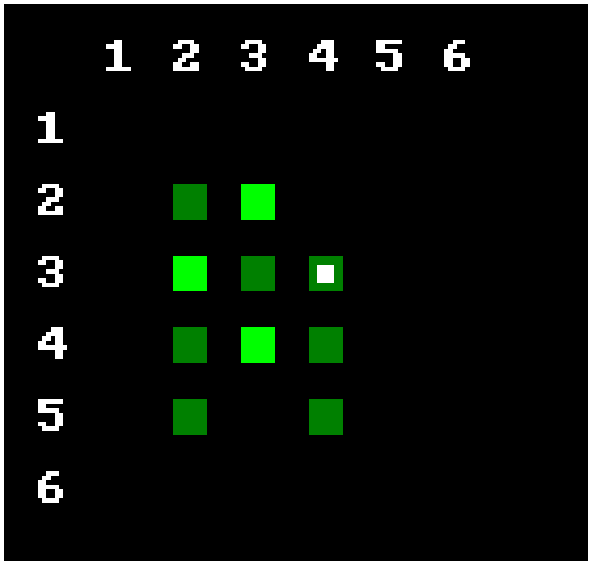
- 1 para mover Arriba
- 2 para mover a la Derecha
- 3 para mover Abajo
- 4 para mover a la Izquierda
- 0 si ha decidido que no quiere mover esa ficha y quiere seleccionar otra

Si la celda seleccionada por el jugador no es válida el juego indicará **la razón por la cual no es válida**. A saber:

- Se trata de una **posición fuera del tablero**. Por ejemplo, cualquier celda que incluya la fila/columna 0 (o menor) o la fila/columna 7 (o mayor) en la imagen anterior.
- Se trata de una **celda sin ficha**, ya sea porque está vacía o porque es una celda nula del tablero. Por ejemplo, las celdas (1,1) o (2,1) de la imagen anterior.
- Se trata de una celda con una **ficha que no se puede mover**. Por ejemplo, la celda (2,4) o la celda (4,2) en la imagen anterior.

Además, el juego deberá informar si se llega a un punto en el que sobre el tablero queda una única ficha que está situada en la celda de meta (en ese caso el jugador habrá ganado) o en el que no se pueden mover más fichas (en ese caso el jugador habrá perdido). Las siguientes imágenes ilustran ambos casos.





3. Versión 1: Visualización, carga y guardado del juego

La primera versión del programa se encargará de mostrar la partida, y cargar y guardarla en un fichero de texto. En este primer momento no nos interesará tener una versión funcionando del juego, sino tomar contacto con la práctica.

El juego lo representaremos mediante un tipo de datos `tJuego` que será un tipo de datos estructurado con campos que representen lo siguiente:

- El tablero, que describiremos más adelante.
- La fila y la columna en la que se encuentra la celda de meta.
- El número de bolas iniciales que tenía el tablero.
- El número de movimientos realizados en la partida (que es igual al número de bolas eliminadas en el tablero).
- El estado del juego, que representaremos mediante un enumerado con los posibles estados del juego, a saber: `BLOQUEO, GANA, JUGANDO, ABANDONO`.

Por su parte el tablero será una matriz bidimensional cuadrada de un tipo `tCelda` y de una dimensión constante `DIM`. Los valores del enumerado serán `NULA, VACIA y FICHA.` Aprovechando las posibilidades de los enumerados en C++, asignaremos los valores enteros `0, 2 y 10` (respectivamente) a cada uno de los valores mencionados. Estos valores los usaremos para pintar el tablero como veremos más adelante.

3.1. Visualización del juego

A la hora de visualizar el tablero ten en cuenta que cada celda usa un espacio con un color de fondo que depende del valor de la celda. Salvo la meta, que usa el carácter `char(254)` en color blanco y el fondo será el correspondiente a celda vacía o con ficha.

Al dibujar el tablero, obviamente por filas, ten en cuenta que las celdas de una fila están separadas por un espacio en blanco, y entre filas se deja una línea. Cada fila empieza por el número de fila (índice + 1).

Sobre el tablero se mostrará también el número movimientos realizados y los números de las columnas (índice + 1) del tablero.

Colores de fondo en la consola

Por defecto, el color de primer plano, aquel con el que se muestran los trazos de los caracteres, es blanco, mientras que el color de fondo es negro. Podemos cambiar esos colores, por supuesto, aunque debemos hacerlo utilizando rutinas que son específicas de Windows, por lo que debemos ser conscientes de que el programa no será portable a otros sistemas.

En nuestro programa el color de primer plano siempre es blanco, pero queremos poder cambiar el color de fondo para representar cada celda con un color diferente.

La biblioteca `Windows.h` que incluye subprogramas para manejar la consola. Uno de ellos es `SetConsoleTextAttribute()`, que permite ajustar los colores de fondo y primer plano. Incluye en el programa esa biblioteca y este procedimiento:

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

Basta proporcionar un color para el fondo (0 a 15) y este procedimiento lo establecerá, con el color de primer plano siempre en blanco (15).

El entero color que le pases puede ser el valor del enumerado `tCelda` equivalente. El color negro se representa con un 0, el verde apagado con el 2 y el verde brillante con el 10. Estos tres valores corresponderán los valores del enumerado (`NULA, VACIA y FICHA`) como se ha explicado anteriormente.

3.2. Carga y guardado del juego

También queremos poder cargar y guardar el juego en mitad de una partida. Como podemos cambiar la dimensión del tablero para jugar con tableros más grandes o más pequeños (de 3x3 a 10x10), los archivos de tableros deben comenzar con una línea que indique cuál es esa dimensión (`DIM`). En el caso de la carga, el tablero sólo se cargará si esa dimensión coincide con la dimensión actual de tablero en el programa. A continuación se guarda el número de bolas iniciales del tablero, y la fila y la columna de la meta.

En la carga y en el guardado del tablero se procesarán `DIMxDIM` enteros. El fichero tiene cada fila del tablero dispuesta en una línea, y dentro de cada línea los valores están separados por tabuladores.

Tras el tablero se guardarán el número de movimientos realizados hasta ese momento en la partida. El archivo termina con esos puntos; no hay centinela (no se necesita).

A continuación mostramos un ejemplo del fichero.

```
6   ← Dimensión
13  ← Nº de bolas iniciales
2   ← Fila de la meta
3   ← Columna de la meta
0    0    0    0    0    0
2    10   10   10   0    0
10   10   0    10   0    0      } Tablero cuadrado
2    10   2    2    0    0      } de DIM x DIM
10   2    10   10   10   10
0    0    0    10   0    0
0   ← Nº de movimientos
```

3.3. Implementación

Como hemos dicho anteriormente, esta primera versión del programa nos debe servir únicamente para probar que hemos implementado correctamente los subprogramas que nos permiten cargar, visualizar y guardar un tablero.

Por ello, la función `main()` tendrá el código necesario para comprobar que estos subprogramas funcionan correctamente. Por ejemplo: puedes inicializar un juego con un tablero inicializado con todas las celdas a nulo y colocar en él unas cuantas celdas vacías y con fichas donde quieras. A continuación puedes probar a mostrar la partida, luego guardarla en el fichero, y cargar a continuación dicho fichero en otra variable de tipo juego y mostrarla a modo de validación.

Deberás implementar, al menos, los siguientes subprogramas:

- ✓ `void mostrar(const tJuego &juego)`: Muestra el número de movimientos realizados y el tablero con la partida.
- ✓ `bool cargar(tJuego &juego, string nombre)`: Intenta cargar desde el fichero llamado `nombre` el juego completo según el formato especificado anteriormente. Si no se puede cargar el archivo, devuelve false, y true en caso contrario.
- ✓ `void guardar(const tJuego &juego, string nombre)`: Guarda en el fichero `nombre` el juego que recibe como parámetro según el formato especificado anteriormente.

Mi primera implementación modular

En esta práctica vamos a tener una primera toma de contacto con los módulos.

Para ello vamos a dividir nuestra implementación de esta práctica en tres ficheros:

- `unosolo.h`: Contendrá los tipos de datos que necesitemos para el juego y las librerías necesarias para definir esos tipos de datos, así como los prototipos de los subprogramas que puede usar `main`. Los ficheros .h se denominan de cabecera (o *header* en inglés).
- `unosolo.cpp`: Contendrá las implementaciones de los subprogramas cuyos prototipos estén definidos en `unosolo.h`. Los ficheros .cpp (código fuente C++) son los archivos que implementan los prototipos del correspondiente archivo de cabecera. Tendremos un fichero .cpp por cada archivo de cabecera, cuya primera línea debe incluir el fichero .h correspondiente. Es decir, el archivo unosolo.cpp debe comenzar con la directiva `#include "unosolo.h"`. A continuación incluirá las librerías (por ejemplo, iomanip o windows.h), las declaraciones de constantes, tipos y prototipos necesarios para implementar los subprogramas declarados en la cabecera, y las implementaciones de todos los prototipos.
- `main.cpp`: Archivo del proyecto en el que se define la función `main` (punto de inicio de la ejecución de la aplicación, sólo puede haber una). En este caso deberá incluir mediante la directiva `#include` el archivo de cabecera `unosolo.h`, para que las declaraciones realizadas `unosolo.h` sean visibles, es decir, se puedan invocar. Además, este fichero también deberá incluir todas las librerías que precise el código del `main` para funcionar correctamente. En general sólo contienen la implementación de la función `main`, pero se puede añadir alguna función auxiliar, referente a la interacción con el usuario.

Crea un proyecto vacío y añade dos archivos de código C++ (`main.cpp` y `unosolo.cpp`) y uno de cabecera (`unosolo.h`). Ya puedes empezar a escribir:

```
// archivo unosolo.h

#include <string>
…
// constantes y tipos
…
typedef strcut {…} tJuego;

// prototipos

bool cargar(…);
void guardar(…);
void mostrar(…);
…
```

```
// archivo unosolo.cpp

#include "unosolo.h"
#include <iostream>
…
// constantes y tipos
…
// prototipos
…
// implementación de los prototipos
// declarados en unosolo.h y aquí

void mostrar(…){ … }
…
```

```
// archivo main.cpp
// autores

#include <iostream>
…
#include "unosolo.h"


int main() {
  tJuego solo;
  …
    …
  return 0;
}
…
```

4. Versión 2: Implementando el juego

Vamos a añadir a la versión 1 la posibilidad de jugar partidas **sucesivas** al Uno Solo. Sin embargo, eso sólo será posible al final de la versión 2. Debemos empezar primero por implementar la mecánica de movimiento de nuestro solitario.

4.1. La mecánica del juego

Suponiendo un tablero cargado de fichero (o inicializado a mano en el código para hacer pruebas), el usuario deberá seleccionar una ficha del tablero especificando su fila y columna. Si la ficha no es válida, se le indicará por qué, y si es válida, se le indicarán los movimientos posibles. Una vez seleccionado un movimiento posible, se realizará el movimiento en el tablero y se volverá a mostrar el mismo. Los detalles de este funcionamiento ya se han explicado en la sección 2 de este enunciado.

Para gestionar los movimientos (jugadas), definiremos en el archivo de cabecera un nuevo tipo de datos `tMovimiento`, que será un estructurado con campos que representen lo siguiente:

- La fila y la columna origen del movimiento. Hay que tener en cuenta que la numeración mostrada al usuario (de 1 a `DIM`) no corresponde con la interna del array (de 0 a `DIM-1`).
- Las direcciones en las que sí se puede realizar un movimiento según la situación del tablero. Como sólo hay cuatro direcciones posibles, usaremos un array de booleanos de `NUM_DIRS` (4) posiciones (`tPosibles`) que representarán los movimientos en el siguiente orden: arriba, abajo, izquierda y derecha.
- La dirección escogida para el movimiento, que representaremos mediante un enumerado con los siguientes valores: `ARRIBA, ABAJO, IZQUIERDA, DERECHA, INCORRECTA`.

Fíjate que por la representación interna de los enumerados como enteros, `ARRIBA` corresponde al valor 0, `ABAJO` al valor 1, y así sucesivamente. Esto hace que si tenemos una variable `tPosibles posibles` (array de booleanos), podemos acceder a ellos de la siguiente manera: el valor `posibles[ARRIBA]` corresponderá al primer booleano del array (en la posición 0), `posibles[ABAJO]` al segundo (en la posición 1), y así sucesivamente.

Con toda esta información, el movimiento de una ficha consistirá en los siguientes pasos:

1. Leer el movimiento especificado por el usuario:
   - Fila y columna de una ficha del tablero con posibles movimientos (o 0 para salir)
   - Dirección del movimiento (una de entre las posibles o 0 para cambiar de ficha).
2. Ejecutar el movimiento:
   - Actualizar las fichas del tablero.
   - Incrementar el número de movimientos realizados en el juego.
   - Recalcular el estado del juego según el movimiento: `BLOQUEO, GANA, JUGANDO, ABANDONO`. Este paso requiere comprobar varias cosas:
     - Si el usuario ha ganado la partida, es decir, si sólo queda una bola en el tablero y ésta está colocada en la celda de meta. En ese caso el estado resultante será `GANA`.
     - Además requiere comprobar si todavía queda al menos una ficha con movimientos posibles, el estado será `JUGANDO`.
     - Si no se cumplen ninguna de las anteriores (no se ha ganado y no hay movimientos para ninguna ficha), el estado será `BLOQUEO`.

Se da la posibilidad al usuario de que en el paso 1 introduzca simplemente un 0 si quiere salir. En ese caso se generará un movimiento con dirección `INCORRECTA`, y en el paso 2 simplemente modificará el estado del juego a `ABANDONO`. Además, cuando el usuario decide salir el programa le preguntará si desea salvar la partida, si es así, se deberá invocar al subprograma `guardar` que implementamos en la versión anterior.

4.3. Implementación

Para llevar a cabo esta versión deberás implementar, al menos, los siguientes subprogramas:

- ✓ `void partida(tJuego &juego)`: Dado un juego ya inicializado, realiza el bucle del movimiento de la ficha explicado anteriormente. En cada vuelta del bucle mostrará el estado del tablero, y terminará cuando el estado del tablero deje de ser `JUGANDO`. Usará los siguientes subprogramas.
- ✓ `bool leerMovimiento(const tTablero tablero, tMovimiento &mov)`: Dado un tablero, lee un movimiento válido y lo devuelve. Para ello, pregunta al usuario qué ficha quiere mover, muestra los movimientos posibles, y guarda la información dada por el usuario en `mov`. Si el usuario desea salir se devuelve `false`, y si se ha seleccionado un movimiento válido devuelve `true`.
- ✓ `void ejecutarMovimiento(tTablero tablero, const tMovimiento &mov)`: Dado un tablero y un movimiento válido, se ejecuta el movimiento y se devuelve el tablero debidamente modificado.
- ✓ `void nuevoEstado(tJuego &juego)`: Dado un juego, calcula el nuevo estado del mismo de acuerdo a lo especificado más arriba.

4.2. Jugando varias partidas

Una partida consiste en la ejecución de la secuencia de pasos descrita en el apartado anterior hasta que el juego alcanza uno de los estados finales (`BLOQUEO, GANA o ABANDONO`).

Un vez hayamos conseguido eso debemos permitir que nuestro programa juegue tantas partidas como el usuario desee. Seguirá la siguiente secuencia:

1. Se preguntará al usuario si quiere jugar una nueva partida o abandonar el juego.
2. En caso de que quiera jugar le debe permitir elegir cargar el tablero que desee. Si la carga se ha realizado con éxito, la partida comienza.
3. La partida proseguirá hasta que se alcance uno de los estados finales. Cuando eso suceda se volverá al punto 1.

5. Versión 3: Generación aleatoria de tableros

Vamos a añadir a nuestro juego la posibilidad de poder generar tableros de manera aleatoria. En realidad, la generación aleatoria de tableros no es más que jugar al juego pero "al revés": partiendo de un tablero con todas las posiciones nulas excepto una única ficha que está en la meta, se irán generando **movimientos inversos** válidos que irán creando nuevas fichas en lugar de hacerlas desaparecer.

En nuestro programa se preguntará al usuario cuántos movimientos inversos van a realizarse para generar el tablero aleatorio (a más movimientos más complejo será resolver el tablero), y se intentará generar un tablero con ese número de movimientos inversos.

Las siguientes figuras representan la generación de un tablero mediante los dos primeros movimientos inversos:

La generación de tableros que vamos a implementar consiste en los siguientes pasos:

1. Inicialización del tablero con todas las posiciones nulas.
2. Fijar la meta en una posición aleatoria, y colocar una ficha en esa posición (ver primera figura).
3. Mientras no se llegue al número de movimientos indicado por el usuario, se elegirá al azar una de las fichas, se buscará un movimiento inverso válido (si hay más de uno se elegirá también al azar), y se realizará el movimiento. Se considera que un movimiento inverso es válido si las dos siguientes celdas en esa dirección están vacías o nulas.

En las figuras, se puede ver que la meta elegida aleatoriamente se encuentra en la posición (3,4). Después, el primer movimiento inverso será usando una ficha aleatoria (que será la única existente, la ficha inicial) y con una dirección aleatoria (en este caso hacia abajo). Así, se colocan dos fichas en las siguientes posiciones en esa dirección, y la posición anteriormente ocupada por la ficha se indica como `VACIA` (no como `NULA`). Para el segundo movimiento inverso se elige de nuevo una ficha aleatoriamente, en el ejemplo (5,4), y una dirección aleatoria (en este caso sólo son válidos los movimientos hacia la derecha y hacia la izquierda). Igual que antes, se deja como `VACIA` la posición ocupada por la ficha, y se colocan fichas en las dos siguientes posiciones en esa dirección.

IMPORTANTE: Aunque el usuario elija un número M de movimientos aleatorios para generar el tablero, puede ocurrir que se llegue a un punto en que, aunque no se hayan hecho todos los movimientos pedidos por el usuario, se tenga que detener la generación porque no se pueda realizar ningún movimiento inverso válido. Para no complicar la implementación, se considerará que si se elige aleatoriamente una ficha para hacer un movimiento inverso y ésta no tiene ningún movimiento válido, se detendrá la generación del tablero. En la parte opcional de la práctica os pediremos implementar una versión más sofisticada del criterio de parada.

5.1. Implementación

Deberás implementar, al menos, los siguientes subprogramas:

- ✓ `void generarTablero(tJuego &juego, int movimientos):` Se intenta generar un tablero con el número de movimientos dado. Si al elegir una ficha para generar un movimiento no tiene movimientos posibles, se parará la generación del tablero. Aunque al usuario no le interesa saber cómo se ha generado el tablero (¡es mostrar cómo se resuelve!), a nosotros sí, ya que nos permite ver que se está generando correctamente. Por ello, este subprograma deberá mostrar cada uno de los pasos (movimientos inversos) de la generación del tablero. Este subprograma usará los siguientes subprogramas.
- ✓ `void iniciarTablero(tTablero tablero):` Inicializa el tablero con todas las casillas al valor `NULA`.
- ✓ `void fijarMeta(tJuego &juego):` Se elige aleatoriamente una meta en el tablero y se coloca una ficha en ella.
- ✓ `bool movimientoInverso(tJuego &juego):` Intenta realizar un movimiento inverso seleccionando una ficha y eligiendo entre uno de sus posibles movimientos. Si ha podido realizar un movimiento devuelve `true`, y en caso contrario devuelve `false`.

Además, te vamos a pedir que no borres al generar el tablero la consola para que muestre toda la secuencia de pasos, es decir, que no se limpie la consola y se vuelva a escribir. De esa manera, podrás (podremos) comprobar si la generación es correcta.

6. Partes opcionales

6.1. Nuevo criterio de parada en la generación aleatoria de tableros

Como ya se ha visto, para simplificar la generación de los tableros la generación se detiene cuando se escoge una ficha sin movimientos inversos posibles. Sin embargo, podría haber otras fichas con movimientos posibles con las que se podría seguir generando el tablero.

Se propone que se modifique la práctica para que la generación aleatoria de tableros sólo termine cuando se llegue al número de movimientos indicados por el usuario, o cuando no haya ninguna ficha que tenga movimientos inversos posibles.

6.2. Deshacer movimientos

Una funcionalidad útil en el juego sería poder deshacer los N últimos movimientos realizados. Para ello, en `tJuego` se deberán guardar los N últimos movimientos realizados en una lista de tamaño variable.

Entrega de la práctica

La práctica se entregará a través del Campus Virtual en una nueva tarea **Entrega de la Práctica 3** que permitirá subir un zip con los tres archivos de la práctica (`main.cpp`, `unosolo.h`, y `unosolo.cpp`). También se pueden incluir archivos con tableros de prueba. Asegúrate de poner el nombre de los miembros del grupo en un comentario al principio de cada archivo.

El fichero subido deberá tener el siguiente nombre: Practica3GrupoXX.zip, siendo XX el número de grupo. Sólo uno de los miembros del grupo (no los dos) será el encargado de subir la práctica.

Fin del plazo de entrega: **28 de marzo a las 23:55**