El Solitario 1.0

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 **21 de marzo a las 22:00**.
2. Debe entregarse a través del Campus Virtual, en la actividad Entrega práctica (versión 1).
3. Sigue los pasos que aparecen en el enunciado y ten en cuenta todas las indicaciones.
4. Aunque recomendamos que vayas haciendo la práctica de manera incremental, tal y como aparece en el enunciado, debes leerlo completamente antes de empezar.
5. En el Campus Virtual encontrarás un archivo `zip` con ficheros que puedes utilizar para ayudarte a dibujar los tableros, además de algunos tableros de ejemplo.

Primero resuelve el problema. Entonces, escribe el código.

— John Johnson

El juego

En esta práctica vamos a explorar la implementación de un clásico juego de estrategia llamado *Solitario*. Este juego, también conocido como *Peg Solitaire* en inglés, ha entretenido a jugadores de todas las edades a lo largo de la historia.

El Solitario tiene sus raíces en la Europa del siglo XVII, aunque su origen exacto es difícil de rastrear. En esa época, los juegos de mesa eran una forma popular de pasar el tiempo, y juegos como el Solitario proporcionaban un desafío intelectual, estimulando la mente de los jugadores con sus intrincados movimientos y decisiones tácticas.

Al Solitario se juega en un tablero de madera, generalmente circular, con bolitas colocadas en hendiduras sobre la madera, dispuestas en filas y columnas, formando una cruz, como se aprecia en la siguiente imagen:

Las reglas del juego son simples pero requieren de una cuidadosa planificación para ganar:

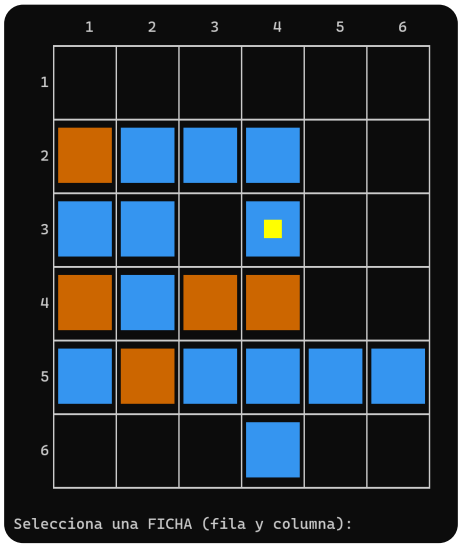
- Se comienza con un tablero lleno de bolitas, que llamaremos *fichas*, excepto el hueco central que queda vacío.
- Las fichas se mueven saltando sobre otras fichas adyacentes, en horizontal o en vertical (no hay saltos en diagonal). La ficha sobre la que se salta se retira del tablero.
- El objetivo final es eliminar todas las fichas excepto una, la cual debe quedar situada en el centro del tablero.

Nosotros vamos a generalizar el juego, trasladándolo a un tablero rectangular, dividido en celdas distribuidas en filas y columnas. No todas las celdas tendrán "hendidura", de tal forma que solo algunas celdas de la cuadrícula podrán contener una ficha (las llamaremos *celdas útiles*). Cada tablero podrá tener una configuración de celdas útiles diferente. Algunas celdas útiles contendrán una ficha mientras que otras estarán vacías.

A pesar de este cambio, el funcionamiento del juego es prácticamente el mismo. A partir de la configuración inicial, el jugador va moviendo fichas de una en una, siempre saltando sobre otra en horizontal o en vertical y cayendo en una celda útil vacía. En cada movimiento, la ficha sobre la que ha saltado es retirada del tablero. El objetivo es quedarse con una única ficha situada en una celda útil distinguida, que llamaremos *celda meta*.

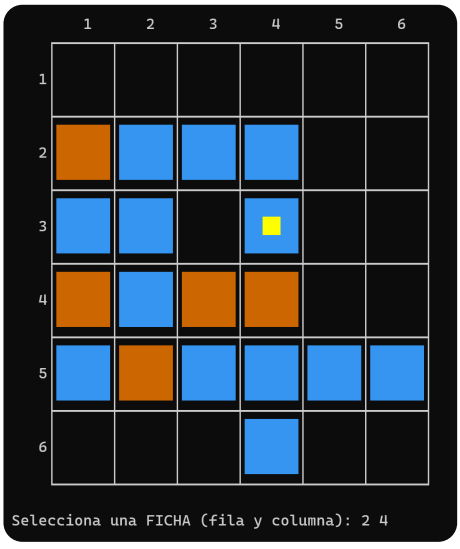
Paso 1. Jugar al Solitario

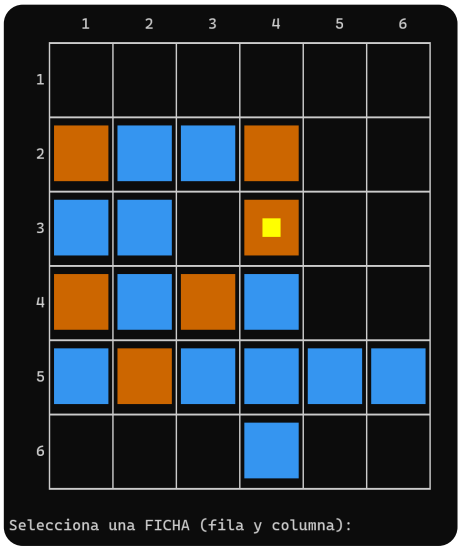
Comencemos a desarrollar una aplicación que permita jugar al Solitario. Al comienzo, el usuario elegirá un archivo que contendrá la configuración inicial del tablero. El tablero se cargará y se mostrará en la consola de la siguiente manera:



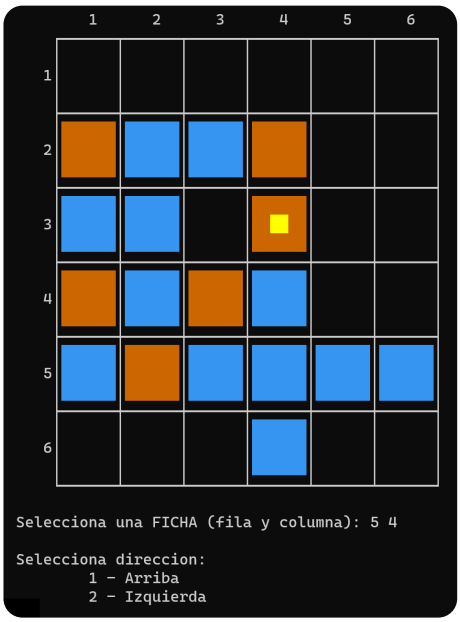
Las celdas negras se corresponden con posiciones no útiles del tablero (en las que nunca podrá colocarse una ficha), las celdas marrones representan posiciones útiles vacías (no contienen una ficha) y las celdas azules posiciones útiles con ficha. La celda meta, la celda en la que debe terminar la última ficha no eliminada, se distingue del resto por contener en su centro un cuadrado amarillo (la celda en la posición (3, 4) del tablero anterior). Esa marca aparece siempre en una celda útil, ya esté vacía o contenga una ficha.

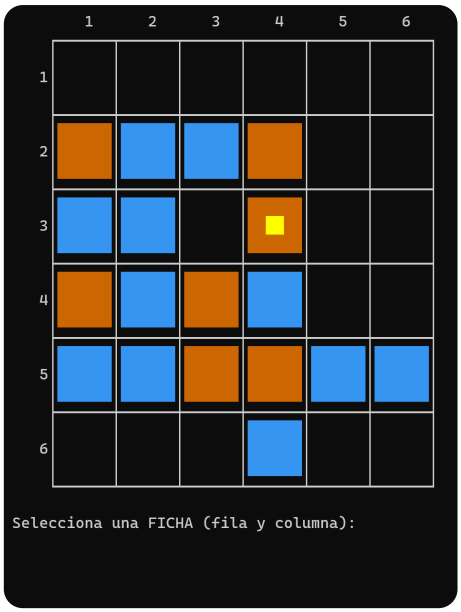
El juego discurre solicitando al usuario la elección de la ficha a mover, denotándola mediante la fila y la columna en la que se encuentra. Si la celda no tiene ficha o esta no se puede mover en ninguna dirección, se mostrará un mensaje de error y se solicitará de nuevo la posición de una ficha. Si la celda tiene una ficha que puede moverse en una única dirección, se realizará el movimiento automáticamente, tal y como ocurre en el ejemplo si seleccionamos la celda en la posición (2, 4):





En cambio, si se elige una ficha que puede moverse en más de una dirección, la aplicación mostrará al usuario las distintas opciones numeradas para que indique el movimiento que quiere hacer seleccionando la dirección en la que debe saltar la ficha. Esto es lo que ocurre con la ficha situada en la celda (5, 4) de la imagen de la izquierda, que puede moverse tanto hacia arriba como hacia la izquierda. Si se elige la opción 2, la ficha se mueve hacia la izquierda, eliminando la ficha en la posición (5, 3), que pasa a estar vacía en la imagen de la derecha:

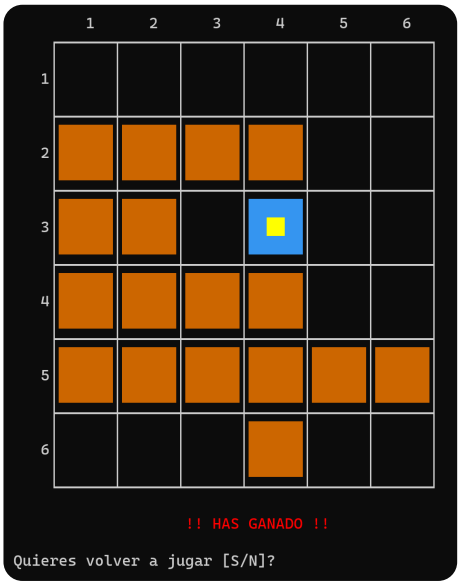


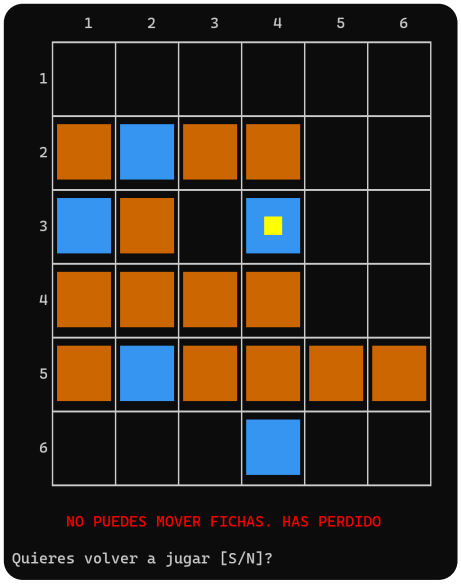


El juego termina cuando no se puede realizar ningún movimiento:

- Si solamente queda una ficha y esta se encuentra en la celda meta, el jugador habrá ganado.
- Si solo queda una ficha pero no está en la celda meta, o si quedan varias, el jugador habrá perdido.

El resultado de la partida se indica con un mensaje informativo y se pregunta al usuario si quiere seguir jugando:





Si el jugador elige `S`, el juego volverá a empezar, mientras que si elige `N` la aplicación terminará.

¡Vamos a implementar nuestro propio juego!

Con el fin de asegurar un progreso efectivo, vamos a utilizar un enfoque modular en el desarrollo de la aplicación. Esto facilitará la ampliación y el mantenimiento del sistema a lo largo del tiempo, simplificando la adición de nuevas características y funcionalidades, así como la gestión de modificaciones.

Módulo `Tablero`

Es el módulo para representar tableros como matrices bidimensionales. Declara un enumerado `Celda` con los valores que puede tener una celda: `NULA` para las celdas no útiles, `VACIA` para las celdas útiles sin ficha y `FICHA` para las celdas útiles con ficha.

```
enum Celda { NULA, VACIA, FICHA };
```

También declara el tipo de datos `Tablero` para la gestión de la matriz bidimensional, con al menos las siguientes operaciones:

```
const int MAXDIM = 10;

class Tablero {
public:
   Tablero(int fils, int cols, Celda inicial);
   int num_filas() const;
   int num_columnas() const;
   bool correcta(int f, int c) const;
   Celda leer(int f, int c) const;
   void escribir(int f, int c, Celda valor);
private:
   int filas, columnas;
   Celda celdas[MAXDIM][MAXDIM];
};
```

El comportamiento de las operaciones es el siguiente:

- `Tablero(fils, cols, val)` es el constructor.
- `num_filas()` y `num_columnas()` sirven para consultar el número de filas y columnas del tablero.
- `correcta(f,c)` devuelve cierto si la posición `(f,c)` es una posición válida dentro del tablero. Aunque en la *interacción con el usuario* las filas y columnas se numeran desde `1`, internamente en la implementación estarán numeradas desde `0`, como es habitual.
- `leer(f,c)` devuelve el contenido de la posición `(f,c)` que supone correcta.
- `escribir(f,c,v)` asigna el valor `v` a la posición `(f,c)` que supone correcta.

Módulo `Movimiento`

Este módulo aglutina todo lo relacionado con el movimiento de una ficha. Está muy relacionado con el juego pero tiene suficiente entidad como para conformar un módulo por separado.

```
enum Direccion { ARRIBA, ABAJO, IZQUIERDA, DERECHA, INDETERMINADA };
std::string to_string(Direccion d);

class Movimiento {
public:
  Movimiento(int f, int c);

  int fila() const;
  int columna() const;
  Direccion dir_activa() const;
  void fijar_dir_activa(Direccion d);

  void insertar_dir(Direccion d);
  int num_dirs() const;
  Direccion direccion(int i) const;

private:
  int fil;
  int col;
  Direccion activa; // de todas las direcciones posibles, contiene la dirección
                    // activa, i.e., la dirección que se va a ejecutar
  int cont; // número de direcciones a las que se puede mover
  Direccion direcciones[NUMDIR]; // direcciones a las que se puede mover

};

// número de direcciones a considerar
const int NUMDIR = 4;
// vectores de dirección: {dif. fila, dif. columna }
const std::pair<int, int> dirs[NUMDIR] = { {-1,0}, {1,0}, {0,-1}, {0,1} };
```

- El enumerado `Direccion` tiene como valores las cuatro direcciones en las que se puede mover una ficha, en vertical o en horizontal. Incluye también el valor `INDETERMINADA` para indicar que una dirección está pendiente de decidir.
- La función `to_string` convierte un valor del enumerado `Direccion` en una cadena de caracteres, para mostrarla en la consola al jugador, por ejemplo.
- El tipo de datos `Movimiento` sirve para representar un movimiento: `fil` y `col` indican la posición de la ficha que va a moverse y `activa`, la dirección en la que se va a mover. Un valor de este tipo se construye por etapas. Primero se fija la fila y la columna a través del constructor. El tipo tiene la operación `insertar_dir(d)` para extenderlo con las direcciones posibles en las que una ficha colocada en esa posición puede moverse, según el estado del juego. Las direcciones posibles se van almacenando en la *lista* `direcciones`. La operación `fijar_dir_activa(d)` sirve para fijar una dirección, cuando esta haya sido elegida por el jugador.

- Las funciones `fila()`, `columna()`, `dir_activa()` y `num_dirs()` permiten consultar los valores de estos atributos, mientras que `direccion(i)` permite consultar la `i`-ésima dirección posible (almacenada en la posición `i` de la lista de posibles `direcciones`).
- El array `dirs` contiene los *vectores de dirección* de las cuatro direcciones, es decir, cómo hay que modificar las filas y las columnas para seguir una dirección. Por ejemplo, para ir hacia arriba se resta `1` a la fila.

Módulo `Juego`

Este es el módulo más importante de la aplicación. Define el tipo de datos `Juego` que representa la configuración actual de un juego, incluyendo el tablero, la posición de la meta y el estado del juego respecto al jugador. También contiene la lógica del juego, es decir, es donde se agrupan y establecen las *reglas del juego*: cuándo un movimiento es correcto, qué opciones de movimientos legales tiene una ficha, en qué consiste realizar una jugada, cuándo hay ganador, etc.

El enumerado `Estado` contiene valores para representar que se está jugando (`JUGANDO`), es decir, que aún quedan fichas con posibilidad de moverse; que se ha terminado con éxito y hay ganador (`GANADOR`); y que se ha llegado a un bloqueo porque las fichas que quedan no tienen ninguna opción de movimiento (`BLOQUEO`).

```
enum Estado { JUGANDO, GANADOR, BLOQUEO };
```

El tipo de datos `Juego` contiene al menos las siguientes operaciones (distingue cuáles son públicas, definiendo la interfaz del tipo de datos, y cuáles privadas, que sirven para la implementación de las otras):

```
class Juego {
public:
   Juego();
   bool cargar(istream &/*ent/sal*/ entrada);
   bool posicion_valida(int f, int c) const;
   void posibles_movimientos(Movimiento &/*ent/sal*/ mov) const;
   Estado estado() const;
   void jugar(Movimiento const& mov);
   void mostrar() const;
private:
   Tablero tablero;
   int f_meta, c_meta;
   Estado estado_int;
   void ejecuta_movimiento(Movimiento const& mov);
   void nuevo_estado();
   bool hay_ganador() const;
   bool hay_movimientos() const;
};
```

En cuanto a la interfaz, el comportamiento de las operaciones es el siguiente:

- `Juego()` es el constructor por defecto. Crea un juego bloqueado con un tablero vacío. Sirve por si hay que crear valores de este tipo antes de conocer su contenido.

- `cargar(entrada)` inicializa el valor del juego a partir del contenido en el flujo de entrada `entrada` (que muy probablemente será un flujo asociado a un archivo de texto, pero no necesariamente). En el flujo el juego se describe de la siguiente manera: primero aparecerán dos números, indicando el número de filas y de columnas del tablero. A continuación, aparecerán filas × columnas enteros (correspondientes a los valores del enumerado `Celda`) con los valores de la matriz guardados por filas de arriba abajo y cada fila de izquierda a derecha. Por último, habrá una línea más con la posición de la meta: dos enteros que describen la fila y la columna donde se encuentra la meta. Por ejemplo, este es el contenido del archivo correspondiente al primer tablero que se ha mostrado en este enunciado:

  ```
  6 6
  0 0 0 0 0 0
  1 2 2 2 0 0
  2 2 0 2 0 0
  1 2 1 1 0 0
  2 1 2 2 2 2
  0 0 0 2 0 0
  2 3
  ```

  El estado inicial del juego es `JUGANDO`. La operación devuelve cierto si toda la lectura ha ocurrido con éxito, y falso en caso contrario.

- `posicion_valida(f,c)` devuelve cierto si y solo si la posición `(f,c)` es una posición correcta del tablero que contiene una ficha.

- `posibles_movimientos(mov)` recibe el movimiento `mov` donde ya se ha fijado una fila y una columna, y lo completa con las direcciones en las que se puede mover la ficha colocada en esa posición, asumiendo que en esa posición hay una ficha. Si no fuera posible ningún movimiento (la ficha está *bloqueada*), la lista de direcciones del movimiento quedaría vacía.

- `estado()` sirve para consultar en qué estado está el juego.

- `jugar(mov)` modifica el juego realizando el movimiento descrito por `mov`. En particular, modifica el tablero (con la operación privada `ejecuta_movimiento(mov)`) y el estado del juego (con la operación privada `nuevo_estado()`, que a su vez hace uso de las operaciones `hay_ganador()` que averigua si ya hay ganador por haber llegado al estado final y `hay_movimientos()` para averiguar si hemos llegado o no a una situación de bloqueo), determinando si el juego ha finalizado, con éxito o fallo.

- `mostrar()` muestra el tablero por la consola, como hemos visto en las capturas al inicio de este enunciado.

En cuanto a la parte privada, el atributo `tablero` guarda la configuración del tablero, los atributos `f_meta` y `c_meta` representan la posición (fila y columna) de la meta, y `estado_int` es el estado actual del juego (seguimos jugando, hay ganador o hemos llegado a un bloqueo).

Módulo `Main`

Este módulo es el encargado del control de la interacción del usuario con el juego. Tiene un bucle principal que permite jugar varias partidas. Cada una comienza pidiéndole al usuario

el nombre del archivo que contiene los datos del juego y después entra en un bucle donde se van realizando jugadas, con la intervención del jugador para elegir esas jugadas.

Este segundo bucle será algo de este estilo:

```
ifstream archivo;
archivo.open(...);
...

Juego solitario;
if (solitario.cargar(archivo)) {
   solitario.mostrar();  // se muestra el estado inicial
   // empezamos a jugar
   do {
      Movimiento movimiento = leer_movimiento(solitario);
      solitario.jugar(movimiento);
      solitario.mostrar();
   } while (solitario.estado() == JUGANDO);
   // mostrar resultado de la p artida (ganador o bloqueo)
}
```

Paso 2. Generación aleatoria de tableros

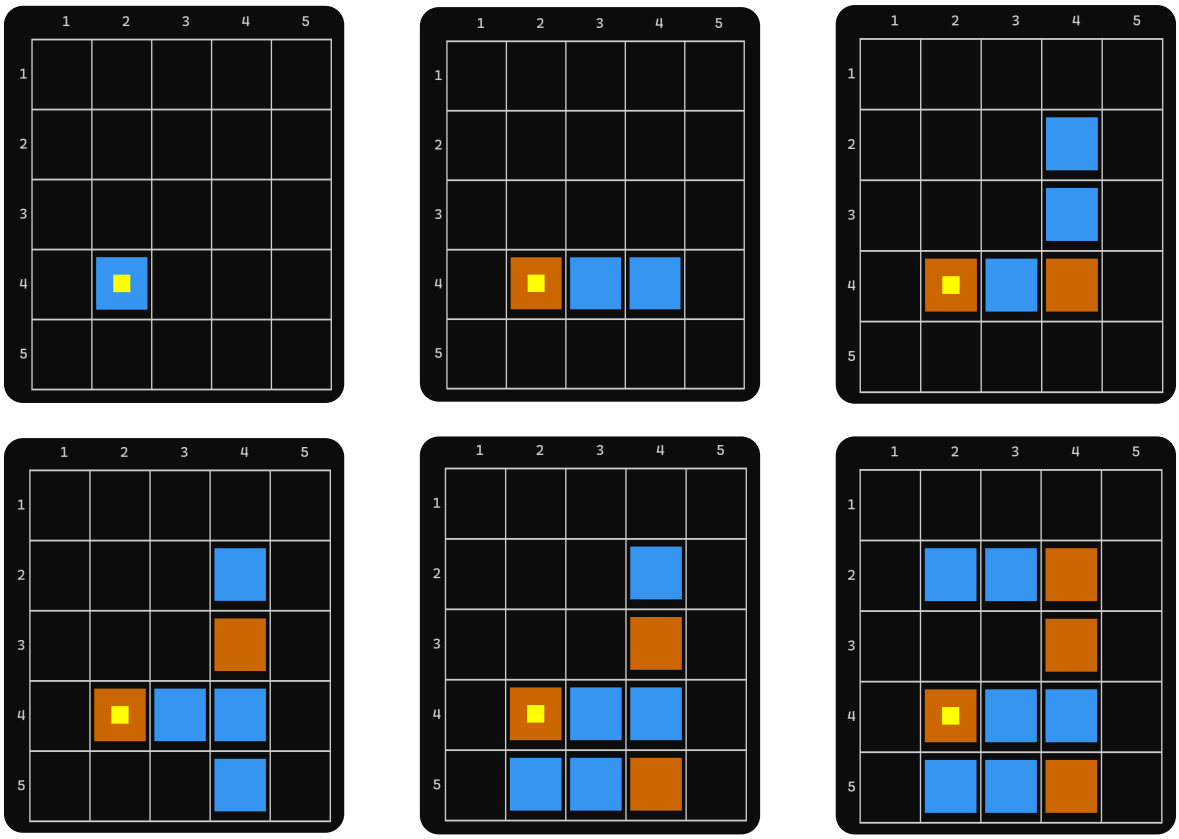
¡Bien! Ya podemos jugar. Pero para divertirnos de verdad necesitamos tableros nuevos, y que la dificultad vaya creciendo.

Por eso vamos a generar tableros de forma aleatoria. Pero si lo hacemos al tuntún, eligiendo de forma totalmente aleatoria qué celdas son útiles o no, o qué celdas útiles tienen ficha y cuáles no, seguramente la mayoría de tableros no se podrían resolver, en el sentido de que no habría una secuencia de pasos que llevaran al estado final con una única ficha en la meta.

Por eso vamos a generar tableros que estemos seguros de que se pueden resolver. Otra cosa es que los jugadores encuentren esa secuencia de pasos que llevan a la victoria (que no tiene por qué ser única, por otro lado). Cuantas más fichas coloquemos en el tablero más difícil será resolverlo.

Una manera de garantizar que esa secuencia de pasos hacia la victoria existe es generar el tablero *hacia atrás*, desde el estado final (con una única ficha en la meta) dando pasos hacia atrás eligiendo en cada momento, de forma aleatoria, un movimiento posible más cuya ejecución nos llevaría al estado actual.

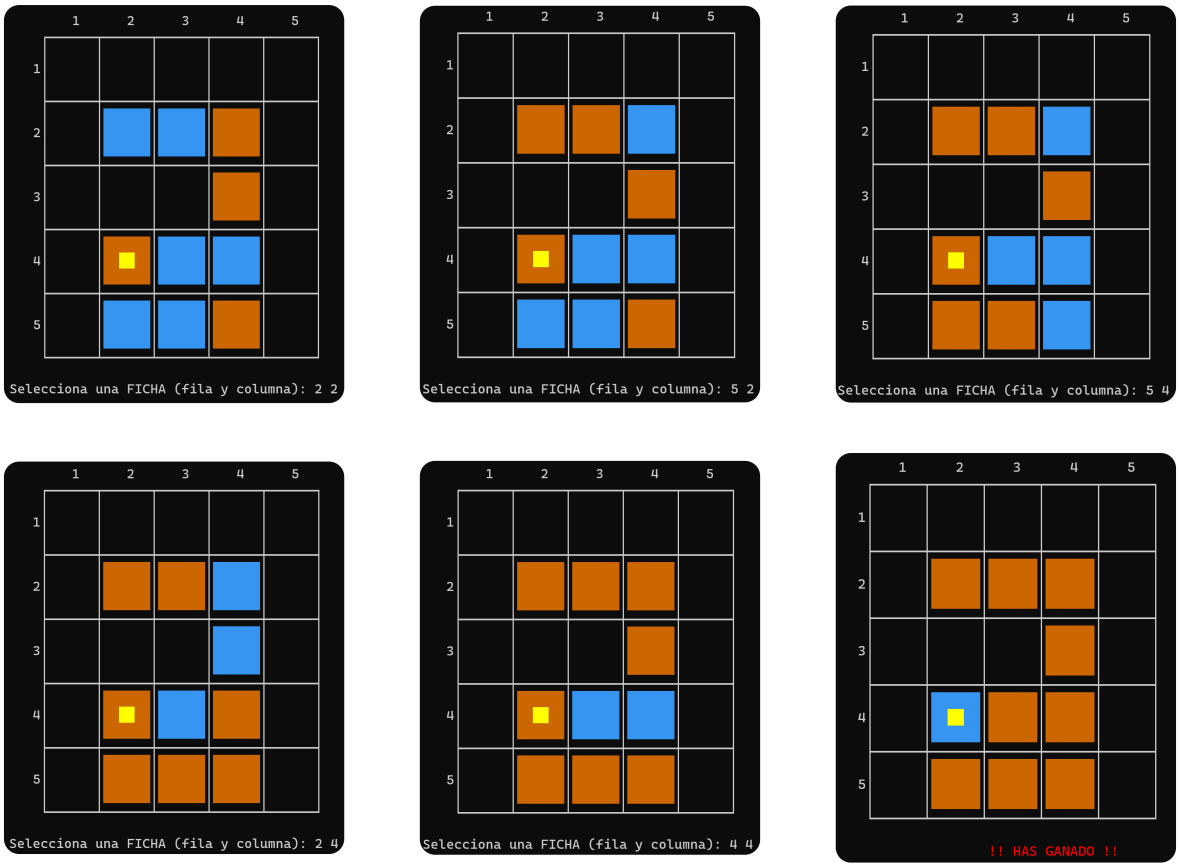
Veamos un ejemplo, en el que pedimos al algoritmo de generación aleatoria que dé 5 pasos:



Se comienza con un tablero con todas sus celdas no útiles. El primer paso consiste en decidir dónde colocar la meta. En este ejemplo se ha colocado la meta en la celda (4, 2), convirtiendo esa celda en una celda útil y colocando una ficha en ella. A partir de ahí el proceso repite una serie de pasos: escoger de forma aleatoria una ficha *F* ya colocada, elegir aleatoriamente una dirección (arriba, abajo, izquierda o derecha) donde encontremos dos celdas seguidas sin ficha (pueden ser ya útiles o no, pero no pueden tener ficha), y aplicar el movimiento inverso, haciendo desaparecer la ficha *F* y haciendo aparecer dos fichas en esas dos posiciones. Si alguna de esas celdas aún no era útil, se convierte en útil.

En el ejemplo, en el primer paso solamente se puede escoger la ficha en la meta. Se mueve a la derecha, colocando una ficha nueva en la posición (4, 3) y la ficha que estaba en la meta en la posición (4, 4). En el siguiente paso se escoge aleatoriamente la ficha en la posición (4, 4) (podría haber sido la otra), y se mueve hacia arriba, llevándola a la posición (2, 4) y habiendo hecho aparecer una ficha nueva en la posición (3, 4). Las celdas que han tenido una ficha pero ya no la tienen permanecen como celdas útiles, pero vacías (las marrones). A continuación, el proceso aleatorio escogió la ficha en la posición (3, 4) y la movió hacia abajo, a la posición (5, 4), haciendo aparecer una ficha nueva en la posición (4, 4). En el cuarto paso se elige la ficha en la posición (5, 4) que se mueve a la izquierda a la posición (5, 2), apareciendo una ficha nueva en la posición (5, 3). Y, por último, se elige la ficha en la posición (2, 4) y se mueve también hacia la izquierda a la posición (2, 2), apareciendo una ficha nueva en la posición (2, 3).

Si ahora jugáramos desde este tablero generado, podríamos pasar por los siguientes pasos (observa que, como es natural, al jugar el número de celdas útiles no cambia, pero cada vez va habiendo más vacías, las marrones):



Para generar tableros de este estilo vamos a añadir a la clase `Juego` otra constructora que recibe un entero, el número de pasos que queremos que dé el algoritmo de generación.

```
class Juego {
public:
    ...
    Juego(int movimientos); // generación aleatoria
```

Para implementar esta constructora puede venir bien añadir más operaciones privadas para seleccionar una ficha de forma aleatoria, o una dirección en la que esa ficha pueda moverse de manera inversa.

Para simplificar el algoritmo, puedes hacer que este pare si al elegir una ficha de forma aleatoria resulta que esta no puede moverse en ninguna dirección. Por tanto, el parámetro de la constructora sería una cota superior al número de pasos que da el algoritmo de generación (y, en consecuencia, al número de fichas con las que termina el tablero). Si quieres hacerlo mejor, puedes hacer que pare solamente si no existe ninguna ficha que pueda moverse de manera inversa. En cualquier caso, no es necesario deshacer movimientos previos si llegamos a una situación de bloqueo, para intentar después que la generación siga caminos alternativos. Eso convertiría el algoritmo en un algoritmo de *vuelta atrás* que será algo que estudies en segundo.

¿Cómo dibujamos el tablero?

Junto al enunciado de la práctica, en el Campus Virtual tienes un fichero `main.cpp` que dibuja un pequeño tablero, de forma muy *ad hoc*. Puedes tomar ideas de ahí pero programándolo

de forma genérica, para tableros de cualquier tamaño. En el fichero `colores.h` hay definidos una serie de colores que puedes utilizar, tanto para el fondo como para el texto. Son cadenas de caracteres con comandos especiales que al escribirse en la consola consiguen cambiar de color.

Por otro lado, no es obligatorio pintar los tableros tal y como aparecen en las capturas de este enunciado. Si prefieres no usar colores, al menos en una fase inicial, puedes hacerlo. Utiliza caracteres diferentes para representar los tres tipos de celdas.

Y recuerda

> *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