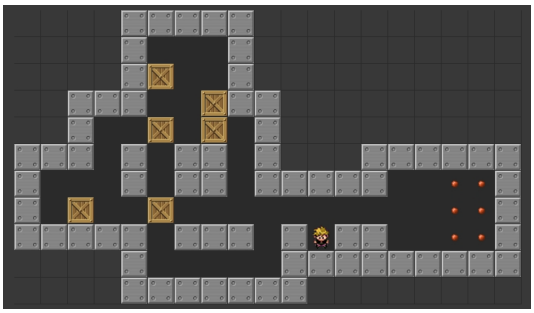
Fundamentos de la programación 2

Práctica 1. Sokoban

Indicaciones generales:

- La línea 1 del programa (y siguientes) deben contener los nombres de los alumnos de la forma:
  `// Nombre Apellido1 Apellido2`
- **Lee atentamente** el enunciado e implementa el programa tal como se pide, con los métodos que se especifican, respetando los parámetros y tipos de los mismos. Adicionalmente pueden implementarse los métodos auxiliares que se consideren oportunos, comentando su cometido, parámetros, etc.
- El programa, además de ser correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.
- La entrega se realizará a través del campus virtual, subiendo únicamente el archivo `Program.cs` con el programa completo.
- El **plazo de entrega** finaliza el 24 de marzo.

En esta práctica vamos a implementar el *Sokoban*, un conocido rompecabezas creado por el japones Hiroyuki Imabayashi en 1981[1]. Se desarrolla en un tablero que representa un almacén con cajas y un empleado que debe empujarlas hasta unas *posiciones destino*. A continuación se muestra un posible nivel del juego[2]:



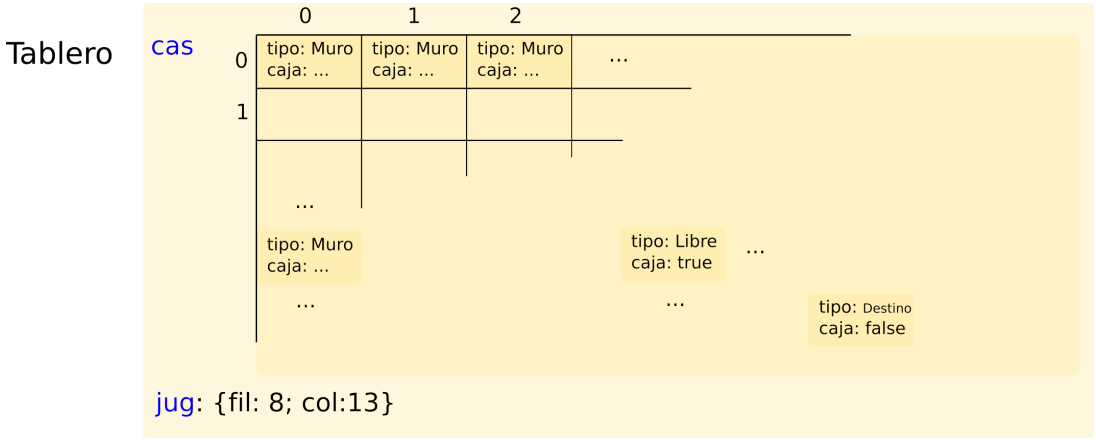
El jugador puede moverse en horizontal y en vertical (sin atravesar los muros, ni las cajas) y puede empujar las cajas a casillas libres. Las cajas no pueden apilarse y tampoco puede desplazarse más de una simultáneamente. El puzle está completado cuando todas las cajas están en las posiciones destino (marcadas con un punto rojo en la imagen anterior).

Vamos a utilizar la siguiente representación para el tablero:

```
struct Coor{ public int fil, col; };  // coordenadas fila y columna en el tablero

enum TipoCasilla {Muro, Libre, Destino};  // 3 tipos de casillas en el tablero

struct Casilla {
  public TipoCasilla tipo; // información fija de la casilla (muro, libre o destino)
  public bool caja;        // información variable: si tiene o no caja
}

struct Tablero{ // tipo tablero
  public Casilla [,] cas; // matriz de casillas
  public Coor jug;        // posición del jugador
}
```

[1] Se puede obtener más información y ver ejemplos en acción en `https://en.wikipedia.org/wiki/Sokoban`
[2] Imagen tomada de `http://sokoban.info/`

El tipo `Coor` servirá para definir posiciones en el tablero. El tipo enumerado `TipoCasilla` define los 3 tipos de casillas que hay en el tablero: muro, libre o destino. El tipo `Casilla` agrupa la información fija de la casilla (el tipo de casilla) y la información variable (si la casilla tiene una caja o no). Por último, el struct `Tablero` contiene una matriz bidimensional de casillas y las coordenadas del jugador. El *estado del juego* quedará representado con una única variable de este tipo `Tablero`. Con esta representación, el tablero del ejemplo anterior tendría el siguiente aspecto en memoria:



Los tableros de juego de los distintos niveles se leerán de un archivo de texto (con tantos niveles como queramos), que tendrá el siguiente formato (véase el archivo `levels` que se proporciona[3]):

```
Level 0
############
#####   ####
#####$  ####
#####  $##
###  $ $ ##
### # ## ##########
#   # ## #####  ..#
# $  $          ..#
##### ### #@##  ..#
#####     #########
#############

Level 1
...
```

En este archivo los niveles vienen separados por una o más líneas en blanco. La línea `"Level x"` indica el comienzo del nivel `x` y a continuación aparece una matriz de caracteres que representa el tablero. Este formato textual es un estándar[4] que utiliza la siguiente codificación para las casillas:

`'#'`	muro	`'$'`	caja sobre casilla libre	`'@'`	jugador sobre casilla libre
`' '`	libre	`'*'`	caja sobre casilla destino	`'+'`	jugador sobre casilla destino
`'.'`	destino				

Es fácil razonar que con esta codificación quedan representados todos los posibles estados de una casilla en el juego.

[3]La mayoría de los niveles se han descargado de `http:/sneezingtiger.com/sokoban/levels.html`.
[4]Puede verse por ejemplo en `http://www.sokobano.de/wiki/index.php?title=Level_format`

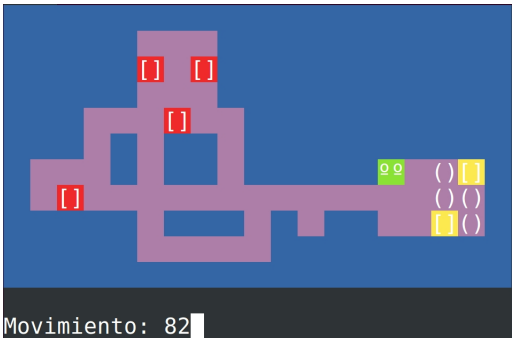
Para programar el juego, implementaremos los métodos que se detallan a continuación. Para cada uno de los parámetros especificados debe determinarse la forma de paso (entrada, salida o entrada-salida):

- `Tablero LeeNivel(string file, int n)`: busca en el archivo `file` el nivel `n` y lo devuelve como salida. En primer lugar, leerá líneas del archivo hasta localizar la línea `"Level n"`). Las siguientes líneas (hasta encontrar una en blanco) contienen el tablero. Como a priori no conocemos el número de filas y columnas del tablero a cargar (y no queremos leer de archivo más de una vez), **leeremos todas las líneas de la matriz en una cadena de texto `s`, utilizando ';' como separador de línea**. Por ejemplo, para el tablero del ejemplo anterior tendremos:

  ```
  s = "###########;##### ####;#####$ ####;##..."
  ```

  A partir de esta cadena podemos determinar el número de filas y columnas del tablero, y crear una matriz `cas` del tamaño exacto. A continuación se rellena esa matriz de acuerdo a la codificación explicada arriba.

- `void Dibuja(Tablero tab, int mov)`: dibuja en pantalla el estado actual del juego dado en `tab` y el contador de movimientos `mov`. Para el tablero de nuestro ejemplo, tras algunos movimientos, podríamos tener este resultado:



  Para que las casillas tengan aspecto cuadrado en el dibujo, utilizamos dos caracteres por cada casilla: `ºº` para el jugador, `[]` para las cajas en casillas libres, `()` para las casillas en destino, etc. Nótese que las cajas que están en posiciones destino se muestran en amarillo para que la caja no oculte esa información.

Importante: en este juego utilizamos 3 representaciones distintas para el estado el juego. La representación textual del archivo, la representación en memoria con el tipo `Tablero` (junto con los tipos `Coor`, `TipoCasilla` y `Casilla`), y la representación gráfica en pantalla. La representación textual solo se utiliza en `LeeNivel` para generar la representación memoria (`Tablero`). La representación gráfica se genera en `Dibuja` y solo se utiliza ahí. La representación en memoria **es la única** que se utiliza en los métodos que vienen a continuación.

- `char LeeInput()`: recoge de teclado una pulsación de teclado y la devuelve (entrada no bloqueante). Esto puede hacerse del siguiente modo:

```
if (Console.KeyAvailable) {
  string tecla = Console.ReadKey ().Key.ToString ();
  ...
```

Las pulsaciones de las flechas de cursor se recogen en `tecla` como `"LeftArrow"`, `"UpArrow"`, `"RightArrow"` y `"DownArrow"` y este método las devolverá respectivamente como `'l'`, `'u'`, `'r'`, `'d'`. Más adelante será necesario procesar otras teclas como `'q'` para abortar juego, `'z'` para deshacer jugada, etc.

- `bool Siguiente(Coor pos, char dir, Tablero tab, Coor sig)`: calcula en `sig` la posición siguiente a `pos` en la dirección `dir` (que traerá uno de los valores `'u'`, `'d'`, `'l'` o `'r'`). Además, devolverá `true` en caso de que `sig` esté dentro del tablero `tab` y `false` en otro caso.

- `char Mueve(Tablero tab, char dir)`: mueve el jugador en la dirección `dir`, si es posible, empujando además la caja correspondiente si hay tal caja. Si el movimiento no es posible (porque choca contra un muro o una fila de dos o más cajas), el tablero no se modificará. Será **muy útil el método** `Siguiente` descrito en ítem precedente.

  Si el movimiento ha sido posible, devolverá además el carácter de entrada `dir`. Nótese que si solo guardamos los movimientos como `'u'`, `'d'`, `'l'`, `'r'`, cuando el jugador mueve una caja, en el tablero resultante no podremos saber si el caja ya estaba ahí o lo acaba de desplazar el jugador. Para deshacer esta ambigüedad, cuando el jugador mueva una caja guardaremos el movimiento en mayúscula `'U'`, `'D'`, `'L'`, `'R'`. Esto nos permite determinar el estado previo del tablero y facilitará después la opción de deshacer jugada.

- `void ProcesaInput(Tablero tab, char dir, string movs)`: modifica el tablero `tab` de acuerdo con el input recibido en `dir`. En una primera versión solo tiene que procesar los movimientos del jugador, llamando al método método anterior `Mueve`.

  En la cadena `movs` irá *acumulando* los caracteres correspondientes a los movimientos del jugador. Después puede incluirse la opción de deshacer cuando el usuario pulse `'z'` (hay que incluir este carácter en el repertorio de `LeeInput`).

- `bool Terminado(Tablero tab)`: comprueba si el nivel está completado: todas las cajas están en un posición destino.

- `void Main()`: en una primera versión, solicitará un nivel de juego e implementará el bucle principal del juego utilizando los métodos anteriores.

Una vez terminada la versión inicial, vamos a implementar algunas extensiones:

- Implementar el juego de manera que guarde memoria sobre los niveles ya superados junto con la secuencia de movimientos utilizada. Esta información se guardará en otro archivo de texto en un formato adecuado. De este modo podrá arrancar directamente con el primer nivel que aun no se ha superado (bloquea el acceso a los siguientes), o bien se podrá repetir alguno de los niveles ya superados, para intentar reducir el número de movimientos empleados.

  Cuando se selecciona un nivel ya superado, también se puede permitir la opción de reproducir la solución *en modo demo*, mostrando movimiento a movimiento con un retardo entre ellos.

- Implementar la opción de *salvar estado*, de modo que si jugador pulsa `'s'` se salve el estado actual del juego en un archivo de texto para recuperarlo en otra ejecución. El formato de guardado debe ser el mismo que el del archivo de entrada, de modo que se pueda reutilizar el código de lectura.