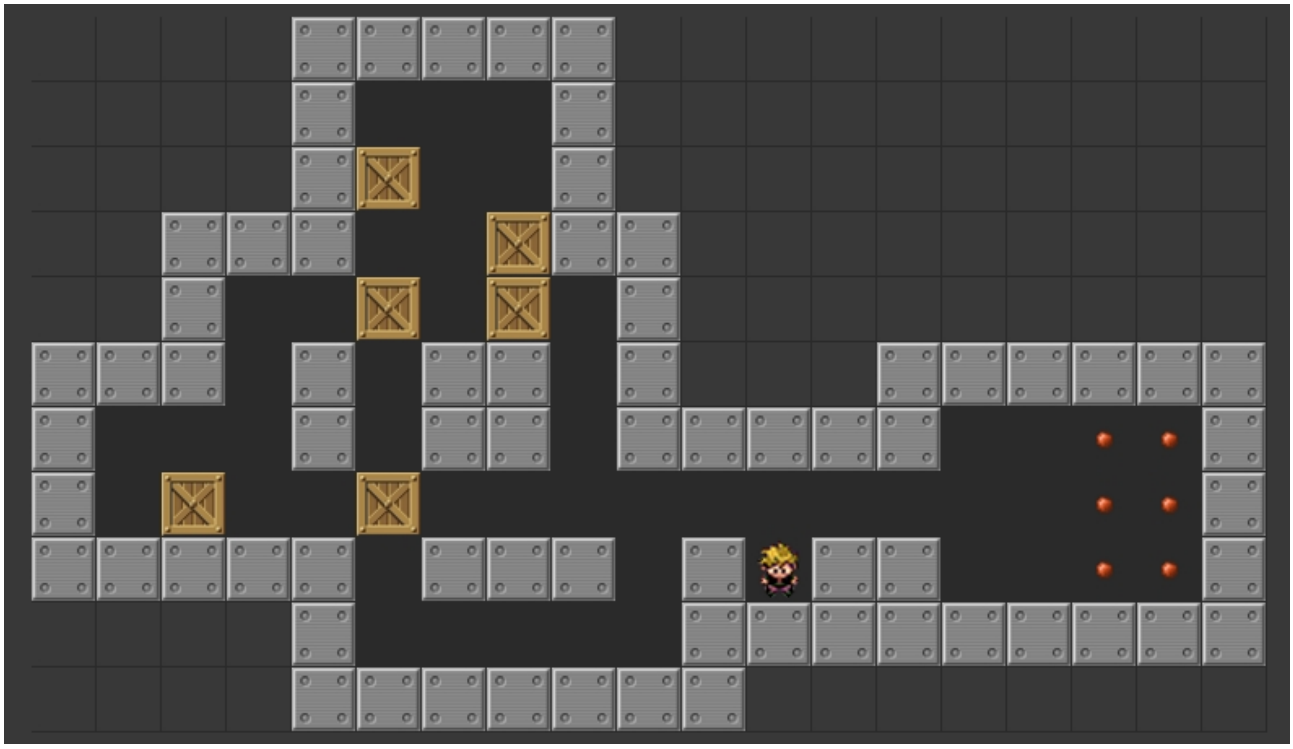
Práctica 3
Sokoban

Fecha de entrega: 26 de marzo

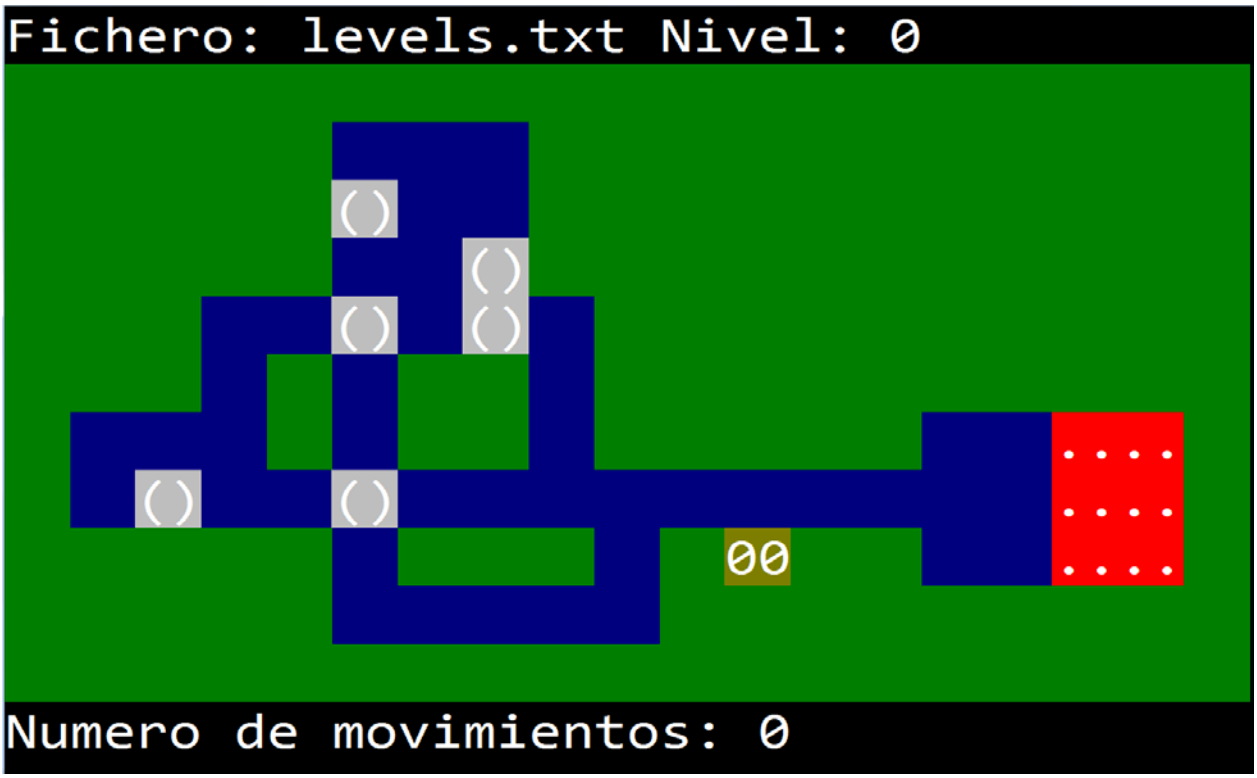
1. Descripción del juego

La práctica consiste en desarrollar un programa en C++ para jugar al **Sokoban**, un conocido rompecabezas creado por el japonés Hiroyuki Imabayashi en 1981, aunque después se han implementado distintas versiones y variantes. Se puede obtener más información y ver ejemplos en acción en https://en.wikipedia.org/wiki/Sokoban.
El juego se desarrolla en un tablero que representa un almacén con cajas y un empleado que debe empujar dichas cajas hasta unas determinadas posiciones destino. Dependiendo de la dificultad, los tableros tienen asignado un nivel. A continuación se muestra un posible nivel del juego (imagen tomada de http://sokoban.info/):



El jugador puede moverse en horizontal o en vertical (sin atravesar los muros, ni las cajas). Puede empujar las cajas a una casilla contigua libre (las cajas no pueden apilarse, ni desplazar más de una simultáneamente). El puzle está terminado cuando

todas las cajas están en las posiciones destino (marcadas con punto rojo en la imagen anterior).
El tablero anterior, que se ha cargado desde el fichero `levels.txt` y tiene nivel 0, lo podemos representar de la siguiente forma sobre la consola:



Versión 1 del programa

El programa simulará de forma realista la dinámica del juego, aunque en modo de consola. Inicialmente mostrará un menú con dos opciones:

```
1. Jugar partida.
0. Salir
```

La opción 1 permitirá cargar un tablero de un determinado nivel desde el fichero que indique el usuario. Se podrán jugar partidas hasta que se elija la opción 0.

1. Datos del programa

El tipo enumerado `tCasilla` nos permite representar las casillas del tablero:

```
typedef enum tCasilla{Libre,Muro,DestinoL,DestinoC,DestinoJ,Jugador,Caja}
```

donde `DestinoL`, `DestinoC` y `DestinoJ` representan posiciones destino que están libres, con caja o con jugador respectivamente.

Para mantener el estado del tablero el programa usa un array bidimensional de `tCasilla`. Declara la correspondiente constante `MAX=50` y el tipo `tTablero` para representarlo.

Define el tipo estructurado `tSokoban` para describir el estado del tablero, conteniendo:

- ✓ El tablero de tipo `tTablero`.
- ✓ El número de filas `nfilas` y de columnas `ncolumnas` del array bidimensional que está ocupando el tablero (ambas <= `MAX`).
- ✓ La `fila` y la `columna` donde se encuentra colocado el jugador.
- ✓ El número de `cajas` del tablero.
- ✓ El número de cajas ya `colocadas` en casilla destino.

Define también el tipo estructurado `tJuego` que describe el estado del juego:

- ✓ El estado del tablero `sokoban` de tipo `tSokoban`.
- ✓ El número de movimientos realizados hasta el momento `numMovimientos`.
- ✓ El nombre del fichero `nFichero` del que se ha cargado el juego.
- ✓ El `nivel` que estamos jugando.

El programa usa también un tipo enumerado `tTecla` con los siguientes valores: `Arriba`, `Abajo`, `Derecha`, `Izquierda`, `Salir` y `Nada`.

2. Visualización del tablero

Cada vez que vayas a visualizar el estado del tablero borra primero el contenido de la ventana de consola, de forma que siempre se muestre el tablero en la misma posición y la sensación sea más visual. Para borrar la consola utiliza:

```
system("cls");
```

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 Visual Studio, por lo que debemos ser conscientes de que el programa no será portable a otros compiladores.

Disponemos de 16 colores diferentes entre los que elegir, con valores de 0 a 15, tanto para el primer plano como para el fondo. El 0 es el negro y el 15 es el blanco. Los demás son azul, verde, cian, rojo, magenta, amarillo y gris, en dos versiones, oscuro y claro.

Visual Studio incluye una biblioteca, `Windows.h`, que tiene, entre otras, rutinas para la consola. Una de ellas es `SetConsoleTextAttribute()`, que permite ajustar los colores de fondo y primer plano. Incluye en el programa esa biblioteca y esta rutina:

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

Basta proporcionar un color para el fondo (1 a 14) y esa rutina lo establecerá, con el color de primer plano en blanco (15). Debes cambiar el color de fondo cada vez que tengas que *dibujar* una casilla y volverlo a poner a negro (0) a continuación.

Al menos debes implementar estos subprogramas:

- ✓ `void dibujaCasilla(tCasilla casilla)`: muestra una casilla del tablero.
- ✓ `void dibujar(const tJuego &juego)`: muestra el tablero del juego, el nombre del fichero desde que se ha cargado, su nivel y el número de movimientos realizados.

2. Carga de un nivel del juego

Los tableros de juego de los distintos niveles se leerán de un archivo de texto que puede guardar tantos niveles diferentes como queramos.
Por ejemplo, si el tablero del ejemplo anterior corresponde al nivel 0, en el archivo tendremos:

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

La primera línea define el número del nivel. A continuación aparece una matriz de caracteres, donde '#' representa muro, ' ' (blanco) es casilla vacía, '.' es casilla destino libre, '$' es caja y @ representa el jugador. Aunque no aparece en este ejemplo, en otros niveles el carácter '*' representa caja en casilla destino y '+' representa jugador en casilla destino. En este archivo los niveles vendrán separados por una línea vacía.
En el Campus Virtual se proporcionarán varios archivos con un repertorio de niveles, la mayoría descargados de http://sneezingtiger.com/sokoban/levels.html.

Observa que la matriz de caracteres no es cuadrada, y que los campos `nfilas` y `ncolumnas` quedan determinados por la lectura.

Implementa al menos los siguientes subprogramas:

- ✓ `void inicializa(tJuego &juego)`: inicializa el `tablero`, haciendo que todas las `MAX x MAX` casillas estén libres y el número de movimientos a 0.

- ✓ `bool cargarJuego(tJuego & juego)`: solicita al usuario el fichero y el nivel que desea jugar y lo carga desde dicho fichero.
- ✓ `bool cargarNivel(ifstream &fichero, tSokoban &sokoban, int nivel)`: busca en el fichero el nivel solicitado y carga el tablero correspondiente. Devuelve un booleano indicando si lo ha encontrado.

3. El juego en acción

3.1 Lectura de teclas especiales

Para leer las teclas pulsadas por el usuario implementa el siguiente subprograma:

- ✓ `tTecla leerTecla()`: devuelve un valor de tipo `tTecla`, que puede ser una de las cuatro direcciones si se pulsan las flechas de dirección correspondientes; el valor `Salir`, si se pulsa la tecla Esc; o `Nada` si se pulsa cualquier otra tecla.

La función `leerTecla()` detectará la pulsación por parte del usuario de teclas especiales, concretamente las teclas de flecha (direcciones) y la tecla Esc (salir). La tecla Esc sí genera un código ASCII (27), pero las de flecha no tienen códigos ASCII asociados. Cuando se pulsan en realidad se generan dos códigos, uno que indica que se trata de una tecla especial y un segundo que indica de cuál se trata.

Las teclas especiales no se pueden leer con `get()`, pues esta función sólo devuelve un código. Podemos leerlas con la función `_getch()`, que devuelve un entero y se puede llamar una segunda vez para obtener el segundo código, si se trata de una tecla especial. Esta función requiere que se incluya la biblioteca `conio.h`.

```
cin.sync();
dir =_getch(); // dir: tipo int
if (dir ==  0xe0) {
   dir = _getch();
   // ...
}
// Si aquí dir es 27, es la tecla Esc
```

Si el primer código es `0xe0`, se trata de una tecla especial. Sabremos cuál con el segundo resultado de `_getch()`. A continuación puedes ver los códigos de cada tecla especial:

↑ 72 ↓ 80 → 77 ← 75

3.2 Movimiento del tablero

Una vez que el usuario indica la dirección, tenemos que realizar el movimiento del jugador sobre el tablero. Para esto, implementa el siguiente subprograma:

- ✓ `void hacerMovimiento(tJuego &juego, tTecla tecla)`: realiza el movimiento del jugador en la dirección indicada. Si no se puede realizar el movimiento, no tiene efecto y no se incrementa tampoco el número de movimientos registrados.

Ya tienes todo lo que necesitas para implementar en `main()` la dinámica del juego.

Versión 2 del programa

En esta versión del programa añadiremos dos nuevas funcionalidades:

- Informar al jugador de que se ha quedado bloqueado porque alguna de las cajas que aún no ha sido colocada en destino ha quedado atrapada en una esquina.
- Ofrecer al jugador la posibilidad de deshacer hasta los últimos `MAXH=10` movimientos.

De esta forma si el juego se queda bloqueado por la razón mencionada en algún momento el jugador puede deshacer los últimos movimientos.

Para informar al jugador del bloqueo define una función:

- ✓ `bool bloqueado(const tJuego &juego)`: que indica si alguna de las cajas no colocadas en destino ha quedado atrapada en una esquina

Para deshacer movimientos vamos a añadir al enumerado `tTeclas` un nuevo valor `Deshacer` que se obtendrá al pulsar la tecla `d` o `D` (códigos ASCII 100 y 68). Dicha tecla se podrá pulsar hasta un máximo de `MAXH` movimientos.

Además vamos a extender el tipo `tJuego` con un nuevo campo de tipo `tHistoria` que almacena los últimos movimientos realizados. Contiene al menos:

- ✓ Un array `tableros` de `tSokoban` conteniendo los últimos estados del tablero.
- ✓ El número de elementos `cont` del array que están ocupados, inicialmente 0.

La historia del juego se modifica de la siguiente forma:

- Con objeto de almacenar solamente los últimos estados del tablero, el array se va rellenando de izquierda a derecha y se ha de tener en cuenta que el máximo número de tableros que se pueden almacenar es `MAXH.`
- Cada vez que se realiza un movimiento que modifica el estado del tablero se almacena en el array `tableros` el estado del tablero antes de realizar el movimiento. Modifica adecuadamente el subprograma `hacerMovimiento`.
- Cada vez que el usuario pulse la tecla `d` o `D` se deshace el movimiento anterior para lo cual debes implementar un subprograma:
  - ✓ `bool deshacerMovimiento(tJuego &juego)`

Versión 3 del programa

En esta versión del programa se pretende gestionar la información sobre los niveles que ha superado un jugador y en cuantos movimientos lo ha hecho.

La información del jugador estará almacenada en un fichero que se cargará al comenzar el programa y se guardará en dicho fichero al salir del programa. Antes de mostrar el menú se solicitará al jugador su nombre, que coincidirá con el nombre del fichero con su información con extensión `.txt`.
El fichero contendrá la siguiente información: nombre del fichero desde donde se cargó el juego, nivel del juego y el menor número de movimientos en que se resolvió. El fichero está ordenado de menor a mayor con respecto a los dos primeros campos. Un ejemplo de fichero podría ser el siguiente:

```
levels.txt -1 15
levels.txt 0 250
microban.txt 0 20
microban.txt 2 40
microban.txt 3 100
```

Se añadirá al menú una nueva opción 2 para visualizar dicha información.

Define:

- ✓ Un tipo estructurado `tPartida` para representar la información de cada partida resuelta.
- ✓ Un tipo `tExitos` que es un array conteniendo todas las partidas ganadas de tamaño máximo `MAXE=100.` Dicho array se mantendrá ordenado como en el fichero.
- ✓ Un tipo estructurado `tInfo` que contiene el nombre del jugador, el array de éxitos y un contador del número de partidas resueltas.

Cada vez que el jugador resuelve un juego, se actualiza la información del jugador:

- ✓ Si el jugador ya había jugado previamente al mismo juego (fichero y nivel) y ha conseguido resolverlo en menos movimientos se actualiza la información correspondiente.
- ✓ En caso contrario, se inserta de forma ordenada la nueva información.

Para buscar e insertar un valor de tipo `tPartida` en el vector de partidas ganadas redefine los operadores == y < para dicho tipo.

Aspectos de implementación

No uses variables globales: cada subprograma, además de los parámetros para intercambiar datos, debe declarar sus propias variables locales, aquellas que necesite usar en el código.

No uses instrucciones de salto como `exit`, break (más allá de las cláusulas de un `switch`) y `return` (en otros sitios distintos de la última instrucción de una función).

4 Entrega de la práctica

La práctica se entregará a través del Campus Virtual. Se habilitará una nueva tarea **Entrega de la Práctica 3** que permitirá subir el archivo con el código fuente.

Fin del plazo de entrega: **26 de marzo a las 23:55**