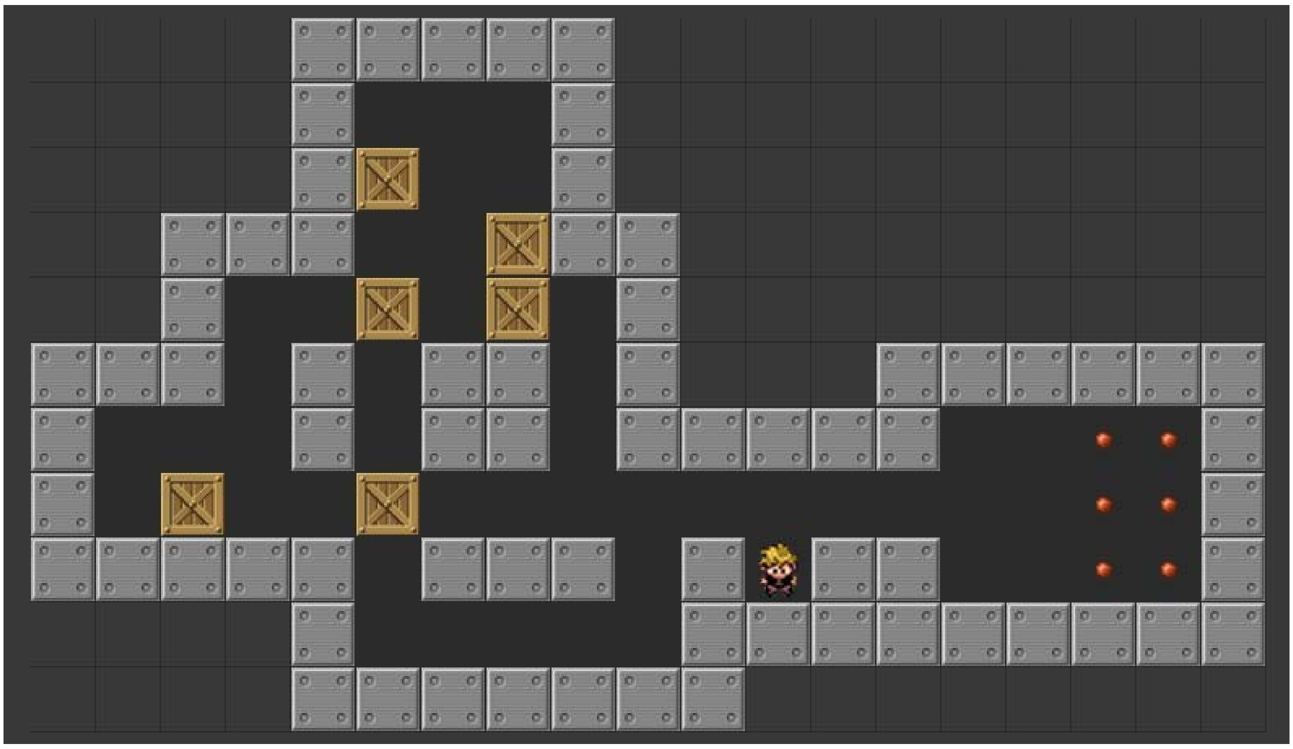
Project 3
Sokoban

Deadline: March 26, 2016

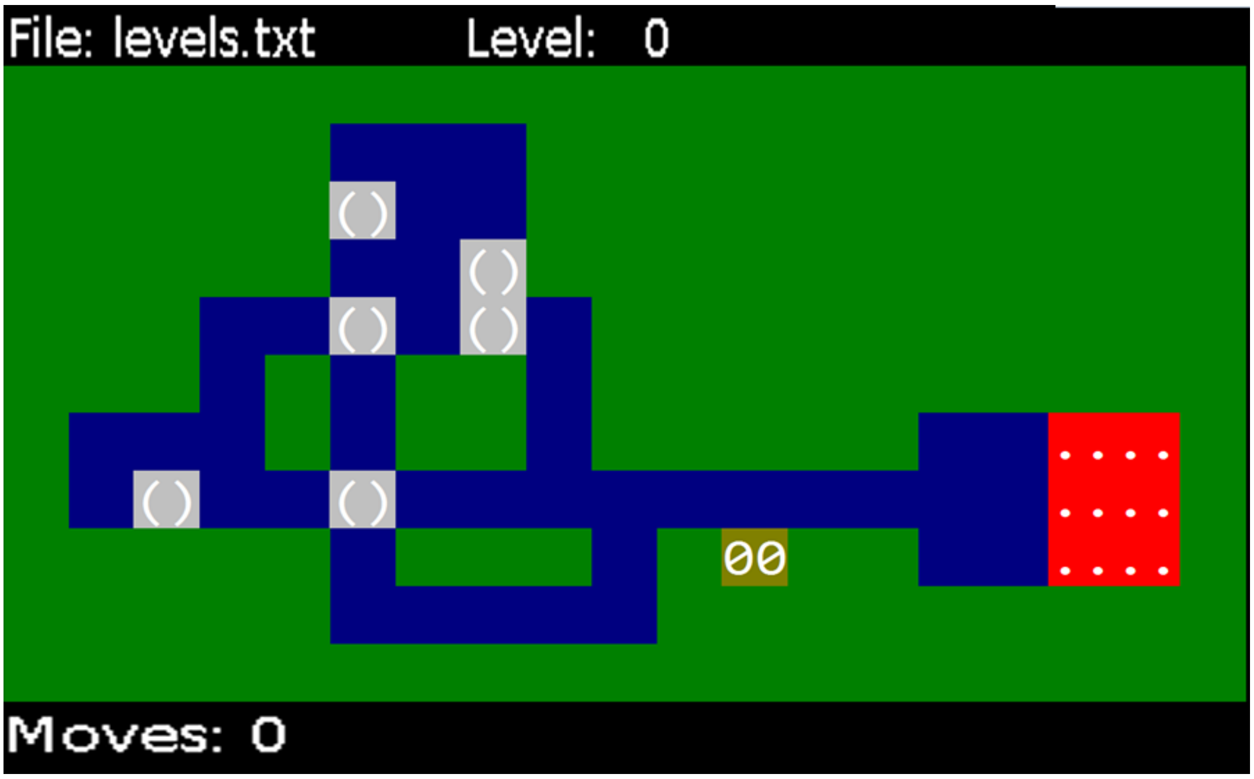
1. Game description

The project consists on the development of a C++ program that lets the user play **Sokoban**, a well-known game created by the Japanese Hiroyuki Imabayashi in 1981, although several variants and versions have been implemented since. More information and examples can be checked in https://en.wikipedia.org/wiki/Sokoban. The game takes place in a board representing a store with boxes and an employee that must push those boxes to certain goal positions. Depending on the difficulty, the boards are assigned a level. Next, a Sokoban level is shown (from http://sokoban.info/).



The player can move horizontally or vertically (it cannot cross walls or boxes). It can push the boxes to a free adjacent tile (the boxes cannot be stacked, and no more than one box can be pushed at the same time). The puzzle is solved when all boxes are placed in goal positions (marked with a red dot in the previous example).

The previous board, which has been loaded from the file `levels.txt` and is of level 0, can be represented in the next way in the terminal:



2. Version 1

The program will simulate the game dynamics realistically, although in console mode. Initially, it will provide a menu with two options:

```
1. Play match
0. Exit
```

Option 1 will load a board of a certain level from a user-specified file. Option 0 will exit the game.

2.1 Program data

The enumerated type `tTile` lets us represent the board tiles:

```
typedef enum {Free, Wall, GoalFree, GoalBox, GoalPlayer, Player, Box} tTile;
```

where `GoalFree, GoalBox and GoalPlayer` represent positions on which there is a free slot (with a goal), a goal with a box, and a goal on which the player is standing.

In order to store the board state, use a 2-dimensional array of type `tTile`. Declare the corresponding constant `MAX=50` and the type `tBoard` to represent it.

Define the structured type tSokoban to describe the board status, containing:

- The board (type tBoard).
- The number of rows nrows and columns ncolumns of the 2-dimensional array describing the board (both <= MAX).
- The row and the column where the player is located.
- The number of boxes on the board.
- The number of placed boxes already placed in a goal position.

Define the structured type tGame that describes the game state:

- The board state sokoban, whose type is tSokoban.
- The number of moves taken so far, numMovs.
- The filename nFileName from which the game has been loaded.
- The level we are playing.

The program will also use an enumerated type tKey with the next values: Up, Down, Right, Left, Exit and Nothing.

2.2 Board display

Each time you have to display the state of the board, clean the content of the console window first, in such a way that the board is always displayed in the same position and you get a more pleasant experience. You can clean the console with:

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

By default, the foreground color, the one that is used to print the characters, is white, whereas the background color is black. We can change these colors, of course, although we have to do it by using Visual Studio-specific function. This means our code will not compile in other compilers. If equivalent behavior is desired in other platforms, you are allowed to implement it, but you will get no official support.

We have 16 different colors available to choose. The values range from 0 to 15, both for the foreground color and the background color. 0 is black and 15 is white. The others are blue, green, cyan, red, magenta, yellow and grey, in light and dark versions. Visual Studio includes a library, Windows.h, that includes, among others, terminal-specific functions. One of them is SetConsoleTextAttribute(), that allows to adjust the foreground and background colors. Include in your program this library and this code:

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

You just have to provide a background color (1 to 14) and this function will set it for you. The foreground will be set to white (15). You must change the background color each time you *draw* a tile and set it back to black (0) afterwards.

At least, you have to implement these subprograms:

- `void drawTile(tTile tile)`: draws a tile in the board.
- `void draw(const tGame &game)`: displays the board, the name of the file that has been loaded, its level and the number of taken moves.

2.3 Loading game levels

Game boards will be read from a text file that can store many levels.

For instance, if the board from the previous example corresponds to level 0, this is what we will have in the file:

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

The first line sets the level. Next, a character matrix describes the level. In this matrix, '#' represents a wall, ' ' (whitespace) represents an empty tile, '.' is a goal tile, '$' is a tile with a box, and '@' represents the player. Although it is not depicted in this example, '*' represents a box on a goal tile, and '+' represents the player on a goal tile. In this file, levels are separated by an empty line. Several files with levels are provided in the Virtual Campus, most of them downloaded from:

http://sneezingtiger.com/sokoban/levels.html.

Keep in mind that the character matrix **is not square**. And the value of the field `nrows` and `ncolumns` are set on file load.

Implement at least the next subprograms:

- `void init(tGame &game)`: initializes the `board`, making all `MAX` x `MAX` tiles to free and the number of moves to 0.
- `bool loadGame(tGame &game)`: asks for a file name and the desired level and loads it from the file.

- `bool loadLevel(ifstream &stream, tSokoban &sokoban, int level)`: searches the chosen level in the file and loads the corresponding board. It returns a boolean informing whether the file was found or not.

2.4 The game

Reading special keys

Implement the next subprogram to read the keys pressed by the user:

- `tKey readKey()`: returns a value of type `tKey`, that can be one of the four possible directions when the corresponding keys are pressed; and `Exit`, if the Esc key is pressed; or Nothing if any other key is pressed.

The function `readKey()` will detect the keypress from the user, specifically the arrow keys (directions), and the Esc key (exit). The Esc key generates an ASCII code (27), but arrow keys do not have associated ASCII codes. When the arrow keys are pressed, two codes are generated, one informing that a special key has been pressed, and a second code letting the programmer know which special code it is.

Special keys cannot be read with `get()`, since this function returns only one code. We can read them with `_getch()`, that returns an integer and can be called a second time to obtain the second code, if it is a special key (the library `conio.h` has to be included).

```
cin.sync();
dir =_getch(); // dir: int
if (dir == 0xe0) {
   dir = _getch();
   // ...
}
// If dir is 27, it's the Esc key
```

If the first code is `0xe0`, it is a special key. We will know which one it is after we analyze the value of the second call to `_getch()`. These are the codes of the arrow keys:

↑: 72   ↓: 80   →: 77   ←. 75

Moving

Once the user has input the direction, we have to apply the move to the player on the board. Implement the next subprogram:

- `void applyMove(tGame &game, tKey key)`: applies the move in the given direction. If the move cannot be applied, the call has no effect and the number of moves is not increased either.

You have now all the pieces to implement the main game mechanics in `main()`.

3. Version 2

In this version, new functionalities will be added:

- Inform that the game is impossible to be won because a box has been placed on a corner.
- Let the user undo the last `MAXH=10` moves.

In this way, if a box gets locked, the user can undo the last moves.

In order to inform the player of this possible lock, implement this function:

- `bool blocked(const tGame &game)`: indicating if there is a box that has been locked in a corner.

In order to undo the moves, we will add, to the enumerated value `tKeys` a new value, `Undo`, that will be obtained when pressing d or D (ASCII codes 100 y 68, respectively). Such a key can be pressed up to `MAXH` times.

Additionally, we will extend the type `tGame` with a new field of type `tHistory` that stores the last taken moves. At least, it contains:

- An array of boards of type `tSokoban`. These are the last board states.
- The number of filled elements in the array (initially, 0).

Game history is modified according to these rules:

- The array will be filled from left to right, keeping in mind that the maximum number of storable steps is `MAXH`.
- Each time a board-modifying move is taken, the current board (the board before applying the move) is stored in the boards array. Modify the subprogram `applyMove` accordingly.
- Each time the user pressed the d or D key, the last move is undone. You have to implement the next subprogram for that:

  ```
  bool undoMove(tGame &game)
  ```

4. Version 3

In this version, user information will be managed. Won levels and the number of moves will be stored.

Player information will be stored in a file that will be loaded when the program starts and it will be saved once the program exits. Before displaying the menu, the user will be asked for his or her name. This name will match the file name (the file name will also have the extension `.txt`).

The file will contain the next information: the file name from which the game was loaded, the level of the board and the best score (the minimum number of moves the user applied to solve the board). The file will be sorted in ascending order, taking into account the two first fields.

An example file:

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

A new option (2) that displays the information will be added to the menu.

Implement:

- A structured type `tMatch` to represent the information of a won match.
- A new type `tWon`, which is an array storing all matches the user has won (maximum `MAXE=100` elements). Such array will be stored, just like the file.
- A structured type `tInfo` containing the name of the player, an array of won matches and a counter for that array.

Every time the players wins a game, the user information is updated:

- If the player had already won the game (file and level), the corresponding information is updated.
- Otherwise, a sorted insertion is applied.

In order to find and insert a value of type `tMatch` in the array of won matches, redefine the operators == and < for that type.

4.1 Implementation aspects

Do not use global variables. Each subprogram, besides its parameters to get data in and out the function, has to declare its own local variables.

Do not use jump instructions like `exit` or `break` (apart for `switch-case` blocks) and return (except for returning, at the end of a function).

5. Submission

Project will be submitted by the Virtual Campus. A new task will be enabled, and only a single file with the source code will be accepted.

Deadline: **March 26, 23:55**