C++ 2026A - MTA - Exercises

Requirements and Guidelines

The exercises in the course require you to implement a Text Adventure World as a console application. The game consists of multiple interconnected rooms defined by external text files (note: files support would be required only from the 2nd exercise).
The player navigates between rooms, interacts with items, and solves in-game challenges. While the concept is inspired by classic text-adventure games, you must follow the specifications and requirements set forth in this document.

Environment and Submission Instructions:

The exercise should be implemented in Visual Studio 2022 or later, with standard C++ libraries and run on Windows with Console screen of standard size (80*25), using gotoxy for printing at a specific location on screen (X, Y), using _kbhit and _getch for getting input from the user without blocking, and Sleep for controlling the pace of the game.
(All the above would be explained to you during one of the lab sessions).
Submission is in MAMA, as a single zip file containing **only** the code and the vcxproj and sln files + readme.txt with IDs -- but without the DEBUG folder and without any compiled artifact. (Please **make sure to delete** hidden directories and unnecessary binary files).

Note that you MUST provide a readme.txt file that would contain the IDs of the students submitting the exercise. Even if there is only a single submitter, you still need to put your ID in the readme.txt file. Your readme file will also list any additional bonus features you've implemented, if you'd like to request bonus points for them.

Important notes on GenAI and using external code snippets:

- You may use ChatGPT or other GenAI tools, but remember that the responsibility for meeting the requirements and submitting working code is on you. Which means that you have to carefully read and understand any piece of code that you integrate into your submission.
- There are some existing C++ implementations for this project out there on the web. Please note that any implementation that you find would probably not fit exactly the requirements of this exercise. Adapting existing code for the purpose of the exercise would probably be much more complicated than implementing your own code.
- **Important:**
  If you do choose to rely on AI generated code or on existing project that you found on the web, or from any other source, you are required to add comments above any big portion of code that you didn't write and is integrated into your project (for any snippet of 5 lines of code or more). The comment should point at the source of the code (link to the web, a brief summary of the prompt used with the AI tool to get this code, etc.). It would be emphasized again that: (a) it is valid to integrate code that you didn't write! (b) you are still responsible for making sure the code works and fits its purpose. (c) you are required to have a clear comment indicating the origins of this code snippet. (d) slight adaptations to the code do not make it "yours", so even if you rearrange it a bit, change variable names etc., but it is still an external code of 5 or more lines, you are still required to add the origins comment as explained above.
- The instructions above are relevant also for integrating pieces of code that you got from the lab exercise or other resources provided by the course staff, etc. You should document that as well with a proper comment, indicating the origins of the code.

Exercise 1
In this exercise you will implement the basic components of the Text Adventure World console game.

You will determine how to use the console screen size (80×25 characters) for:
- Presenting the room map.
- Displaying player status (inventory, messages, hints, etc.).

Menu
The game shall have the following entry menu:
(1) Start a new game
(8) Present instructions and keys
(9) EXIT

When the game begins, there are two player characters, placed somewhere inside the first screen of the adventure world. The world is displayed in the console using text characters only. The user controls both players, which should work together to solve each screen.
At this stage of the course, saving or loading game data from files is not required (but allowed, if you want to go ahead with that), all world data may be initialized directly in the program.

Once the game starts, the player chooses a movement direction using the designated keys, as defined below. After a direction is selected, the character continues moving automatically in that direction without requiring additional key presses, until the STAY key is pressed (as defined below), or the player character reaches a wall, boundary, or any other item that should stop its movement. The "movement speed" is 1 unless otherwise stated, which means that for each "game cycle" the player moves 1 cell (unless speed becomes bigger than 1, as can seen below, or if the player is in STAY mode).

Input must be case-insensitive (both lowercase and uppercase letters should be accepted).

Keys:

	Player 1	Player 2
RIGHT	D	L
DOWN	X	M
LEFT	A	J
UP	W	I (i)
STAY	S	K
Dispose Element	E	O

Game Elements and Flow

The gameboard is consisted of the following elements, each taking one char on screen, unless otherwise stated:

Two players

Standing somewhere on screen, the two players are not competing with each other but rather working together. Each player can hold up to one collectible item.

Spring (one char or more)

A spring is composed of one or more spring characters, all aligned in the same direction (either a row or a column) and adjacent at one end to a wall. When a player moves over a spring, the spring characters temporarily “collapse” as the player advances toward the wall. Once the player reaches the wall, presses the STAY key, or attempts to change direction, the spring releases its stored energy and returns to its original length within a single game cycle. The release accelerates the player in the spring’s release direction. The magnitude of this acceleration equals the number of spring chars compressed, and it lasts for a number of game cycles equal to the square of that number. For example, if a spring has currently two chars being compressed, the player’s movement speed would become 2 in the release direction for 4 game cycles; If one char is compressed, the player’s movement speed stays 1 in the release direction, for 1 game cycle.
Once the player is “launched” by the spring, they cannot halt or move backward against the release direction (such attempts are ignored). However, they may move sideways at normal speed (1), and this lateral movement is added to the movement caused by the spring. During movement affected by a spring, all other rules still apply: the player does not “fly” over other elements, so any encounter with other elements must be properly managed. If a player moves under spring impact and collides with the other player, the second player would get the same speed and direction from the first player (ignoring previous movement direction).

Torch

A Torch is a collectible item, which can illuminate dark areas of the room (or entire dark screens) while the player carries it. Picking up a torch is optional, but certain rooms may be easier to navigate only when a torch is held. When a torch is taken by a player the player holds it till the player disposes of it (in order to take another element). When an element is disposed of, it stays at the position where it was disposed, till taken again. You can decide how to mark a room, or part of a room, as “dark,” so a torch will be required to illuminate it.

Key

Keys are another collectible item, used to open specific locked doors in the world. Each key may match one or more doors, depending on the game design. Some doors may require more than a single key to be opened, some may also require switch operations as described below. As all other collectible items, keys can be collected and disposed of by the user. When a key is used to open a door, the key disappears from the player’s inventory and from the world (i.e. it does not appear on screen again after use).

Obstacle (one char or more)

Obstacles block movement and cannot be walked through. As opposed to Walls, obstacles can be pushed to any direction, if not blocked by a wall. All directly adjacent chars that represent an obstacle belong to the same obstacle (not including diagonals). To push an obstacle there is a need to apply the required “force”, equal to the obstacle size.
The force a player can normally apply is 1.
If a player's movement is affected by a spring reaction his force is set according to its speed.
If the two players are adjacent and move to the same direction, the force they apply is the sum of the force of both. Remember that the two players can be both affected by a spring release, as described above.

On/Off Switch
Switches can be toggled ON or OFF by stepping on them.
A group of switches may be linked to a specific door. Only when the required switches are in the correct state (all ON, all OFF, or a specific combination set by the screen designer) does the connected door open (or be ready to be opened, with the proper keys).

Bomb
A Bomb is another collectible item. When disposed of, the bomb is activated, counting 5 game cycles and then it explodes, diminishing walls adjacent to it, in all directions including diagonal. It also diminishes any other object including obstacles, players, other objects, within a distance of 3 in all directions, including diagonally, ~~unless shielded by a wall~~*.

* Note: “unless shielded by a wall” became optional, you can interpret it yourself it you wish to implement it, and you can ask for a bonus if you do that!

Door
A Door can appear anywhere on board. It has a number (1 to 9) which is associated with the rules for opening it and the associated destination. Once a player steps on a door, if the door is open or can be opened with a key held by this player the player steps through the door. If the player holds a key that should be used by the door, but does not yet open it, the key will be automatically taken from the player, but the player would step through the door only if it is opened (thus, would stay near the door). The game will always follow the second player that leaves a room, so when there is still a player in the room we stay in the room, if the second player to leave a room goes through a door that leads to another room (not the same room that the first player went to), the game will continue with the second player.

Riddle
A Riddle can appear anywhere on board. Once a player tries to step into a riddle, an actual riddle associated with this position, as set by the screen designer, will appear on screen. The player cannot pass through without solving the riddle correctly. Wrong answers to a riddle will keep the player in her position near the riddle char but allow her to continue playing. Solving the riddle will position the player on the position where the riddle was, discarding the riddle from screen and the riddle char from its position. You can decide to reduce points or remaining life for wrong answers to a riddle.

Suggested Chars

Game Element	Suggested Char*
Players	$ and &
Torch	!
Bomb	@
Wall	W
Obstacle	*
Spring	#
Switch On/Off	/ \
Door	A digit 1-9
Key	K
Riddle	?

* Notes:

1. You may use other chars and document it both in your readme file and the game instructions screen, make sure to use reasonable chars.
2. You may use colors, but as explained below you should still allow the game to run in a non-colored mode.
3. It’s OK to have different chars presented on screen than those representing the element in memory or in file.

The game ends when both players reach a room that is marked as the last room.
Once a player reaches this room, the game will take us back to the screen where the other player is in. A player that reached the last room cannot step back or move in the last room.

Pausing a game

Pressing the ESC key during a game pauses the game. You should present a message on screen saying: “Game paused, press ESC again to continue or H to go back to the main menu”.
When the game is at a pause state, pressing ESC again would continue the game, with all moving game objects continuing their movement exactly as it was before pausing, as if the game hadn’t paused. Pressing in pause mode ‘H’ or ‘h’ (the letter H standing for “Home”) ends the current game returning to the main menu (see comments about avoiding a recursive flow).

Notes to be aware of:

1. Do not use the command ‘exit’:
   Using the command exit for finishing the program is **not allowed**. In general using ‘exit’ is a bad sign. You should finish the program by normally finishing main.
2. Make sure your program does not perform unnecessary bad recursive calls. For example, if your menu calls “game.run()” for starting a game, and then the game calls “menu.run()” for presenting the menu after the game ends, this sounds like a bad recursion that should be avoided.

Bonus Points:

You may be entitled for bonus points for additional features, such as adding colors or other nice features. Note: there will not be any bonus for music or any feature that requires additional binary files to be part of your submission! Adding large required binary files to your submissions may subtract points!

Important notes for getting bonus:

1. If you decided to add colors (as a bonus feature) please add an option in the menu to run your game with or without colors (the default can be to use colors, but the menu shall allow a switch between Colors / No Colors) - to allow proper check of your exercise in case your color selection would not be convenient for our eyes. The game MUST work properly in the No Colors selection.

   Note: If you do want to support colors, use the following way for doing that:

   ```
   #include <windows.h>
   ...
   HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
   SetConsoleTextAttribute(hStdOut, FOREGROUND_RED);
   std::cout << "Hello" << std::endl;
   ```

   For color constants list see: Console Screen Buffers - Windows Console
   See also: Colorizing text in the console with C++ - Stack Overflow

   Please note that alternative methods of coloring text suggested on the web, such as using color codes with system commands, may not always work. It is advisable to avoid these methods and, instead, use the approach mentioned above if you intend to add colors.

2. To get bonus points, you must indicate inside a *bonus.txt* file the bonus additions that you implemented.

Note:
In exercise 1 you are not required to support all items!
You must support: two players, walls, keys, doors~~, spring~~.
You can select to support only ~~two~~ three from the following:
bomb, switch, obstacle, riddle, torch, spring. (If the torch is not supported, no need to support dark rooms).

Important!
Your submission should include a challenge that goes through 2 screens + a final end screen (which has no items).
Your grade will take into account the quality of the challenge your game provides.

Self Decisions:
Any decision that you have to make, which does not have any clear guidance in the requirements, you can take a reasonable decision as long as it doesn't contradict any of the requirements or make the game naive or too restricted.

In case you have a question for which you don't have an answer and you don't want to make an assumption or you feel the assumption might be wrong or contradicts other requirements, please ask in the Exercise Forum in Mama.

Important – Important!!!
We will explain to you in one of the lab sessions how to use gotoxy for printing at a specific location on screen (X, Y), _kbhit and _getch for getting input from the user without blocking, and Sleep for controlling the pace of the game.
BUT, you don't have to wait for that, you can already watch now Keren Kalif explaining that in this great video tutorial: C++ תכנות מונחה עצמים | קרן כליף | משחק הנחשים
The proper code would be published.

Exercise 2
In this exercise you will implement the following additions to your game:

Support for all elements!
Add support for the elements you did not support in exercise 1.

Loading Screens from files
The game would look for files in the working directory, with the names *adv-world*.screen* these files would be loaded in lexicographical order (i.e. *adv-world_01.screen* before *adv-world_02.screen* etc.). The files are text files (NOT binary files).

The screen file should be a text file representing the screen, with the characters as set in exercise 1, with proper adjustments that you should propose (i.e. for marking a dark room or a dark part of a room, for connecting a door to a specific room, etc.). Each screen file will have also the following additional char:

L	Indicating the legend top-left position, where score, inventory per player, lives and other relevant information shall be presented. It is the responsibility of the screen designer (not of your program) to make sure that the L is placed in a position not accessible by the game objects. You may assume that the screen designer follows this instruction. The size of the actual printed legend shall be not more than 3 lines height * 20 characters.

You may add any info that you find relevant into the screen file.

You MUST submit 3 screens with your exercise! (not including the “final” screen, which actually does not require a file, probably).

If there are no files, a proper message would be presented when trying to start a new game.

Riddles file, named *riddles.txt*. You should decide on the format and the connection between riddles in the file and the riddles in the screen files. Each riddle should have the expected solution in the file. In case there can be more than a single solution you may want to list several possible solutions.

You MUST provide a riddles file.

Important!
Your grade will take into account the quality of the challenge your files provide.

Score, Lives
Add score and lives to your game. You should decide about the formula / triggers for both.

Error Handling
The program should never crash. In case of problems in files, always prefer overcoming the problem and allowing using the file. However if a file cannot be used, provide a proper message to the user, clearly pointing at the problem, and return from main.

Note
Where appropriate, change your original code, to use new materials that we learned.

Bonus
You may add additional elements and ask for a bonus in the *bonus.txt* file.

Exercise 3

Part 1 - Saving game state to file

When ESC is pressed, add an option to hit S or s for saving game state.
Add an option in the menu to start a game from a previous state. When this option is selected all previous saved states would be presented and the user can select which one to use (you should decide how to present them, e.g. by date and time, or asking the user for a name when saving state and using this name for loading it).

Part 2 - Saving and Reading Game Files

You should also implement an option to run a game from files and to record a game into files, mainly for testing. This would work as following:

- There would be a text file with a list of steps, in a structure of your choice. The structure shall be concise, i.e. shall include only information that cannot be deduced from previous file info and the game rules. Do not add to the file information that can be deduced from previous data in file, for example there isn't a need to save into the file the players' positions, only the change of directions. It is important to save each "step" with a "time" (game cycle) so it can be reproduced exactly as it happened. If you added anything that is not deterministic, you can save to the file a random generator seed, using the same seed can guarantee the same random numbers are used when the game is played from file.
- The name of the file shall be *adv-world.steps* (there isn't a need to support more than one file, the user can move the file to another directory if they want to record another game and keep this one).
- The file corresponds specifically to certain game screen files, it is advisable to write to file the name of the game screen files to make sure we play the same files when replaying a game.
- There would be a file with the expected result for the game, by the name: *adv-world.result*, the file shall include the following information:
    - points of time* when a player moved to another screen and to which screen
    - points of time* when a player lost a life
    - points of time* when a player got a riddle, the riddle, the answer and whether the answer is correct
    - points of time* when game ended and score gained

* point of time = game's time counter, not actual time, each iteration can be counted as "1 time point"

You have the freedom to decide on the exact format of the files but without changing the above info or adding unnecessary additional info.

You should provide at least 3 examples, with the steps and results files, in addition to at least three screen files and a riddle file.

You should add a newly dedicated readme file called **files_format.txt** explaining your *steps* and *results* file formats, to allow creation of new files or update of existing files.

The game shall be able to load and save your files!
All files shall be retrieved / saved from /to the current working directory.

Running the option for loading or saving game files would be done from the command line with the following parameters:

```
adv-world.exe -load|-save [-silent]
```

The *-load/-save* option is for deciding whether the run is for saving files while playing (based on user input) or for loading files and playing the game from the files (ignoring any user input).

In the -load mode there is NO menu, just run the loaded game as is, **and finish!** Also you should ignore any user input, including ESC - there is no support for ESC in load mode!

In the -save mode there is a menu and the game is the same as in Ex2, except that files are saved. Note that each new game overrides the files of the previous game, i.e. we always keep the last game saved (you can always take the files and copy them to another folder manually if you wish).

The ***-silent*** option is relevant only for load, if it is provided in save mode you should ignore it, if it is provided in load mode, then the game shall run without printing to screen and just testing that the actual result corresponds to the expected result, with a proper output to screen (test passed or failed, and if failed - what failed). In silent mode you should avoid any unnecessary sleep, the game should run as fast as possible. Without -silent, the loaded game would be presented to screen, with some smaller sleep values than in a regular game.

Note: you should still support running your game in “simple” mode, without any command line parameters, as in Ex2: `adv-world.exe`
In which case it will **not save or load files** and would behave as in Ex2 (except for the other addition of Ex3 above, in part 1).

Exercise 4
Exercise 4 is a separate exercise built as a rehearsal for the exam.
It would be published separately.