VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

THE MATCHING GAME PROJECT C++

Course: CSC10002 - Programming Techniques

Students (22CLC10):
Van Ba Duc Kien (22127218)
Vo Viet Long (22127252)

Instructors:
Dr. Nguyen Thanh Phuong
MSc. Bui Huy Thong
MSc. Nguyen Ngoc Thao
MSc. Tran Thao Nhi

Ngày 15 tháng 4 năm 2023

Contents

1) Introduction

1.1) Game overview

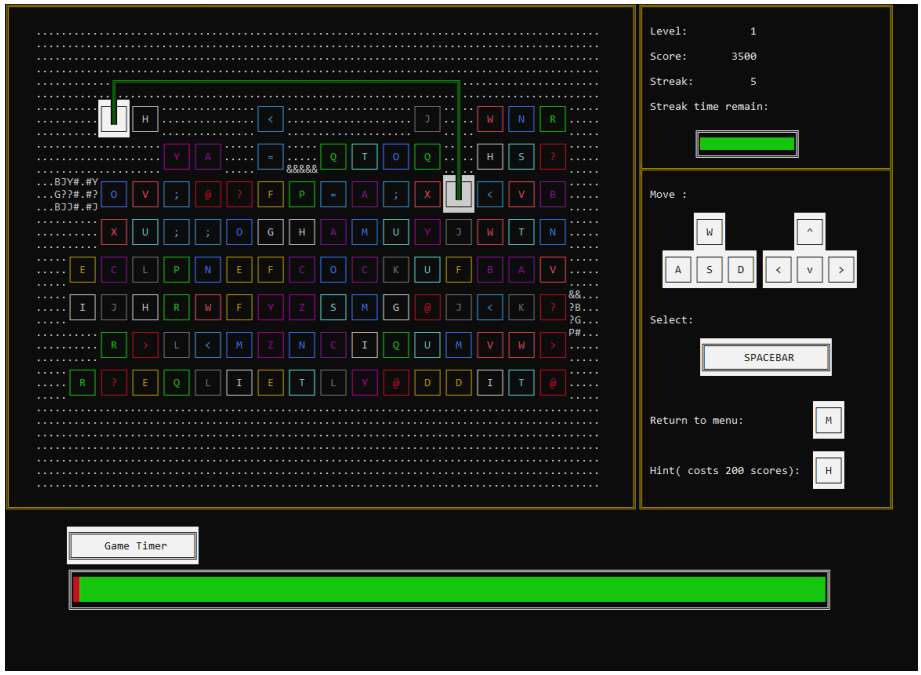
The **Matching Pokemon Game** (also known as Classic Pikachu Kawai or Pikachu Classic or Onet Connect Animal) focuses on a board of multiple cells, each of which presents a Pokemon. The game requires players to find and match pairs of Pokemon which need to satisfy these **following rules**:

- The chosen pair must contain the same Pokemon.
- The path that connects the chosen pair consists of less or equal to 3 lines, each line must be parallel with either side of the board and the lines can't trespass any Pokemon.
- In case there are no possible pairs to be matched, the cells in which contain Pokemons will be randomly shuffled.
- The player wins if there was no Pokemon left on the board.
- The player lose if the time ran out.

Onet Connect Animal (from google.com)

1.2) Project overview

In this project, we will create a simplified version of the game with our "homemade" matching algorithm and some extra features using the C++ programming language with no external graphical library.



Our implementation of the game

2) Project setups and tools

2.1) Coding environment

- Operating System: **Windows 11**
- Compiler: **GCC** for **C++ 11** (GCC version: 12.2.0)
- Code Editor/ IDE: **Visual Studio Code** 1.77.1, **Code::Blocks** 20.03
- Other tools:
  - Version control: **Git** and **Github**
  - **Overleaf** and **Typst** for writing report with LaTeX and Typst
  - **draw.io** for drawing case diagram, linked list visualization.
  - **HexEd.it** for openning binary file in hexadecimal mode.

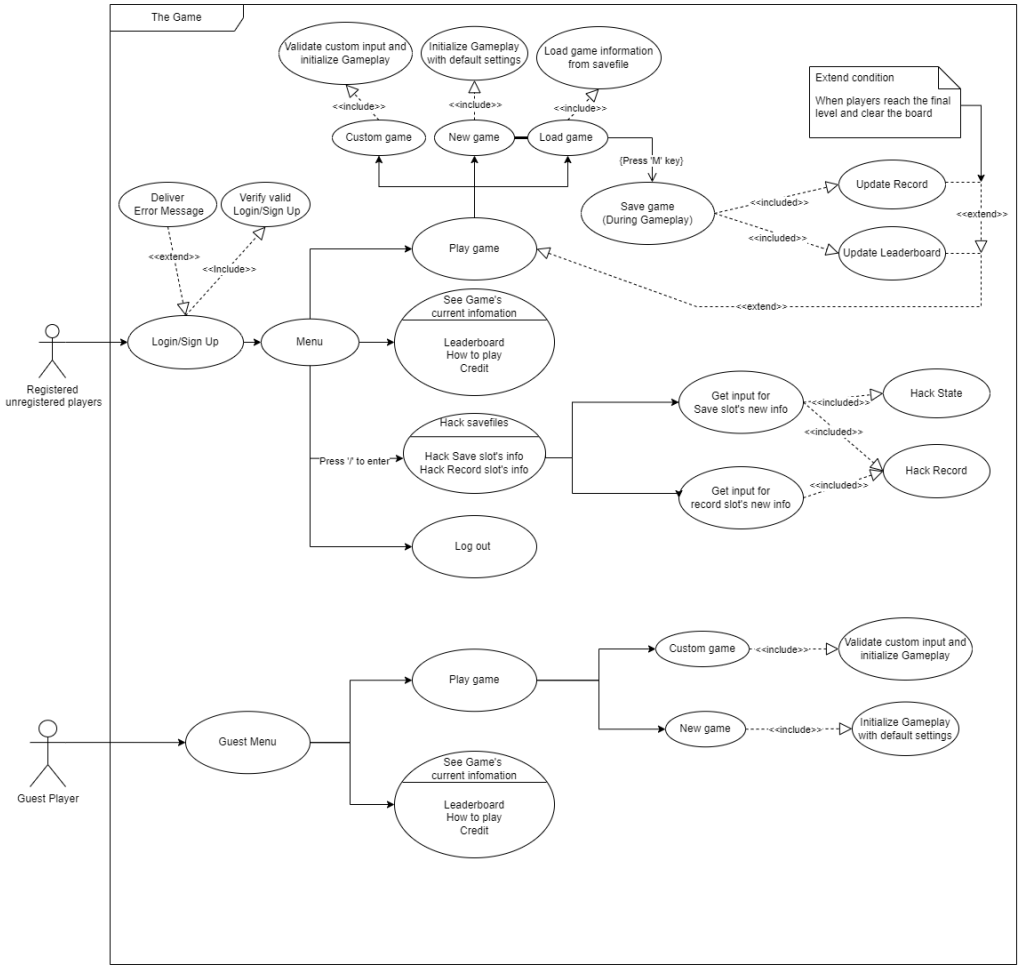
2.2) System requirements and game setups

- Operating System: **Windows** (because we use the library windows.h)
- Compiler: **G++ or GCC version 12.2.0** (if you do not have GCC 12.2.0, follow this tutorial to download: "https://code.visualstudio.com/docs/cpp/config-mingw")
- The reason that G++ or GCC 12.2.0 is needed because one of us used vector in c++ for the first time and he write size(<vector_name>) instead of <vector_name>.size and some older version of G++ do not work.

- Console: use normal cmd, do not use terminal this is because of the windows.h library can not work with terminal.
- How to download and run the game:
  - Install the game from our github page: "https://github.com/Keineik/ClassicVaporeonKawai"
  - Unzip the file.
  - Navigate into Project folder then run the **main.bat** file.
  - Enjoy the game.

3) How the game works and demonstrations

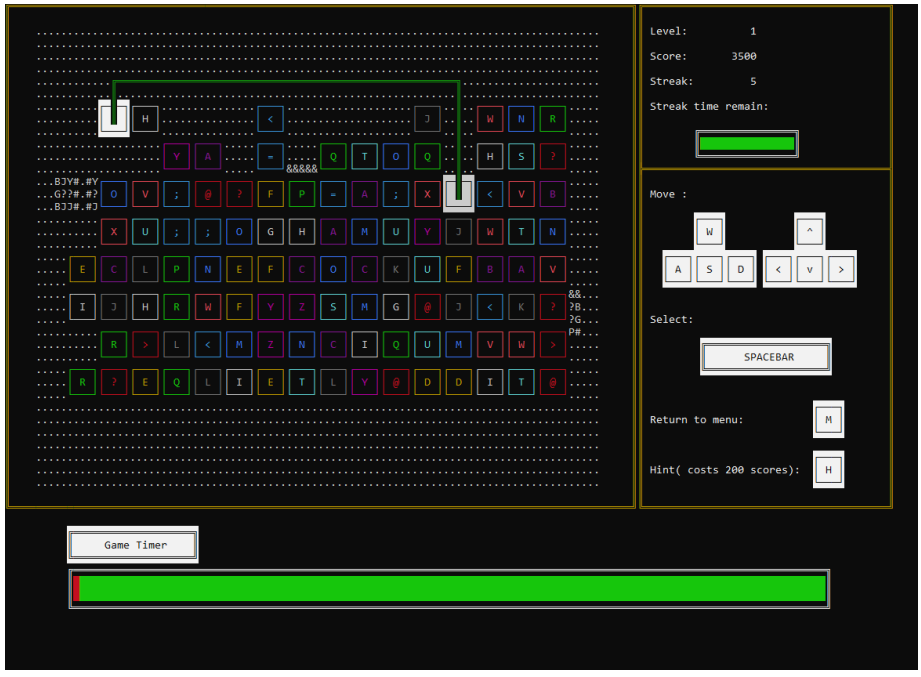
- This is a case diagram to show how the game works:



Case diagram of how the game work

:

The first game menu to be encounter



A captured Gameplay image

- Tutorial video link: [“https://www.youtube.com/watch?v=oO2WpKDn7Mg”](https://www.youtube.com/watch?v=oO2WpKDn7Mg)

4) Code descriptions

4.1) Files management

Files management is essential in coding, it improves the modularity of the project which improves the readability, ease of changing/ updating the code when you fix bugs or add new features. This is all of our files and the contents of each file.

- **header.h**: This file includes all of the necessary libraries, variables and defines some useful things that are used throughout the entire game.
- **[filename].h**: These files contain all the necessary libraries, define structures, declare functions all of which are used by **[filename].cpp** files.
- **gamelogic.cpp**: This file contains the main logic and mechanisms behind the game.
- **account.cpp**: This file contains all interactions with user account including reading/writing binary save file, signing up, login, logout, saving/loading playing game from save slots and updating leaderboard.
- **drawscreen.cpp**: This file contains the functions to draw the UI and update the screen when needed using the logic in all of the functions in the files above.
- **main.cpp**: This file will include all of the previous files and combine them to make the full game.

Supporting files and folders:

- **savefile.bin**: Stores users' data.
- **leaderboard.bin**: Stores leaderboard data.
- **assets** folder:
  - **txtimages** folder: contains images for the game UI and background images.
  - **sounds** folder: contains sound files for sound effects.

4.2) Standard features

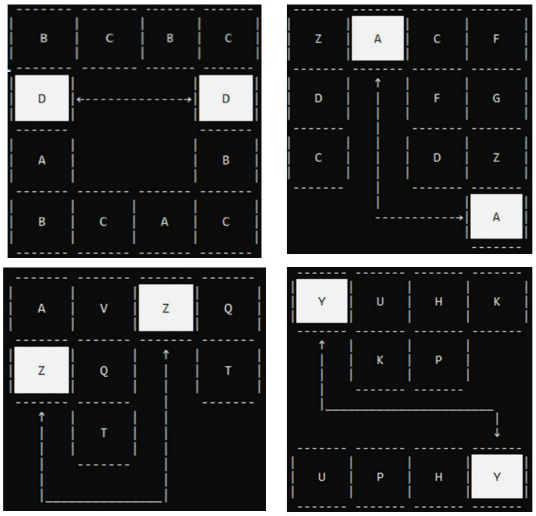
4.2.1) Board initialization

- Let's M, N consecutively be the number of rows and columns of the board
  - In classic mode, $M = 8, N = 16$
  - In custom mode, the player can change the value of M, N however they want, as long as these following conditions are satisfied: $M.N \leq 8.16$ and $M.N$ is divisible by 2.
- Each Pokemon will be represented by an ASCII code range from 59 (';') to 90 ('Z') so there will be at most 32 Pokemons, each Pokemon will appear 4 times at most. From these rules, we thought of the following algorithm for generating the board:

- Board initialization is handled by the **initializeBoard()** function in the board manipulation section in the gamelogic.cpp file, which follows these steps:
  1. Dynamically allocate memory according to M and N, we allocated $M+2$ rows and $N+2$ columns including the borders of the board to change the indexes of the cells to $[1, M][1, N]$ instead of $[0, M-1][0, N-1]$ to have a more natural accessibility to the board's cells and easier match checking later.
  2. Create two integer variables: poke = 59 (ASCII code of ';') and count = 0, the variable poke represents the ASCII code for the cells' value.
  3. Iterate over every cell, for each cell, assign poke to the cell's value, then increment count by 1 and if count is divisible by 2, increment poke by 1 and if poke = 91, reset poke to 59.
  4. Shuffle the board. (for more details, see 4.5.1)
- When the game ended or before initializing new board, use the function **deleteBoard()** to delete to assigned memory the the 2D pointer array

4.2.2) Check matching pair

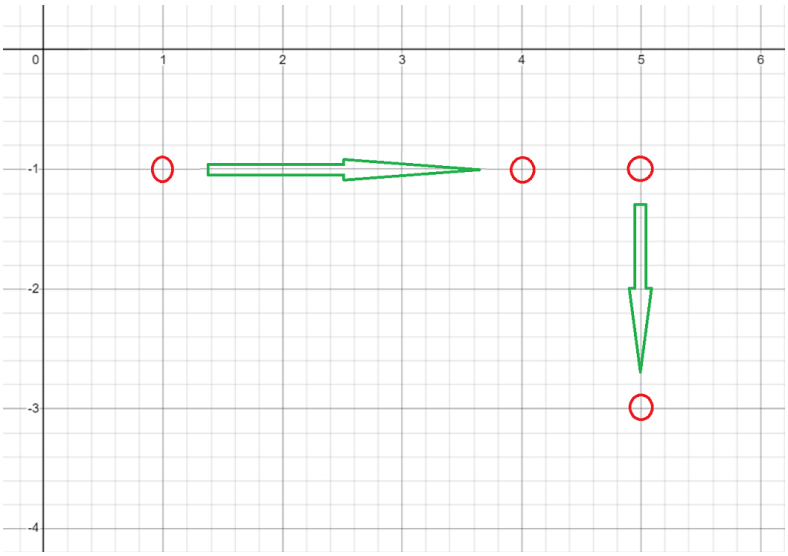
The check matching pair part is the soul of the game itself. According to the project description, matching patern must be one of these motifs:



I, L, U, Z matching patern examples

After much thoughts and discussions, we designed our own algorithm to solve this problem. Here is the main steps that make up the backbone of our algorithm (this part is implemented in the **isLegalMove()**) function):

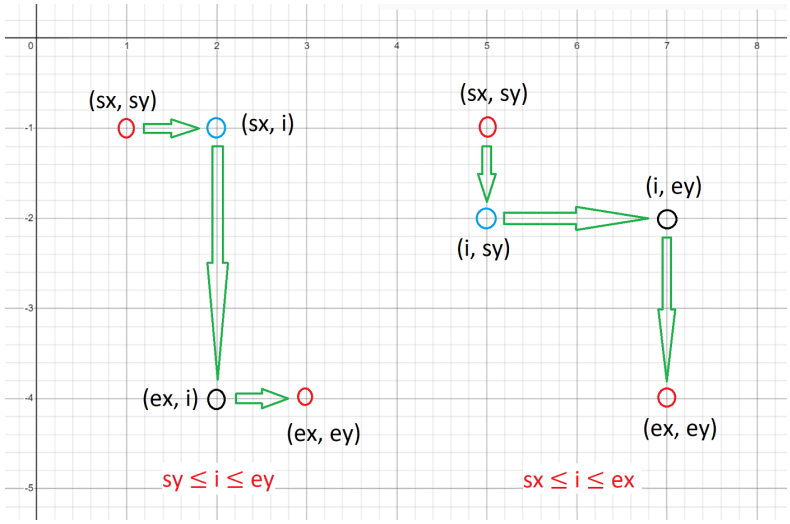
1. Preprocessing:
   - Check if either of the two chosen cells are a blank space.
   - Check if the two chosen cells have the same value.
   - Check if two chosen cells have the same coordinate.
2. Check I matching patern (implemented in the **checkLine()** function):
   - Check if the two chosen cells are on the same line or column.
   - Vertically or horizontally check every other cells between them, if they are all blank spaces, return true.



Two cases that checkLine() function checks

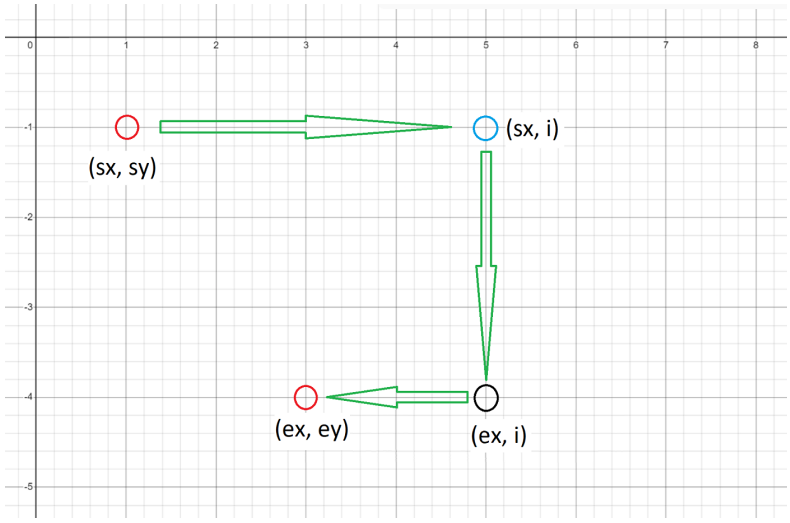
3. Check L, Z and U matching paterns baseline ideas:
   - Let $(\text{sx}, \text{sy}), (\text{ex}, \text{ey})$ consecutively be the coordinates of start point and end point. Let $(\text{mx1}, \text{my1})$ and $(\text{mx2}, \text{my2})$ be two running middle points.
   - Checking the L, Z and U matching paterns can be broken down into checking three lines: (sx, sy) to $(\text{mx1}, \text{my1})$, $(\text{mx1}, \text{my1})$ to $(\text{mx2}, \text{my2})$ and $(\text{mx2}, \text{my2})$ to (ex, ey) (use the **checkLine()** function in the previous step)
4. Check L and Z matching patern in details (implemented in the **checkSmallRect()** function)
   - In case the patern is a vertical Z, the two running middle points will create a vertical line, we can change the format of those two points to (sx, i) and (ex, i) with $\min(\text{sy}, \text{ey}) \leq i \leq \max(\text{sy}, \text{ey})$.

- In case the patern is a horizontal Z, the two running middle points will create a horizontal line, we can change the format of those two points to (i, sy) and (i, ey) with $\min(\text{sx}, \text{ex}) \leq i \leq \max(\text{sx}, \text{ex})$.
- The L patern are special cases of the Z patern when two middle points have the same coordinates.



Two cases that checkLine() function checks

5. Check U matching patern in details (implemented in the checkBigRect() function)
   - Depending on which direction the rectangle (or the bottom part of the letter U) expands, we can change the format of the two middle points accordingly:
     - Expands horizontally to the right:
       - (sx, i) and (ex, i) with $\max(\text{sy}, \text{ey}) \leq i \leq \text{boardLength} + 1$
     - Expands horizontally to the left:
       - (sx, i) and (ex, i) with $0 \leq i \leq \min(\text{sy}, \text{ey})$
     - Expands vertically to the top:
       - (i, sy) and (i, ey) with $0 \leq i \leq \min(\text{sx}, \text{ex})$
     - Expands vertically to the bottom:
       - (i, sy) and (i, ey) with $\max(\text{sx}, \text{ex}) \leq i \leq \text{boardHeight} + 1$



Example of a case where the rectangle expands horizontally to the right

4.2.3) Game finish verification

- According to the requirements in the project file, the game will end when one of the following conditions is fulfilled:
  - There are no cells left (win)
  - There are no valid matching pairs left
- For the first condition (implemented in the **isWin()** function), we iterate through every cell on the board, if every cell is blank, stop the game.
- For the second condition (implemented in the **isPlayable()** function), we iterate through every cell on the board, for every cell, iterate through every other cells on the board and use the function **isLegalMove()** (See 4.2.2 for more details) to check if there are any legal moves left. If there are no legal moves, stop the game.
- Because in the original game, when the second condition is met but the first condition is not, the contents of all the remaining cells will be shuffled. When this happens, we will not end the game but shuffle the board just like in the original game (See 4.5.1 for more details) and this will be one of our extra features.

4.3) Advanced features

4.3.1) Color and sound effects

- **Color**
  - For the coloring in this game, we try several external graphic libraries (e.g. SFML and SDL2) to liven up the overall look of the gameplay.

- However, due to a lack of confirmation about using an unavailable library, which requires additional installation, using functions based on <Windows.h> library is our final choice and the following **SetColor(int x)** function is what we implemented and took the most advantage of.
- **SetColor(int x)** takes a value of type integer, then the background color and characters color in console are defined respectively by the quotient and the remainder of the division between x and *16*. The value for each color can be refered in the table below(Note: This table presents these values in hexadecimal)

Color id	Color	Color id	Color
1	Blue	9	Light Blue
2	Green	0	Black
3	Aqua	A	Light Green
4	Red	B	Light Aqua
5	Purple	C	Light Red
6	Yellow	D	Light Purple
7	White	E	Light Yellow
8	Gray	F	Bright White

Color id table. Credit: GeeksforGeeks

- **Sound effects**
  - To play sound in console and implement that idea in C++, we opt for **PlaySound()** function to play sound from **.wav** file, which belongs to <winmm.lib> (this library requires another library called <mmsystem.h> that is not included in <bits/stdc++.h>).
  - **PlaySound()** takes many different arguments but in most of our cases, this function is used as this example:
  - PlaySound(TEXT(filepath),NULL,SND_ASYNC|SND_LOOP);
    - **filepath** is the path of directory of the audio file that we want to play.
    - After **filepath**, **NULL**, **SND_ASYNC** and **SND_LOOP** are the flags to tell how PlaySound() will play the sound during throughout the program. In this case, flag **SND_ASYNC** and flag **SND_LOOP** are telling the program to play and loop the sound from **filepath** asynchronously but do not stop the other statements after the call of PlaySound() function from being executed.
  - However, PlaySound() does not support playing multiple sounds simultaneously so we have to sacrifice playing theme musics over sound effects or the opposite.

4.3.2) Visual effects

Before digging into further details of how we implement visual effects, these are some essential functions in used:

- **gotoxy(int x, int y)**: This is a function from <windows.h> to manipulate the console cursor's position in the console. Regularly, after a *cout* statement, the cursor moves forward one character horizontally and is set place at the left most position after an **newline** character, and there are not many conventional ways to set it back freel, this is when **gotoxy(int x, int y)** comes in handy.
- **resizeConsole(int posx, int posy, int width, int height)**: This function changes the width and height **in pixels** of the window console to prevent loss of game content and places the window at position of *(x,y)* on the user's screen.
- **clearCanvas(int x, int y, int width, int height)**: Unlike its name, this function actually prints **white space characters** on a rectangular area of width and height, starting from *(x,y)*.
- **clearScreen()**: Another function from <windows.h>. This clears the entire console window, its ability resemble that of system("cls") statement. Moreover we chose this over system("cls") due to this following artical **Why system() is evil**

- **calculateBoardPosX() and calculateBoardPosY()**: These two functions calculate the position of the very first element of the 2D array that contains Pokes and return two integers known as coordinates, these coordinates also act as the starting position of the entire board.
- **calculateCellPosX() and calculateCellPosY()**: We reuse the previously mentioned coordinates to calculate each cell *(x,y)* starting coordinates, these two functions also return two intergers.
- **drawBox(int x, int y, int w, int h, int color, string s)**: Draw a box of size (width $\cdot$ height) with a string insdide
- **drawCell(int x, int y, int w, int h, int color, char c)**: A similar function like drawBox, but it is used specifically for drawing cells of board, we make it another function for easy code reading and other purposes.
- **drawBar(int startx, int endx, int posy)**: Draw a bar from *(startx, posy)* to *(endx, posy)*
- **drawColumn(int starty, int endy, int posx)**: Draw a column from *(posx, starty)* to *(posx,endy)*

Those are the basic functions that are used repeatedly in many other complex functions such as (to advoid wordy sentences, we do not show the functions' parameters this time):

- **printBoard()**: Draw the entire board on the console window.
- **drawPath()**: Draw legal move based on the information from function **isLegalMove()** (see 4.2.2 Check matching pair).

And some of those advanced function are create to enhance game performance.

- **clearVfx()**: This function is made to particularly remove every output of the **drawPath()** function, and replace the cleared areas with positionally corresponding characters of the background.
- **updateBoard()**: This function demonstrate every change inside the game screen during game play (e.g. changing the game cursor position, demonstrating chosen cells, redrawing wrong choices of matches), and functions **clearVfx()** and **drawPath()**.
- **redrawCol() and redrawRow()**: These are level-based functions, used only when there are presences of **shiftColUp(), shiftColDown(), shiftRowLeft() and shiftRowRight()** functions during gameplay. Their names say it all, they help us to redraw up to 2 columns or rows instead of redrawing the entire board.

The reason for creating these functions is:

- After many research, we have realized that window console and C++'s built-in libraries do not support game layers, if there is something new shown on

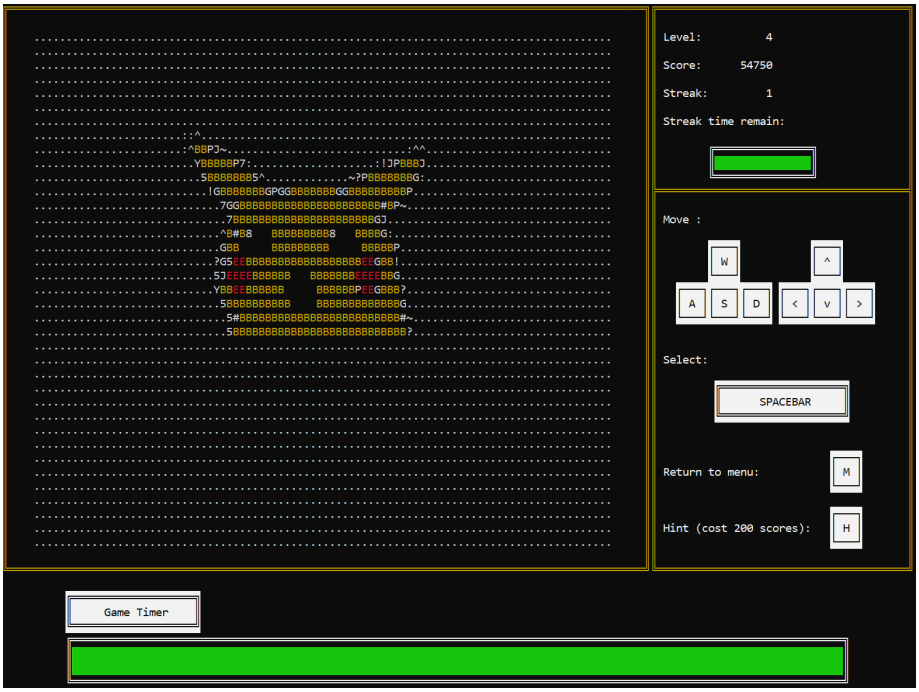
the console, it will overwrite the old contents of that area instead of just overlay it.

- Without using other libraries that support game layers like SDL2 or SFML, we had no other choice other than leading to us constantly redrawing the entire board using **printBoard()** after every single change being made in the very first versions of the game. Thus there was significant latency between the user interactions and information shown on screen, resulting in serious inconvieniences and high resource consumption.
- Therefore, we proceeded to create **clearVfx()** and **updateBoard()** in order to only show the differences created by the players and the systems, the latency was then decreased beyond out expectations with little of in-game delay and smoother operation of game system.
- The old **updateBoard()** worked well until we implemented more level-based functions, which shifted the entire board whenever there was a matches. This kind of function caused more cells to be updated than before, making **clearVfx() and updateBoard()** out-dated. To neutralize that problem, **redrawCol() and redrawRow()** were added into **updateBoard()** and they have been working as well as intended since then.

4.3.3) Background

"You can design anything for a background. Then, when a matching pair disappears, the background content corresponds to those emptied cells". As quoted from the project description, a background hidden under a board of cells is recommended and we also love to do a little bit of "happy features", so these are some functions used to implement that idea:

- **drawImage()**: Draw image (a combination of ASCII characters) with colors, from **.txt** file. Usually, this is used to add some colorful detail onto the menu, but then we also reused it to make an outro of every win for our precious players.
- **drawBackground()**: Another version of the function above, however, it only draw all white image as the background.
- **exposeBackground()**: This is a function modified from **clearCanvas()** (mentioned in 4.3.2). Instead of draw an area with *whitespace character*, it uses positionally corresponding characters of the background. This function is also included in the **updateBoard()** function.



Background with colors appear when you complete the level

4.3.4) Move suggestion

This part of the game is implemented in the moveSuggestion() function. The idea of this function is simple:

- Iterate through every cell on the board, for each cell, iterate through every other cells and use the function isLegalMove() (see 4.2.2 Check matching pair).
- If a legal move is found, the function returns a pair of coordinates of the legal move. This function will always return a legal move because if there are no moves left, it should already be handled by the isPlayable() function (see 4.2.3 Game finish verification).

In the game, press 'H' to use this feature, when used, the game will automatically select both of the cells for you, you just need to press space to confirm your move.

4.3.5) User accounts

Because we implement the savefile "hacking" feature, our structures for user account is the same as the project file with some tweaks but still fit the predefined ones. We added some more variables to save more information about the saved games. Here are the explanation of all the variables that we used:

- Strings of **name** and **password** are self-explanatory.

- An array of 5 **record**: according to the project requirements file, this is a list of **sorted best records**. Each record element contains a **date** and **points**.
- An array of 5 **state**: according to the project requirements file, this is 5 saved slots of a signed up user. Each state element contains:
  - **p, q, p_, q_**: consecutively is the height, the length of the board and the coordinate of the current cursor.
  - An array of characters representing the **board**: a 2D representation of the board when saved.
  - **file_background**: link to the background file.
  - **level**, **points**, **date** are self explanatory, **time** is the remaining time that the player has.

All the functions that are in this part include:
- **readBinFile()**: This function will read all the data from the binary file and put all the data into a vector of struct at the start of the game.
- **writeBinFile()**: This function will write all the data from every user to the binary file. This function has a sub-function which is xorCstr() to mask all of the characters before writing to binary file for more security.
- **signUp()**: This function allows user to create new accounts, it also checks for existing username, invalid username using sequential search and prevent users from creating faulty accounts.
- **login()**: This function will find the username using linear search, check the password and will obtain all the data of that user from the vector of struct when successfully signed in.
- **logout()**: Sign out of current account.
- **saveGame()**: Find the current user's data in the vector of struct, save the current game state to the save slot desired by the user. This function also updates the best records of the user.
- **loadGame()**: Find the current user's data in the vector of struct, load the game from the save slot to continue playing.
- **updateRecord()**: sort the player's best records increasingly, update the best records with the current game state.

4.3.6) Leaderboard

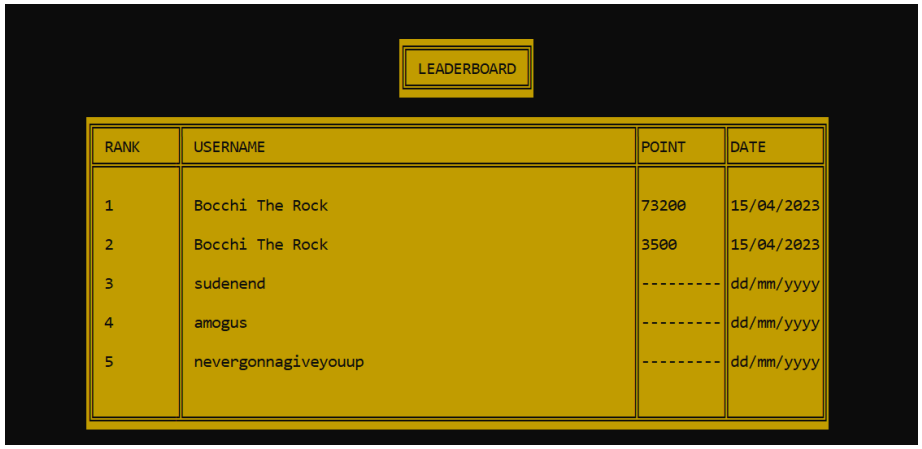
Score in this game is calculated by function **updateScore()** accoring to the following formula:
- During a gameplay: **score** += **streak** $*$ **streakScore**
- After a cleared Level: **score** += **timeRemain** $*$ **50**
- **score** is the score players gain
- **streak** is the number of matches in a row that players get right, the higher the **streak**, the more points players can get. The **streak** increases by one

unit everytime players get another match right until it reaches 5. However, it resets when **streakTimeRemain** comes to 0 (please read part 4.5.2 for more details).

- **streakScore** is how much a unit of streak worth, we set it at 100 points.
- **timeRemain** is how much time left after the players clear the entire board, the higher the **timeRemain**, the higher the bonus points players can get after each round. In our game, every second left is equal to 50 bonus points.

For this section, we use leaderboard.bin file to store the leaderboard's data. We use binary file to make it harder for people to change the contents of the leaderboard. Our leaderboard feature includes:

- Struct highScore contains points, date, name.
- The leaderboard is represented as an array of highScore elements.
- **readLeaderboardFile()** function: reads the data from leaderboard.bin file.
- **writeLeaderboardFile()** function: write the data from the leaderboard array to the leaderboard.bin file.
- **updateLeaderBoard()**: sorts the leaderboard array, if the current game's point is higher than the record with the lowest points on the leaderboard, swap them and sort the leaderboard one more time.



Leaderboard is filled with random name if there are less than five record

4.4) Extra advanced features

4.4.1) Stages difficulty increase

4.4.1.1) 2D Array Pointer implementation

The implementation idea of sliding neighboring cells into the newly emptied spaces in a particular direction is considered a trivial task.

- Slide neighboring cells into the newly emptied spaces is the same as moving all the blank cells to the end of the opposite direction.
- From the above idea, we use a loop that is similar to the bubble sort inner loop to make one blank cell go to the end of the opposite direction.
- shiftColUp(), shiftColDown(), shiftRowLeft(), shiftRowRight() are four functions that do the same thing as their name suggested, they are implemented using the same idea above with column/row index as parameter.
- For each function calls, only one blank cell is moved to the right position, when two cells are matched, we just need to call the function with the column/row parameter twice.

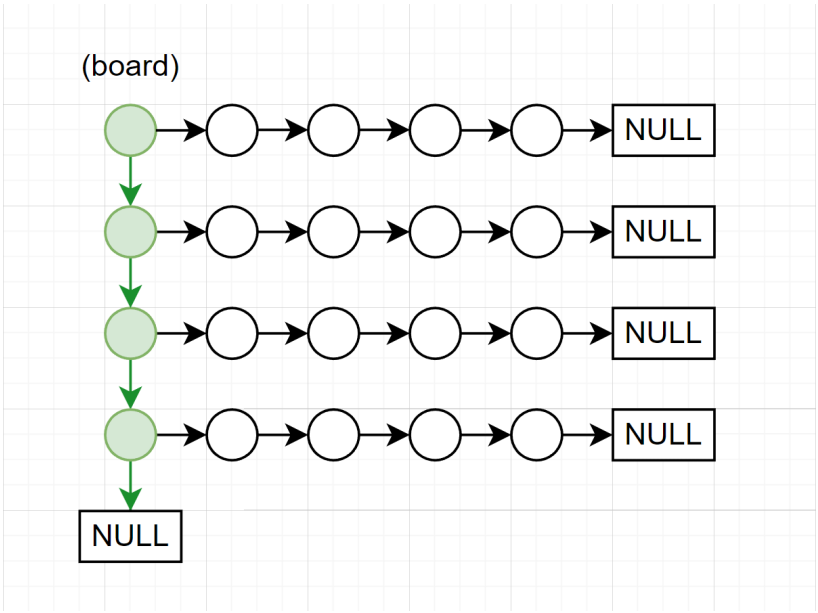
4.4.1.2) Linked list implementation

Our linked list implementation version of the game is in the **"LinkedList Version"** folder

For the linked list implementation, we represent our board as a linked list of linked lists. As for why we chose this representation, the project clearly stated "you will have to represent your game board content in a Linkedlist" so using things like an array of linked lists is just unacceptable and it is just a horrendous disgrace to this subject 🐧.

Let's get to the details about our linked list implementation:

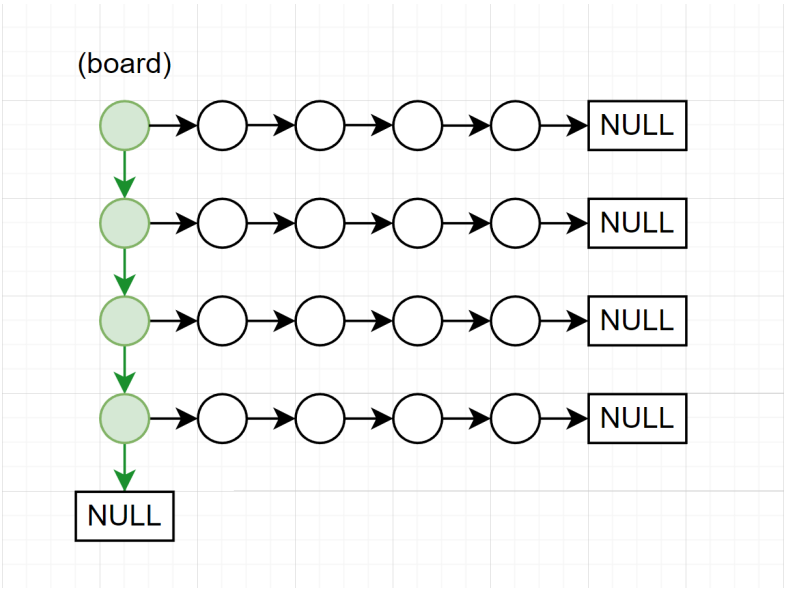
- For ease of writing algorithms, let M, N consecutively be the height and length of the board. Let (x, y) be the coordinate of the cell which is needed to be accessed.
- We use two kinds of node for our linked list:
  - struct Node: your typical linked list's node with an integer variable to store data and a pointer which points to the next node. (These nodes are represented by the black circles in the pictures below)
  - struct LinkedList: this is our special kind of node, these nodes consist of a pointer which points to the next LinkedList node, a pointer which points to the first node of the linked list and an integer to store the current size of that linked list. (These special nodes are represented by the green circles in the pictures below)



Visualization of our board structure using linked lists

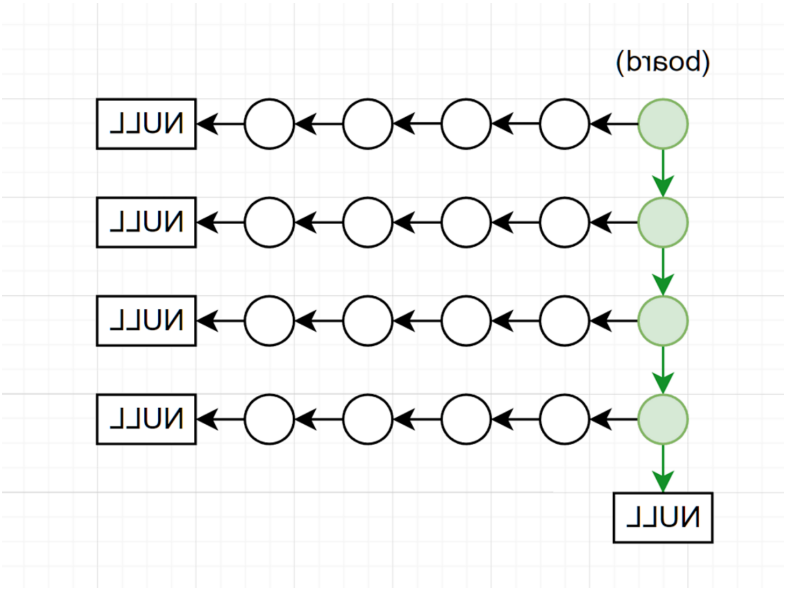
- To initialize the board (implemented in the **initializeBoard()** function), follow these steps:
  - Add a LinkedList node to the head of the linked list of LinkedList.
  - Create a linked list of N nodes, let the Linkedlist node created in the previous step points to the first Node of this linked list.
  - Repeat two previous steps M times.
  - Let "board" be a LinkedList pointer that points to the first LinkedList node of the linked list of LinkedList.
  - The algorithm for generating the nodes' data is the same as the algorithm in 4.2.1)
- Deleting the board is necessary to prevent memory leakage, we implemented this in the **deleteBoard()** function.
- Shuffling algorithm is the same as 4.5.1 part, just change the accessing part from 2D array to accessing linked list, which will be covered in the next step.
- To access a node, we implemented two functions with (x, y) as parameters: **getData()** and **changeData()** both follow the same main ideas:
  - Let curLL is a LinkedList pointer that points to the current LinkedList node that we are at. Let curNode is a Node pointer that points to the current node that we are at.
  - Set curLL to board (the first LinkedList node). Use a loop to move curLL to the x LinkedList node.

    - Set curNode to the node that LinkedList points to (the first node). Use a loop to move curNode to the y node.
    - Return the data or change the data.
- The logic of the isWin(), isPlayable(), isLegalMove() is still the same, just change the accessing elements of 2D array to the accessing elements of linked list using the **getData()** and **changeData()** from the previous step.
- The sliding feature in linked list is the same as deleting the blank nodes from the linked list. This part is implemented in the **deleteAndShiftBoard()** function, which follow these main ideas:
    - To delete a node, we need to get to the node right before the node that we need to delete, which is similar to the idea of accessing the node from the previous part.
    - When got to the previous node of the node that we need to delete, delete the node using the basic linked list deleting operation.
    - Just be careful of the case in which we need to delete the first node, we have to let the node after this node to be the first node, then delete the current node.
- All the previous steps are only applicable to the sliding to the left case because of the way we initialize the linked list of LinkedList. To use the same code for initializing, accessing, sliding to other three directions, we create a **levelDif()** function. This function will be called when any of the above functions are called and when any of the above functions stop.
    - The idea behind this function is rotating the linked lists to the desired direction.



Left sliding board we have initialized

- Now, let's put the above image through the mirror and we will get the below image.

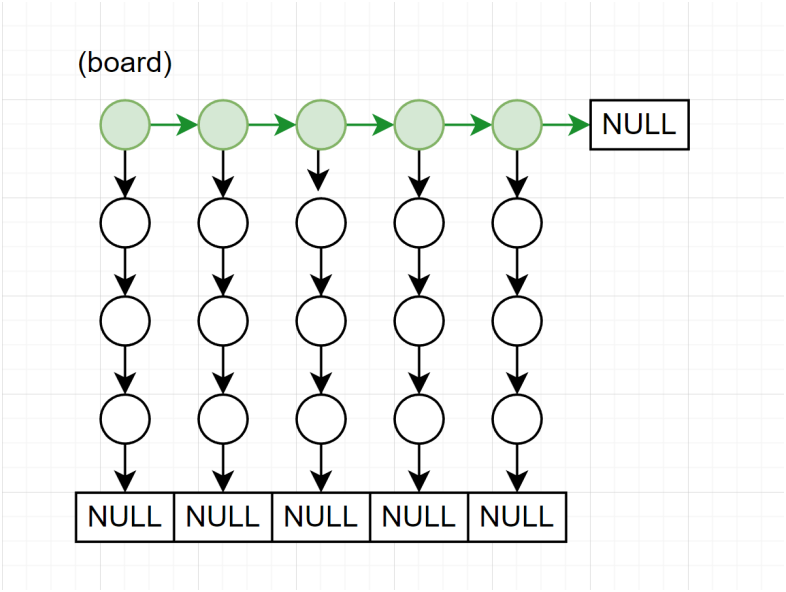


Right sliding board was created by mirroring the left sliding board

- See that it's a right sliding board that we needed? To achieve this, every single functions that accessing the board using the (x, y) coordinate

parameters will become (x, boardLength + 1 - y). This change is made through the **levelDif()** function.

- Take a look at this up sliding board that we want to implement:



Right sliding board created by mirroring the left sliding board

- See that it's basically the same board as the left sliding board but with the board height and length swapped and when accessing an element, we will traverse the length first and then the height rather than the height first then the length like the two previous cases. From this idea, the **levelDif()** function will swap the length with the height of the board and the (x, y) coordinate for accessing cells will become (y, x).
- For the down sliding board, it's similar to the up sliding board but mirrored. From this idea, the **levelDif()** function will change the (x, y) coordinate for accessing cells to (y, boardHeight + 1 - x), after that, swap the length with the height of the board.

4.4.1.3) 2D Array Pointer/ Linked list comparison

We will compare the time complexity and space complexity of 2D array pointer and linked list.

Let M, N consecutively be the number of rows and columns of the board.

- The space complexity is easy to determine, we have $M * N$ cells so for the 2D pointer array version, it takes sizeof(int) $* M * N$ and sizeof(Node) $* M * N$ for the linked list version.

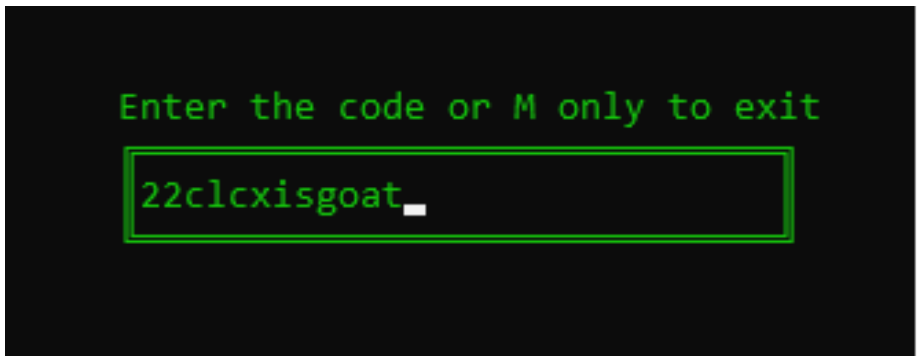
- Randomly accessing a 2D array's element is $O(1)$ because we can directly access a cell through <array>[x][y] without traversing any other cell.
- Randomly accessing a linked list of linked list's element is $O(M+N)$ because we need to move to our x^{th} linked list head and move to the y^{th} node, in the worst case, we have to traverse through M linked list head and N nodes.
- Deleting and Sliding in a 2D array is $O(M)$ or $O(N)$ because we have to shift all the elements on that row or column, in the worst case, we have to shift M or N elements.
- Deleting and Sliding in a linked list of linked lists is O(1) for deleting the node without any changes in the position of the other nodes. But to find the node that is needed to be deleted, the time complexity is $O(M+N)$ for finding that node.

	2D Pointer Array	**Linked List**
Space Complexity	sizeof(int) $*$ M $*$ N	sizeof(Node) $*$ M $*$ N
Accessing an Element	$O(1)$	$O(M+N)$
Deleting and Sliding row or column	$O(M)$ or $O(N)$ for deleting and sliding	$O(M+N)$ for finding $O(1)$ for deleting

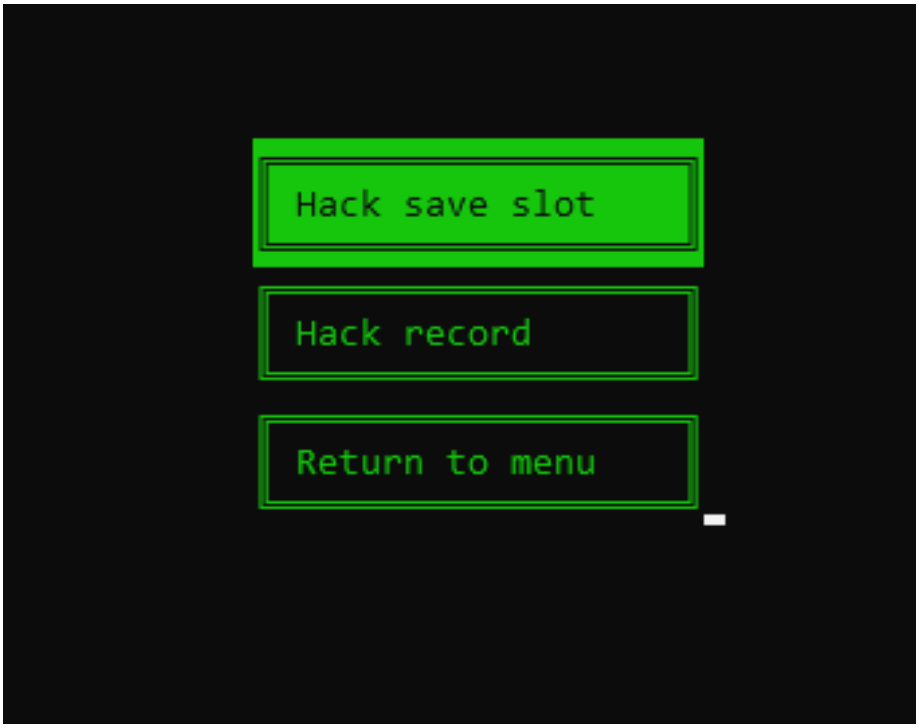
4.4.2) Save file "hacking"

Hacking save file includes two parts, reading/writing from/to the save file and changing the content of the save file. For reading/writing from/to save file and the meanings of all the users' data, check 4.3.5 (user accounts) for more details

To hack your own account's states and records, login to your account first. When in the menu, press '/' to open hacking mode. Type the code "22clcxisgoat" to start hacking.



Typing the **secret** code



Select hacking mode

The player can choose to hack their own records or their saved games' states. These are implemented in the **hackState()** and **hackRecord()** function:

- **HackState()**: This function allows user to choose a save slot and modify it's level, points, remaining time and date. This function will also update the records and/or the leaderboard if needed.
- **HackRecord()**: This function allows user to choose a record and modify it's points and date. This function will also update the leaderboard if needed.

Now, we will cover how were we the first team to detect and report back the missing 500 bytes NULL and the masking problem with the "sample.bin" file.

- The binary file reading part is easy, we just need to follow the predefined structures to read the file.
- The harder part is XORing the mask with all the characters because we don't know the XORing format, it could either be XOR all of the unused bytes or just XOR the content of the character array. So we open the "sample.bin" file in hexadecimal mode and we saw that XORing the content of the character array is enough. However, we also noticed that by the nature

of the XOR operator, if we XOR the mask with other identical characters, it would gives '\0' (00 in 8-bit representation in hexadecimal form) which is fatal if we want to store other data that could contain the same characters as the mask.
- The story about the discovery of the missing 500 bytes NULL is that we followed the predefined structures but we still coudn't read the "sample.bin" file. We decided to write the structures to our files and compare it with the sample file (right click the file -> properties -> see the size of the file). We found out that we are exactly 500 bytes larger than the sample file so we thought the teacher had forgotten to write it down to the file. We tested and it came out true.
- We also reported both of these problems to our teachers, who would probably give us some extra points for this discovery.

4.5) Other features

4.5.1) Shuffling when out of moves

We implemented this extra feature because this feature is in the original game, also with large board size, if we let players lose when there are no legal moves left, the game becomes incredibly hard to win. Shuffling is implemented in the **shuffle()** function.

We use the isPlayable() function in the 4.2.3 Game finish verification to check if there are any legal moves left. If not, we will shuffle the content of the not-blank cells. This is the idea of our shuffling algorithm:
- Iterate through every cell, put all the contents of the not-blank cells to a vector.
- Iterate through every cell again, this time, for each cell, do these steps:
  - Get a random index number between 0 and size of the vector in the previous step.
  - Assign the vector element with the index from the previous step to the cell.
  - Remove that vector element from the vector, hence reducing the size of vector by one.

4.5.2) Game timer

We minupulate time in this game in a simple way. As we know that the statement **Sleep(x)** pauses the program for **x** miliseconds, we create these variables for our timer:
- **timeRemain**: This is how much time our players have to clear a level in our game. When it comes to 0, players fail, and if they clear all cell before

it reaches 0, they gain bonus points which are 50 points for each second remained.

- **streakTimeRemain**: This is set to 45 seconds every time a match is found and reduced alongside with **timeRemain**. When it comes to 0, the **streak** (mentioned in 4.3.6) resets.
- **miliseconds**: We let this variable be 1000, used to check whether a second goes by, if this variable goes to 0, it is set to 1000 again and **timeRemain** is reduced by 1.
- **tick**: This how much we would put as **x** in the **Sleep(x)** statement above. **tick** in our game is set to 20, which makes the game run at approximately 50fps (in one second, there are 50 **ticks**). After every **tick**, the **miliseconds** is redude by **tick** miliseconds.

5) References

- **Special thanks** to Nguyen Anh Kiet (22127220), Nguyen Le Thanh Duy (22127474) for giving feedbacks, discussing and giving us some ideas (We borrow **clearScreen()** function from them). They also test the game, check for grammars errors in the report and give us a tremendous amount of emotional support.
- Source code:
  - "https://www.learncpp.com/cpp-tutorial/generating-random-numbers-using-mersenne-twister/)" (for better random numbers generation)
  - "https://stackoverflow.com/questions/997946/how-to-get-current-time-and-date-in-c" (for getting current time and date)
  - "https://www.geeksforgeeks.org/how-to-use-gotoxy-in-codeblocks/" (-source for **gotoxy()** function)
  - "https://stackoverflow.com/questions/51344985/how-to-center-output-console-window-in-c" (source for **resizeWindow()**)
  - "https://www.daniweb.com/programming/software-development/code/216345/add-a-little-color-to-your-console-text" (source for code of **SetColor()**)
  - "https://theasciicode.com.ar/" (ASCII Table - Where we get the characters for drawing purposes)
- "https://www.geeksforgeeks.org/how-to-print-colored-text-in-c/" (ColorID table for **SetColor()**)
- Sounds:
  - "https://www.youtube.com/watch?v=VFdenS6y_Z4" (Polish Cow Dancing But It's a 16-Bit Remix)
  - "https://www.youtube.com/watch?v=TksTj3h3IfY" (Among Us Drip Theme [8-Bit Remix])

- “https://www.youtube.com/watch?v=WzhZz59TfZk” (2023 LCK trên nền Nhạc “ Tấm Lòng Son remix” - EDIT By Duy2503)
- “https://www.youtube.com/watch?v=tz82xbLvK_k&list=RDQMwZTf6RwxzHA&index=8” (Undertale Ost: 087 - Hopes and Dreams)
- “https://www.youtube.com/watch?v=cKUeoybpGZo&list=RDQMwZTf6RwxzHA&index=15” (Undertale Omega Flowey Finale Theme)

- Images:
  - “https://www.pinterest.com/” (Pinterest)
  - “https://knowyourmeme.com/” (Know your meme)
  - “https://www.text-image.com/convert/result.cgi” (Image to ASCII Art converter)