



PROJET IF2

Sommaire

[Objectif](#)

[Choix réalisés](#)

[Structure du code](#)

[dataType.h](#)

[grid.h, grid.c](#)

[userInput.h, userInput.c](#)

[menu.h, menu.c](#)

[game.h, game.c](#)

[score.h, score.c](#)

[main.c](#)

[Améliorations possible](#)

[Sources](#)

Objectif

Ce projet a pour but de réaliser un jeu de brique inspiré de Tetris:

- Sur une grille 2D de taille allant de 8x4 à 12x6, des lettres A, O, H, X sont présentes (une lettre par case de la grille)
- L'objectif est de permuter des lettres 2 par 2, afin de créer des chaînes de 4 lettres identiques se touchant verticalement ou horizontalement

Choix réalisés

1. Tout d'abord, nous avons choisi d'utiliser la dernière version de langage C, C23 car il est primordial d'être à jour pour éviter l'introduction de bogues informatiques qui serait liée à une défaillance du langage et pour profiter des améliorations qui sont fournies.
2. Puis nous nous sommes questionnés sur le type d'interface à réaliser. Nous avions le choix entre:
 - un programme **GUI** (Graphic User Interface)
 - un programme **CLI** (Command Line Interface)

Nous avons opté pour un programme **CLI** car il ne nécessite pas la prise en main d'une bibliothèque externe ou d'un Framework. Cependant, l'un des désavantages d'un programme **CLI** est la portabilité du code. Contrairement à un Framework graphique qui fait de l'abstraction de code afin de pouvoir compiler sous les systèmes d'exploitation les plus connus, l'intégration dans la console dépend directement et très fortement du système d'exploitation. Nous avons fait le choix de perdre la portabilité du code pour avoir une meilleure intégration dans la console de Windows.

Structure du code

Au lancement du jeu, la fonction **askUsername** est appelée, elle récupère le nom du joueur et le passe à la fonction récursive **startMenu**.

En fonction de la saisie utilisateur le menu principal va appeler

- **newGameMenu**, suite à une autre saisie de l'utilisateur permettant d'obtenir le mode de jeu, si celle-ci est "PUZZLE" ou "RUSH" alors **gridMenu** est appelée. grille est générée à partir des données du joueur puis en fonction du mode de jeu, les fonctions **puzzleGame** ou **rushGame** sont appelée et le score en fin de partie est écrit dans le fichier des scores avec **writeScore**. Enfin, le menu de fin de partie est affiché avec **printGameOver** et il est proposé au joueur de revenir au menu principal.
- **bestScoreMenu**, une fonction récursive également. Dépendant d'une saisie utilisateur, si le joueur souhaite consulter un score selon le nom, **askUserScoreByNameMenu** est appelé, le score est affiché si celui ci existe puis **bestScoreMenu** est rappelée.

De même, si le joueur souhaite consulter les scores selon les tailles de grilles, la fonction **askUserScoreByGridSize** est appelée puis **bestScoreMenu** est exécutée à nouveau.

Si la saisie correspond à un retour au menu principal, alors nous quittons la fonction.

- **printRules**, affichera les règles du jeu et proposera un retour au menu principal

Si le joueur souhaite quitter le jeu alors on sort de la fonction **startMenu**.

dataType.h

Afin d'obtenir un code modulaire, réutilisable et facilement compréhensible, nous avons créé le fichier *dataType.h* dont l'objectif est de regrouper les structures de données usuelles et les énumérations que nous avons défini tel que:

```
typedef enum {
    PUZZLE,
    RUSH,
    NONE,
} GameMode;
```

```
typedef struct {
    size_t row;
    size_t col;
} Coordinate;
```

```
typedef enum : size_t {
    SUCCESS = 0,
    GENERIC_ERROR,
    GRID_SIZE_ERROR,
} ErrorCode;
```

Mais aussi à stocker les constantes importantes au déroulement du jeu comme la taille maximale/minimale de la grille, le nombre de joueurs maximal ou encore l'intervalle de temps du mode rush :

```
constexpr size_t maxRow = 6;
constexpr size_t minRow = 4;
constexpr char Letter[4] = "HOAX";
```

```
constexpr size_t maxCol = 12;
constexpr size_t minCol = 8;
constexpr size_t MaxPlayers = 256;
```

```
constexpr size_t rushModeTimeInterval_s = 15;
constexpr size_t longestSequencePossible = maxCol * maxRow;
constexpr size_t maxSequencePossible = longestSequencePossible / 2;
```

Nous utilisons le mot-clé **constexpr** propre au C23 qui permet de déclarer une constante au moment de la compilation et non de l'exécution comme le fait le mot-clé **const**. Cette nouveauté permet de se substituer au **#define** classiquement utilisé dans le C. Les points positifs à l'utilisation de **constexpr** sont :

- L'indication du type de la constante → *int, float, etc...*

Cela rend le code plus lisible puisque le type permet d'avoir plus d'indications sur la nature de la constante contrairement à un **#define** qui définit une valeur avec pour seule indication son nom.

- L'impossibilité d'une conversion vers un type non constant contrairement au **const**.

Une constante en C peut être modifiée via des conversions et des pointeurs comme décrit dans l'exemple de code ci-dessous.

```
const int x = 10;
int *p = (int *)&x; // conversion forcée
*p = 20; // Modification de la valeur
```

Avec une constante **constexpr**, ceci n'est pas possible, ce qui nous garantit l'immutabilité des données.

- Une variable déclarée **constexpr** peut servir à initialiser la taille d'un tableau statique.

grid.h, grid.c

Concernant la création de la grille nous avons opté pour une grille allouée dynamiquement en fonction de l'entrée utilisateur. Cependant, nous sommes revenus sur notre choix et avons décidé d'utiliser une grille allouée statiquement avec les dimensions maximales souhaitées de 12×6 .

Cela est plus avantageux, car les grilles étant relativement petites, l'espace occupé par une grille de taille maximal reste faible à l'échelle de l'espace disponible sur un ordinateur. Nous évitons donc les problèmes liés à l'allocation dynamique qui peuvent être causés par:

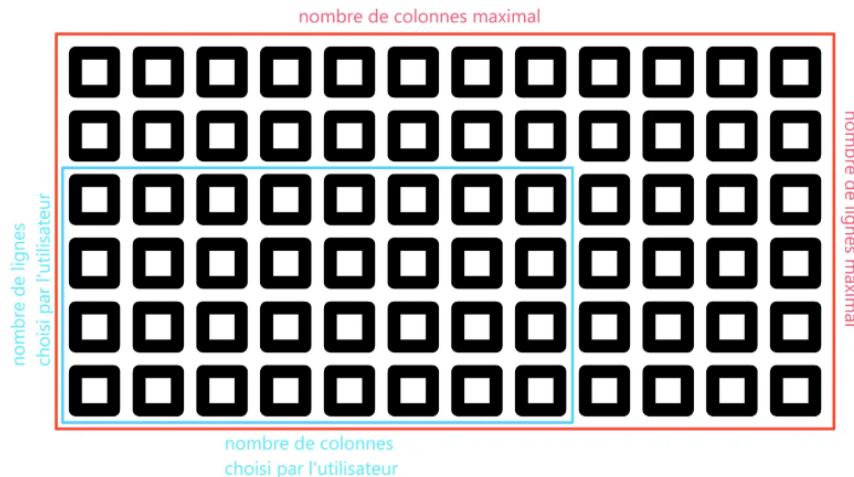
- **le développeur** → En effet, toute allocation doit être accompagnée d'une libération des données, mais un oubli de la part du développeur peut entraîner un remplissage de la **RAM** (Random Access Memory) ce qui causera un dysfonctionnement de l'ordinateur du type **BSOD** (Blue Screen Of Death).
- **la machine** → Ce problème est peu courant, mais il arrive que l'allocation échoue.

Toutes ces possibilités d'erreur doivent être prises en compte ce qui rendrait le code moins lisible dû aux nombreuses structures conditionnelles nécessaire au contrôle d'intégrité des données et la prise en charge des exceptions telles qu'un pointeur non alloué.

Tandis qu'avec une grille de taille fixe, nous associons dans une structure deux paramètres correspondant au nombre de lignes et de colonnes souhaité, nous permettant de définir une "sous" grille.

```
//dataType.h

typedef struct {
    const size_t columns;
    const size_t rows;
    char data[maxRow][maxCol];
} Grid;
```



Ensuite, pour avoir un code modulaire et répondant le mieux possible aux besoins des deux modes de jeu, nous avons appliqué la méthode suivante, "chaque fonction doit effectuer une seule et unique tâche".

Nous pouvons donc en appelant les fonctions suivantes:

- `gridIsEmptyBox` ⇒ vérifier si une case de la grille est vide
- `gridIsValidCoordinate` ⇒ vérifier si des coordonnées sont dans la grille ou en dehors
- `coordEquals` ⇒ vérifier si deux coordonnées sont égales
- `isNeighbour` ⇒ vérifier si deux cases sont adjacentes
- `gridFill` ⇒ remplir la grille aléatoirement avec les caractères A, O, H, X
- `gridPrint` ⇒ afficher la grille
- `gridEmptyBox` ⇒ vider une case de la grille
- `gridSwapBoxes` ⇒ intervertir deux cases de la grille
- `gridFallElement` ⇒ fait tomber les éléments vers le bas de la grille

Pour effectuer la suppression des séquences de lettres nous avons créé un type de données pour conserver les coordonnées des éléments de la séquence ainsi que leur longueur et un indicateur permettant de savoir si la séquence a une longueur minimale de 4 ou non.

```
typedef struct {
    Coordinate data[longestSequencePossible];
    size_t length;
    bool empty;
} Sequence;
```

Nous avons essayé tout d'abord d'élaborer nos propres algorithmes. Cependant, il n'étaient pas concluants. En se questionnant sur le problème et en prenant un cas particulier pour simuler le fonctionnement de l'algorithme de recherche, nous avons supposé une séquence se divisant en deux branches. Il faudrait donc récupérer la première branche, puis la seconde, ensuite, nous étendrions la méthode à n branches.

Cette méthode correspond à un algorithme DFS (Deep First Search) utilisé pour parcourir les graphes. L'algorithme consiste à parcourir le graphe et lorsqu'il arrive à une intersection, il garde en mémoire le nœud de l'intersection, et parcours la branche la plus profonde. Quand la première branche est complètement explorée, il revient à l'intersection et parcourt les autres branches, tout ceci de manière récursive. Avec cette méthode, nous avons pu implémenter la fonction suivante:

```
Sequence getLongestSequences(const Grid *grid)
```

Elle renvoie la séquence la plus longue de la grille. Cette séquence est supprimée par la suite par les fonctions `removeLongestSequences` et `gridRemoveSeqWithScore`

userInput.h, userInput.c

Ces fichiers regroupent les fonctions servant à l'acquisition des données.

Dans un premier temps, nous avons créé la fonction suivante.

```
ErrorCode strToCoord(const char in_coord[2], Coordinate *coordinate)
```

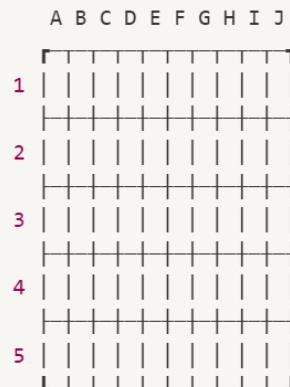
Elle a pour rôle de convertir des coordonnées textuelles en coordonnées sous forme de structure.

L'entrée se présente sous la forme:

- A1 ou 1A
- a1 ou 1a

Nous identifions les cases à la manière d'un jeu d'échec pour que cela soit plus intuitif pour le joueur.

Si la conversion est possible, elle l'effectuera puis renverra le code d'erreur **SUCCESS** sinon **GENERIC_ERROR**



Puis, dans un second temps, nous avons créé la fonction

```
Coordinate getInputCoord(const char *msg)
```

Elle récupère l'entrée utilisateur et vérifie si celle-ci est valide. Nous avions commencé l'acquisition des données avec **scanf** cependant nous voulions récupérer exactement deux caractères.

Nous avons donc opté pour la fonction **getchar** qui récupère le premier caractère du tampon d'entrée. Grâce à celle-ci nous récupérons nos deux caractères et vérifions si le troisième est une fin de ligne. Cela confirmera qu'il y a bien uniquement deux caractères dans le tampon. Dans le cas contraire, nous vidons le tampon et redemandons à l'utilisateur de saisir des données exactes.

Suite à cette première vérification, nous cherchons à déterminer si les coordonnées rentrées sont valides, nous allons donc appeler **strToCoord**, tant que le code d'erreur retourné est **GENERIC_ERROR**, et redemander la saisie.

```
void secureGetCase1(const Grid *grid, Coordinate *coord)
```

Cette fonction appelle **getInputCoord** et vérifie si la coordonnée est effectivement dans la grille, si ce n'est pas le cas la saisie sera redemandée.

```
bool secureGetCase2(Grid grid, const Coordinate coord1, Coordinate *coord2);
```

`secureGetCase2` a un fonctionnement similaire à `secureGetCase1`, mais elle vérifie en plus si la deuxième coordonnée est voisine de la première.

Les trois fonctions ci-dessus étaient suffisantes pour le fonctionnement du mode *Puzzle* car il s'agit d'un mode de jeu du type tour par tour, ainsi bloquer l'exécution du programme en attente d'une entrée utilisateur n'est pas dérangeant. Cependant, ces fonctions ne sont pas utilisables dans le mode *Rush* car nous devons être en capacité d'interrompre l'entrée utilisateur et de décompter 15 secondes en parallèle.

Notre première approche était le multithreading couplé à un design pattern d'observer, l'objectif était d'avoir un thread dédié aux saisies de l'utilisateur et un second qui exécuterait la poussée des éléments vers le haut toutes les 15 secondes. La communication entre les deux threads seraient faites par un système de notification propre au design pattern de l'observer. Cela se rapprocherait de la programmation évènementielle qui permet de déclencher une action lorsqu'un évènement se produit.

Après de multiples essais, le multithreading n'était pas concluant, l'exécution du programme était interrompue malgré tout. Nous avons donc consulté la documentation du C concernant les fonctions `scanf`, `getchar`, `fget` et celle-ci précisait que l'accès au tampon d'entrée standard `stdin` bloque l'exécution sur tous les threads car ce tampon est partagé.

Dans le multi threading, pour garantir l'intégrité des données partagées entre les threads, il faut utiliser des **mutex** (mutual exclusion = exclusion mutuelle) qui empêche l'accès à une donnée par les autres threads si celle-ci subit une modification par un thread tiers. C'est exactement cette mécanique qui bloque l'accès au tampon `stdin` et qui met en pause les autres threads.

Pour palier à ce problème nous avons décidé de ne pas lire l'entrée utilisateur depuis ce tampon, mais plutôt de récupérer les événements du clavier. D'où la fonction suivante:

```
bool getKeyboardInput(char *input)
```

Ainsi en détectant quelle touche est appuyée et en enregistrant le caractère associé dans un tampon, nous avons trouvé une méthode d'acquisition non bloquante.

Suite à cela, nous implementons la fonction

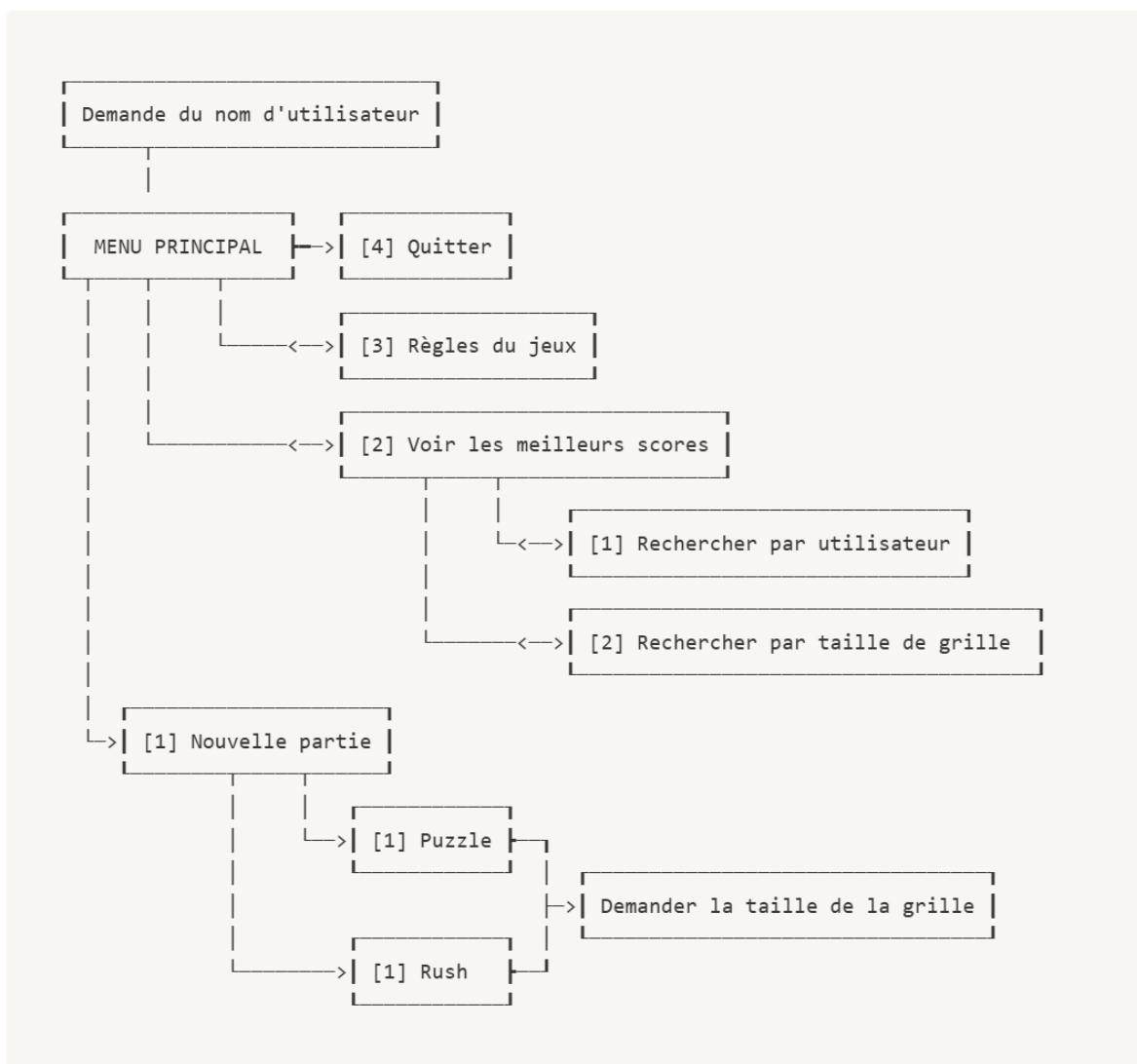
```
bool nonBlockingSecureGet(const Grid *grid, char input[2],  
                           Coordinate *coord, const char *msg)
```

Son fonctionnement est similaire à `secureGetCase1` mais avec une lecture de l'entrée non bloquante

menu.h, menu.c

Ces fichiers regroupent les fonctions liées à l'affichage et l'interaction avec les menus.

La hiérarchie des menus est comme ceci:



Tout d'abord, nous demandons le nom d'utilisateur au début du jeu avec la fonction

```
void askUsername(char* username)
```

Ensuite, nous avons implémenté des fonctions dédiées à l'affichage des menus:

```
void printMainMenu()  
void printNewGameMenu()  
void printRules()
```

```
void printGridMenu()  
void printGameOver()  
void printBestScore()
```

Elles sont utilisées dans la fonction `startMenu` qui appelle sa fonction d'affichage associé et demande une entrée utilisateur. En fonction de l'entrée, elle renvoie l'utilisateur vers le menu adéquat.

Concernant le menu d'affichage des scores, pour ranger les scores dans l'ordre décroissant, la fonction `sortPlayerScore` a été implémentée.

```
void sortPlayerScore()
```

Le corps de cette fonction se compose de l'appel de la fonction `qsort` faisant partie de la bibliothèque standard du C. Nous définissons en parallèle la fonction `compare`, elle a pour but d'indiquer à la fonction `qsort` sur quel critère trier les données. Dans ce cas précis, on trie notre liste de structures en fonction des scores dans l'ordre décroissant.

Dans le menu principal, le choix d'une nouvelle partie appelle la fonction `newGameMenu` qui permettra de choisir le mode de jeu souhaité et de récupérer les données relatives aux dimensions de la grille afin de les transférer aux fonctions associées aux modes de jeu.

game.h, game.c

Ces fichiers contiennent les fonctions principales des modes de jeu tel que

```
size_t puzzleGame(Grid *grid)
```

Cette fonction exécute le jeu en mode Puzzle. L'obtention de la saisie utilisateur est bloquante, mais cela ne perturbe pas le déroulement de la partie étant donné que ce mode de jeu est du tour par tour sans aucune limite de temps. La fin de la partie est déclenchée lorsque plus aucune permutation ne permet la création d'un séquence viable.

```
size_t rushGame(Grid *grid)
```

Cette fonction exécute le jeu en mode Rush. L'obtention de la saisie est non bloquante afin de ne pas perturber l'actualisation de la grille toutes les 15 secondes. La fin de la partie est déclenchée lorsqu'un élément est poussé hors de la grille.

```
bool gameOver(Grid grid)
```

Pour le mode puzzle la fonction `gameOver` vérifie si aucune permutation n'est possible elle renvoie `true` sinon `false`.

```
size_t evaluateScore(const size_t X, const size_t N)
```

Nous déterminons les points engrangés dans cette fonction avec l'expression mathématique suivante.

$$(N + 1)(X - 1)^2$$

Avec N le nombre de séquences préalablement supprimées et X la longueur de la séquence supprimée.

score.h, score.c

Ce couple de fichiers regroupe les fonctions relatives à la lecture, écriture et modification du score dans un fichier.

La modification des données dans le fichier serait une tâche trop délicate, nous avons donc préféré récupérer l'ensemble du fichier dans un tableau de structure.

L'écriture des scores dans un fichier nous a poussés à nous questionner encore une fois sur les formats de fichier à utiliser. Nous avions pensé à un fichier **CSV** ou **JSON** étant donné qu'ils sont spécialement fait pour le stockage d'informations et qu'il existe déjà des bibliothèques permettant leur utilisation. Mais compte tenu de la complexité de la tâche à réaliser, utiliser une bibliothèque ne sera pas pertinent.

Nous avons essayé de lire et d'écrire dans le fichier avec une méthode naïve de parcours. Les problèmes rencontrés étaient:

L'utilisation de délimiteur comme dans un fichier CSV:

- Elle nécessite d'empêcher l'utilisateur d'utiliser le caractère prévu comme délimiteur.
- D'itérer sur tout le fichier et vérifier chaque caractère
- Mais aussi de connaître la position du curseur dans le fichier

L'utilisation de paire "clé valeur" comme dans un JSON nécessite une analyse approfondie du fichier.

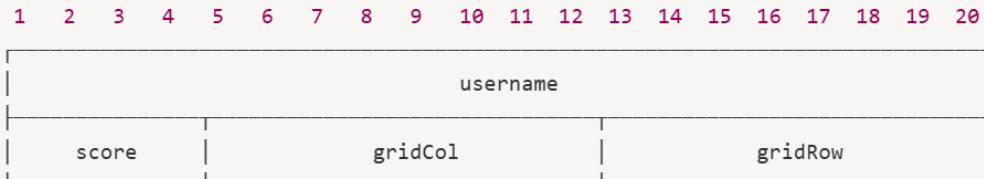
Cela représente trop de paramètres à prendre en compte, ce qui complexifierait le code et l'ensemble du projet.

Pour palier à cette difficulté, nous avons procédé à la création de notre propre format de fichier:

Le **.42** dont le nom est complètement fantaisiste en référence à la réponse à la grande question sur la vie

Inspiré des fichiers WAV et JPG. Nous représentons les informations du joueur sous forme de chunk de données de taille prédéfinie et contiguë.

```
typedef struct __attribute__((__packed__)) {
    char username[20]; // 160 bits = 20 bytes
    unsigned int score; // 32 bits = 4 bytes
    size_t gridCol; // 64 bits = 8 bytes
    size_t gridRow; // 64 bits = 8 bytes
} UserScore;
```



De cette manière, nous n'avons pas à vérifier la présence ou non d'un délimiteur. Chaque donnée à un espace qui lui est propre et de même taille pour chacun. La lecture et écriture dans le fichier se voient également simplifiées puisque au lieu de parcourir le fichier caractère par caractère, nous récupérons la totalité du fichier au début du programme avec la fonction

```
void readUserScore()
```

l'ensemble du fichier va être stocké dans un tableau de structure

```
UserScore user_score[MaxPlayers]
```

Grâce à cette méthode, nous pouvons aisément accéder au n-ième joueur ainsi qu'à ses données.

Lorsque nous souhaitons modifier ou ajouter le score d'un joueur, nous vérifierons si ce joueur est déjà présent dans le fichier avec la fonction **userExist**.

- Si c'est présentement le cas alors son indice dans le tableau **user_score** est récupéré puis sa valeur est mise à jour avec la fonction **setUserScore** dont l'objectif est de retenir la valeur maximale entre le score actuel du joueur et son meilleur score
- Sinon, nous récupérons le premier emplacement vide dans le tableau avec **newUsernameIndex** et enregistrons ses données

```
int userExist(const char* username)
void setUserScore(const UserScore usr_score)
int newUsernameIndex()
```

Enfin, lorsqu'une partie est terminée nous appelons la fonction **writeScore** où nous écrivons dans le fichier par blocs de 40 octets les scores du joueur.

main.c

Ce fichier contient la fonction principale de tout programme C : Le **main**. Cette fonction appelle les autres fonctions nécessaires à l'exécution du jeu.

Améliorations possible

Afin de perfectionner ce projet, le code pourrait être adapté à une interface graphique, ainsi que des ajouts sonores comme des bruitages lors de la suppression des séquences et une musique en fond. Ce projet pourrait aussi voir sa grille améliorée avec cette fois-ci une grille en 3D formant un cube permettant des combinaisons sur trois axes.

Sources

constexpr specifier (since C23) - cppreference.com

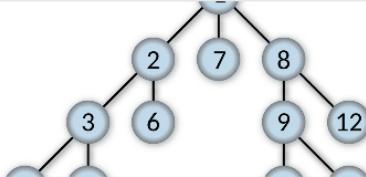
A scalar object declared with the `constexpr` storage-class specifier is a constant. It must be fully and explicitly initialized according to the static initialization rules. It still has linkage appropriate to its declaration and it exists at runtime to have its

🔗 <https://en.cppreference.com/w/c/language/constexpr>

Depth-first search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as

w https://en.wikipedia.org/wiki/Depth-first_search



How can I prevent scanf() to wait forever for an input character?

I want to fulfill the following things in a console application: If user inputs a character, the application will do the corresponding task. For example, if user

🔗 <https://stackoverflow.com/questions/21197977/how-can-i-prevent-scanf-t...>



_kbhit

En savoir plus sur les alertes suivantes : _kbhit

Microsoft Learn  <https://learn.microsoft.com/fr-fr/cpp/c-runtime-library/reference/kbhit?view=msvc-170>

_getch, _getwch

Informations de référence sur l'API pour les `_getch` et les `_getwch` ; qui obtiennent un caractère à partir de la console sans écho.

Microsoft Learn  <https://learn.microsoft.com/fr-fr/cpp/c-runtime-library/reference/getch-g...>

Wave File Specifications

File Description: WAVE or RIFF WAVE sound file File Extension: Commonly .wav, sometimes .wave File Byte Order: Little-endian

<https://www.mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html>

Packed Structures (GNU C Language Manual)

GNU  https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Packed-Structures.html