Rapport C++ - Sudoku

Alexandre Berlia ; Gildas Montcel ; Théophile Roulland Durande

I/ Description de notre programme et comment le faire fonctionner.

Ce programme permet de jouer au sudoku au sein du terminal. Nous avons sélectionné quatre grilles de difficultés croissantes, parmi lesquelles l'utilisateur peut faire son choix.
Au cours du jeu, l'utilisateur a plusieurs possibilités. Il peut d'abord entrer une valeur par la combinaison "ligne colonne valeur" (exemple : 5 3 8). Le programme s'assurera que la valeur est correcte et l'affichera en bleu auquel cas. L'utilisateur peut également demander de l'aide pour remplir la case de son choix en écrivant "a ligne colonne" (exemple : a 3 8). La case sera alors remplie en vert. Enfin, il peut quitter le jeu en appuyant sur 'q'.

Le programme ne nécessite pas d'implémentations particulières pour le faire fonctionner, hormis une extension pour lire un document pdf.

Le projet est constitué d'un fichier header pour la déclaration des variables et des fonctions, ainsi que d'un fichier main, qui contient la fonction d'affichage des grilles, le constructeur de la classe sudoku, les grilles et la boucle principale du jeu.

II/ L'architecture plus détaillée de notre programme.

Le programme est organisé en trois parties : la classe Sudoku, la fonction d'affichage de la grille, et la boucle principale.

La classe Sudoku distingue trois sortes de grilles : 'originale' garde en mémoire les cases fixes, qui ne peuvent être modifiées ; tandis que 'grille' représente l'état actuel du Sudoku et que 'solution' permettra la vérification des combinaisons entrées par l'utilisateur.
Trois booléens sont associés au Sudoku : checkAndSet compare la valeur entrée par l'utilisateur à la valeur de la grille de solution. provideHint permet de faire suite à la demande d'aide de l'utilisateur. estComplete, enfin, permet de mettre fin au jeu quand la grille est complète.

La fonction d'affichage s'occupe d'afficher les bordures du sudoku, ainsi que les valeurs déjà remplies de la grille, selon le code couleur suivant : en blanc pour les cases de départ, en bleu pour les cases remplies par l'utilisateur, et en vert pour celles remplies par la machine.

La boucle principale du jeu laisse à l'utilisateur le choix de remplir une case de Sudoku, de demander de l'aide ou de quitter le jeu. Le programme lit la commande entrée et met à jour le sudoku en conséquence (rien si la valeur est incorrecte, ajoute la valeur en bleu ou en vert si bon). Enfin, la boucle regarde si la grille est complétée, ce qui mettrait fin au jeu.

III/ Les problèmes rencontrés et leurs solutions.

Le tout premier obstacle fut d'abord organisationnel. Certains membres du groupe n'avaient jamais utilisé GitHub ou un Codespace sur VS Code auparavant. Nous avons donc pris un temps méthodologique pour être sûrs que tout le monde sache commit and push, vérifie sa synchronisation avec les autres avant de coder en local, et la répartition du travail. La communication a été la clé ici.

Petite liste de problèmes et de solutions adoptées :

- userMark and solverMark : on voulait avoir des couleurs différentes pour les cases rentrées par l'utilisateur et par la machine. Nous avions essayé d'utiliser des appels de fonction au sein du main, mais c'était compliqué et peu lisible. Passer directement par une condition dans la fonction d'affichage avec deux booléens associés case par case a été salutaire en ce sens. Nous réutiliserons certainement de nombreuses fois cette méthode dans le futur pour lier des conditions à des objets d'une classe ! En mettant tous les booléns sur false au départ (par exemple), nous pouvons facilement activer les conditions qui nous intéressent.
- Une formulation de l'énoncé nous a mis un doute : devions-nous créer un solveur qui calcule la bonne solution pour chaque case ou une aide qui a déjà accès à la solution ? N'ayant pas réussi à implémenter la première méthode, nous avons préféré entrer directement les solutions pour chaque case.
- cin.ignore() : Le message invitant l'utilisateur à entrer une commande s'affichait toujours deux fois et nous ne comprenions pas pourquoi. En fait, le programme gardait en mémoire le retour à la ligne fait par l'utilisateur à chaque commande, qu'il interprétait comme une "non saisie" après. D'où la réapparition du message. L'ajout de cin.ignore() nous a permis de supprimer la "nouvelle ligne" et ainsi, de n'avoir qu'un message à chaque fois !

- break après if(estComplete) : pendant un test, nous nous sommes aperçus que cette condition n'était pas suffisante pour sortir de la boucle infinie. Nous avons donc décidé d'ajouter break en dernière ligne, même si cela a paru contre-intuitif au début.
- utilisation de stringstream : Au début, l'utilisateur devait faire un retour à la ligne entre partie de la combinaison (ligne, entrée, colonne, entrée, valeur), ce qui était fastidieux et posait des soucis de jouabilité, notamment si l'on entrait la mauvaise ligne. La commande était en effet traitée en trois parties par le programme. Cela nous a pris un certain temps pour implémenter le stringstream et vérifier qu'il fonctionnait bien. Pour le solveur, le premier caractère 'a' est supprimé par command.substr(1). Il faut tout de même toujours prévoir un espace entre chaque partie de la commande.